Management and Programming of Reconfigurable Hardware Resources

David Thomas

Imperial College London

Submitted for the degree Doctor of Philosophy in Computing

Abstract

Reconfigurable hardware resources, such as Field Programmable Gate Array (FPGA) based accelerator cards, are often used as an application-specific accelerator. Our approach is to make all such hardware resources available to as many applications as possible, so that all tasks on a computer can off-load computation from traditional processors onto reconfigurable hardware.

To allow us to manage varied hardware resources at run-time, we define a layered framework which abstracts implementation details away from functional specifications. This framework isolates hardware and implementation aspects of an operation from each other, and from the user of the operation. By enforcing this abstraction, it is possible to transparently provide additional services to applications, such as optimisation of data transfer and device sharing between operations. Our implementation of this layered framework is called IGOL, and using it we demonstrate acceleration of a wide variety of commodity applications such as Premiere and DirectShow, using reconfigurable computing resources such as the RC1000 and UltraSonic cards.

To support simulation applications in hardware, we develop a set of random number generation components. The first of these is a set of high quality uniform random number generators, which pass Diehard and BigCrush, the strictest tests of randomness currently available. We implement a medium quality random number source based on XorShift, which outperforms the LFSR base generators currently in general use. We provide a technique for approximating arbitrary probability distributions by using composite distributions. These approximations can be modified at run-time without recompiling hardware, and are designed to be lightweight. Using a custom search and optimisation algorithm, any target distribution can be approximated. These random number generators are also made available to software applications via the IGOL framework.

Acknowledgments

Thanks to Mum and Dad for supporting me through my student days, although I'm sure they'll be glad that I now have a "real" job.

Thanks to Max for providing me with a home through the majority of my studies, and for lots of happy hours spent watching high quality american TV series (and planning to take over the world).

The Custom Computing group have all been great, in particular Henry, Tim, Tero, Shay and Gabriel, but also everyone else who has come and gone over the years. The discussions and arguments I've had in the group have been great fun, and often helped me to see past my own stupidity.

Thanks to Roger for all the beers and curries: sorry I never produced anything particularly useful for Celoxica...

Most of all, thanks to Wayne for putting up with me over the past five years, particularly when I didn't do anything he told me to and occasionally made him stay up even later than he normally would.

Publications

- Thomas, D. B., and Luk, W.: "*A Framework for Development and Distribution of Hardware Acceleration*", in *Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Communications, Proc. SPIE*, Vol. 4867, 2002, pp. 60-70

- Luk, W., Cheung, P. Y. K., and Thomas, D., B., and Rissa, T.: "Modular Design Framework for Video Processing", in *Proc. IEE FPGA Developers Forum*, IEE, 2003

- Styles, H., Thomas, D. B., and Luk, W.: "Pipelining Designs with Loop-Carried Dependencies", in *Proc. IEEE Int. Conf. on Field-Programmable Technology*, 2004, pp. 255-262

- Thomas, D: *Management and Programming of Reconfigurable Hardware Resources*, Ph.D. Thesis, 2005

- Thomas, D. B., and Luk, W.: "High Quality Uniform Random Number Generation for Massively Parallel Simulations in FPGAs", in *Int. Conf. on Reconfigurable Computing and FPGAs*, 2005

- Thomas, D. B., and Luk, W.: "*High Quality Uniform Random Number Generation Through LUT Optimised Linear Recurrences*", accepted for *IEEE Int. Conf. on Field-Programmable Technology*, 2005

- Thomas, D. B., and Luk, W.: "*Efficient Hardware Generation of Random Variates with Arbitrary Distributions*" submitted to *IEEE Transactions on Computers*

Contents

1 Introduction **1**

1.1 Hardware Acceleration of Software Applications 4

1.2 Random Number Generation for Simulation Applications 5

2 Background **7**

2.1 System trends 7

2.2 Hardware Compilation 9

2.2.1 Traditional Hardware Description Languages 9

2.2.2 High-Level Compilers 10

2.2.3 FPGA Based Processors 11

2.3 Environments for Customised Applications 12

2.3.1 Ptolemy 12

2.3.2 System Generator 13

2.3.3 Chimera Port-Based Objects 14

2.3.4 Application-specific Acceleration using FPGAs 15

2.4 Random Number Generation 16

2.4.1 Uniform Random Numbers 17

2.4.2 Non-Uniform Random Numbers 18

2.4.3 Hardware Random Number Generation 23

2.5 Summary 28

3 Analysis and Design **29**

3.1 Analysis of Problem Domain 29

3.1.1 Target Applications 29

3.1.2 Hardware-Software Communication Models 35

3.1.3 Requirements 38

3.2 Framework Design 39

3.2.1 Units of Abstraction 40

3.2.2 Modelling Components 44

3.3 Summary 49

4 IGOL: Implementation and Services **50**
4.1 Framework Implementation 50
4.1.1 Common Interfaces 52
4.1.2 Registry 54
4.1.3 Activator 57
4.2 Plug-In Support 60
4.2.1 Applications Supported by IGOL 61
4.2.2 Characteristics of Successfully Accelerated Applications 67
4.2.3 Summary of Plug-Ins 68
4.3 Device Support 68
4.3.1 Unified Interface 69
4.3.2 Real Devices 74
4.3.3 Simulated Devices 80
4.4 Summary 81

5 IGOL Performance **83**
5.1 Introduction 83
5.2 Test Setup 84
5.3 Basic Filter Performance 85
5.4 Segmented Processing 93
5.5 Device Sharing 95
5.6 Summary 102

6 High Quality Uniform Random Number Generation **104**
6.1 Uniform Variate Sources 105
6.1.1 Testing Statistical Randomness 105
6.1.2 Linear Feedback Shift Registers 108
6.1.3 LFSR Performance 110
6.2 Xorshift 111
6.3 Lagged Fibonacci Generators 114
6.3.1 Basic Algorithm 115
6.3.2 Hardware Considerations 119
6.3.3 Increasing the Tap Count 121
6.3.4 Selecting Good Generator Parameters 124
6.3.5 Evaluation 127
6.4 Summary 130

7 Approximating Arbitrary Probability Distributions **132**
7.1 Hardware Algorithm 133
7.1.1 Hardware Architecture 134
7.1.2 Alternate Component Distributions 135

7.1.3 Algorithm Extensions 137
7.2 Approximation of Target Distributions 139
7.2.1 Approximation Algorithm 140
7.3 Results 146
7.3.1 Hardware Performance 146
7.3.2 Approximation Quality 149
7.3.3 Comparison with Alternate Methods 152
7.3.4 Integration with IGOL 154
7.4 Summary 156

8 Conclusion **157**
8.1 Future Work 159
8.1.1 Data Processors 159
8.1.2 Random Numbers 161
8.1.3 Distribution Approximation 161

A IGOL Operation Specification Schema **181**
A.1 Operation Specification Syntax 181
A.2 Specification Schema 182
A.2.1 Specification Relationships 183
A.2.2 Port Structure 183
A.2.3 Functionality 183
A.2.4 General Purpose 184
A.3 Specification Extension and Restriction Mechanism 185
A.4 Summary 187

List of Figures

1.1 Illustration of the dependencies between the chapters of this thesis. 3

2.1 The Ziggurat method 22
2.2 Non-uniform partion of the function $\sqrt{-2\log x}$ over the range 0..1. The range is split into ten equal sections, then within the leftmost section a further ten division are created and so on. 25
2.3 The same non-uniform partion shown in Figure 2.2, but using a log scale for the input, showing the recursive sub-division that provides finer steps as zero is approached. 26
2.4 Coefficient selection circuit for non-uniform range segmentation of Gaussian generator. 26

3.1 Conceptual model of data processing 31
3.2 Comparison of convolution filter using software and hardware platforms with using generic and parameter specific versions. 35
3.3 Relationship between applications and the components in the framework. . 41
3.4 Sub-typing of Operation Specifications 42

4.1 Overview of the main components in the IGOL framework 51
4.2 Diagram of relative interface complexity at the levels between applications, Operation Instances, and Operation Implementations 54
4.3 Data stores used by the IGOL registry 56
4.4 Screenshot of an IGOL component within the DirectShow GraphEdit application 63
4.5 Image processing in a web page using the Direct-X Transform plug-in . . . 64
4.6 Applying image processing to a live web page using the Direct-X Transform plug-in 64
4.7 Data-flows within the UltraSonic backplane card. 78
4.8 Data-flow within a single UltraSonic Pipe. 79
4.9 Mapping the UltraSonic to abstract devices 80

5.1 DirectShow software processing rates 85

5.2 Software processing frame rate relative to no processing frame rate 87
5.3 RC1000 frame rate relative to software frame rate 88
5.4 UltraSonic frame rate relative to software frame rate 89
5.5 Software and hardware frame rates under DirectShow for 320x240 video . . 90
5.6 Processing speed using RC1000 for varying clock rates 91
5.7 Breakdown of per-frame time for real and emulated RC1000 using 640x360 frames in DirectShow 92
5.8 Breakdown of per-frame time for emulated RC1000 at different bus speeds 92
5.9 Frame rate for UltraSonic filter with different numbers of segments in DirectShow 93
5.10 Frame rate for RC1000 filter with different numbers of segments in DirectShow 94
5.11 Speedup for RC1000 filter at different speeds with different numbers of segments in DirectShow 95
5.12 Frame rate for segmented Median[3] on RC1000 using 640x360 frames in DirectShow 96
5.13 Frame rate for chained filters on the UltraSonic using DirectShow 97
5.14 Frame rate for different RC1000 sharing strategies in DirectShow at 640x360 99
5.15 Frame rate for chained filters using software and hardware in DirectShow (640x360) 101
5.16 Frame rate for chained filters using software and hardware in DirectShow (1280x720) 101

6.1 Diehard score for LFSRs having different sequence lengths and number of taps 111
6.2 Lagged Fibonacci quality as a function of feedback length 117
6.3 Lagged Fibonacci quality ignoring birthday test 117
6.4 Diehard scores for additive lagged Fibonacci generator with different lengths and tap counts. 123
6.5 Diehard scores for multiplicative lagged Fibonacci generator with different lengths and tap counts. 123

7.1 Architecture of hardware distribution generator 136
7.2 Families of trapezoid component distributions 138
7.3 Interactive distribution approximation application. 142
7.4 PDF of approximation to a Gaussian distribution over $\pm 4\sigma$ using 1024 rectangles with an 8 bit range. 143
7.5 Zoomed section of peak of PDF from Fig. 7.4 over $\pm 1\sigma$. 143
7.6 Zoomed section of tail of PDF from Fig. 7.4 from 3σ to 4σ. 144

7.7 Example element index distributions using four elements to provide a four bit index, showing two possible configuration changes. 145
7.8 Uniform and triangular weighted approximations to the exponential distribution 147
7.9 Non-uniform weighted approximations to the exponential distribution . . . 148
7.10 Graph showing the difference in approximation accuracy when using different component types to approximate the Weibull distribution. The maximum absolute error is shown as a percentage of the largest probability in the PDF. 149
7.11 Graph showing the number of samples that can be taken from trapezoid approximations to four distributions without failure of the χ^2 test (at the 5% significance level). Sample sizes of increasing binary powers were tested and are shown in the graph as $\log_2$ the tested sample size. 150
7.12 Graph showing the accuracy of trapezoid approximations with different numbers of elements for four different target distributions. The maximum absolute error is shown as a percentage of the largest probability in the PDF. . 150
7.13 Graph of p-values for χ^2 test of 2048 trapezoid approximation to the Gaussian distribution. In the single sample case one value is taken from the approximation, while in the Central-Limit case two separate values were generated in parallel using the same table and combined through subtraction. 151
7.14 PDFs of the sums of pseudo-antithetic variates generated using a 2048 trapezoid approximation to the Gaussian distribution. Two point-wise correlated lists of variates a and b are combined using addition and subtraction, showing the change in the variance of the resulting PDF. If the sum of offset pairs from a and b are combined then there is no correlation, and the resulting PDF is Gaussian. 152

List of Tables

3.1 Differences in processing rate for software and hardware video filters 34

6.1 Area and speed for XorShift 32 bit generator. 113
6.2 Area and speed for pipelined XorShift 32 bit generator. 113
6.3 Comparison of Xorshift and LFSRs in the Virtex 2 architecture 114
6.4 Primitive polynomials for FIFO lengths using strict constraints 120
6.5 Primitive polynomials for FIFO lengths using relaxed constraints 120
6.6 Comparison of resource usage and speed for lagged Fibonacci generator variants on the Virtex-2 architecture 128
6.7 Percentage of xc2v1000 resources used by each of the generator variants. . 128
6.8 Area and speed for multiple instances of RAM based lagged Fibonacci generators on selected Virtex-II and Spartan-3 devices. Also included are generation speeds for a multi-threaded Mersenne Twister generator on a dual processor Pentium-4 2.66GHz. 129

7.1 Performance and resource usage of approximation units in Virtex-II architecture. 147
7.2 Comparison of different methods for generation Gaussian random numbers. 153

Chapter 1

Introduction

Reconfigurable hardware resources are increasingly used to accelerate software based applications, in areas such as graphics [184, 235, 217], simulation [10, 166, 203], mathematics [80, 225] and matrix operations [218, 252]. The acceleration is typically provided in the form of an FPGA (Field-Programmable Gate Array), contained on an accelerator card which is connected to a host computer. Applications running on the host are written to take advantage of the specific accelerator platform available, integrating the FPGA into their internal data-flow by transferring data between the host and the accelerator platform. Current methods for integrating hardware and software in this way rely on very tight coupling between the two parts, meaning that many such accelerated applications can only use one very specific platform, and will often not even be able to fall back on software processing if hardware is missing or fails. The hardware accelerator will also often be under-utilised, as while no accelerated applications are active the hardware will lie idle; the majority of applications have not been rewritten in order to take advantage of available hardware accelerators, and so are unaware that an accelerator might exist, let alone able to take advantage of it. Addressing these two problems should greatly increase the market and scope for reconfigurable accelerators, by increasing the flexibility and robustness of existing hardware-aware applications, and allowing existing applications to take advantage of hardware acceleration.

One of the areas where FPGAs show particular promise is in simulations and Monte-Carlo methods, as these applications can take advantage of the fine-grain and coarse-grain parallelism available, but only require small amounts of communications between host and FPGA. Examples of the types of simulations implemented in FPGAs include: test signal generation for equipment testing [10, 42]; genetic algorithms [191]; Bit-Error Rate testing [123, 130]; and financial algorithms [185, 251]. One critical component of these applications is the generation of random numbers, both from the uniform and non-uniform distributions. Uniform random number generators must be of extremely high statistical quality, with no non-random artefacts or flaws even over very long generated sequences, but must also require few hardware resources and operate at high speed. Existing techniques

for uniform random number generation only address area and speed concerns and cannot provide the statistical quality required, so new techniques are needed to meet the demands of the new generation of long-running, massively parallel simulations being implemented using FPGAs. Non-uniform random number generation is also key to such applications, but there are currently very few hardware techniques that combine area-efficiency, speed and quality, particularly when non-standard distributions are required. Random number generation is also a key component in software simulations, and one way that software applications may be accelerated is to offload random number generation on to an attached hardware accelerator.

There are two main components in this thesis, and each addresses challenges resulting from the rapid growth in speed, power and potential ubiquity of FPGAs. First we present the IGOL framework, which addresses the problem of integrating hardware processing functions into software applications. Second, we address the need for high quality and flexible random number generation in hardware to support new applications, such as highly parallel simulations, by presenting two random number generators that provide greater quality and flexibility than existing methods. These two methods are combined to provide an accelerated random number generation source for applications, even those that are not hardware aware.

The main contributions of this research are:

- A method for describing data processors and their implementations which allows for rapid development and progressive optimisation of applications (Chapter 3).
- A framework which implements the above method, allowing transparent use of hardware data processors from within multiple applications (Chapter 4).
- Automatic application of generic optimisations to hardware, which increase performance of hardware data processors without requiring modification of the hardware implementations (Chapter 5).
- A set of very high quality uniform random number generators tailored to take advantage of different types of FPGA resources (Chapter 6).
- An efficient method for approximating arbitrary probability distributions with adjustable quality and support for run-time modification of the generated distribution (Chapter 7).

The most important research contribution involving reconfigurable hardware management is the method for approximating probability distribution in Chapter 7. Using this method, a small amount of memory and logic can be used to provide an approximation for almost any distribution, including empirically derived distributions and combinations of other basic distributions. This method has many potential applications in simulation, where the ability to change the distribution without altering the hardware allows more freedom to

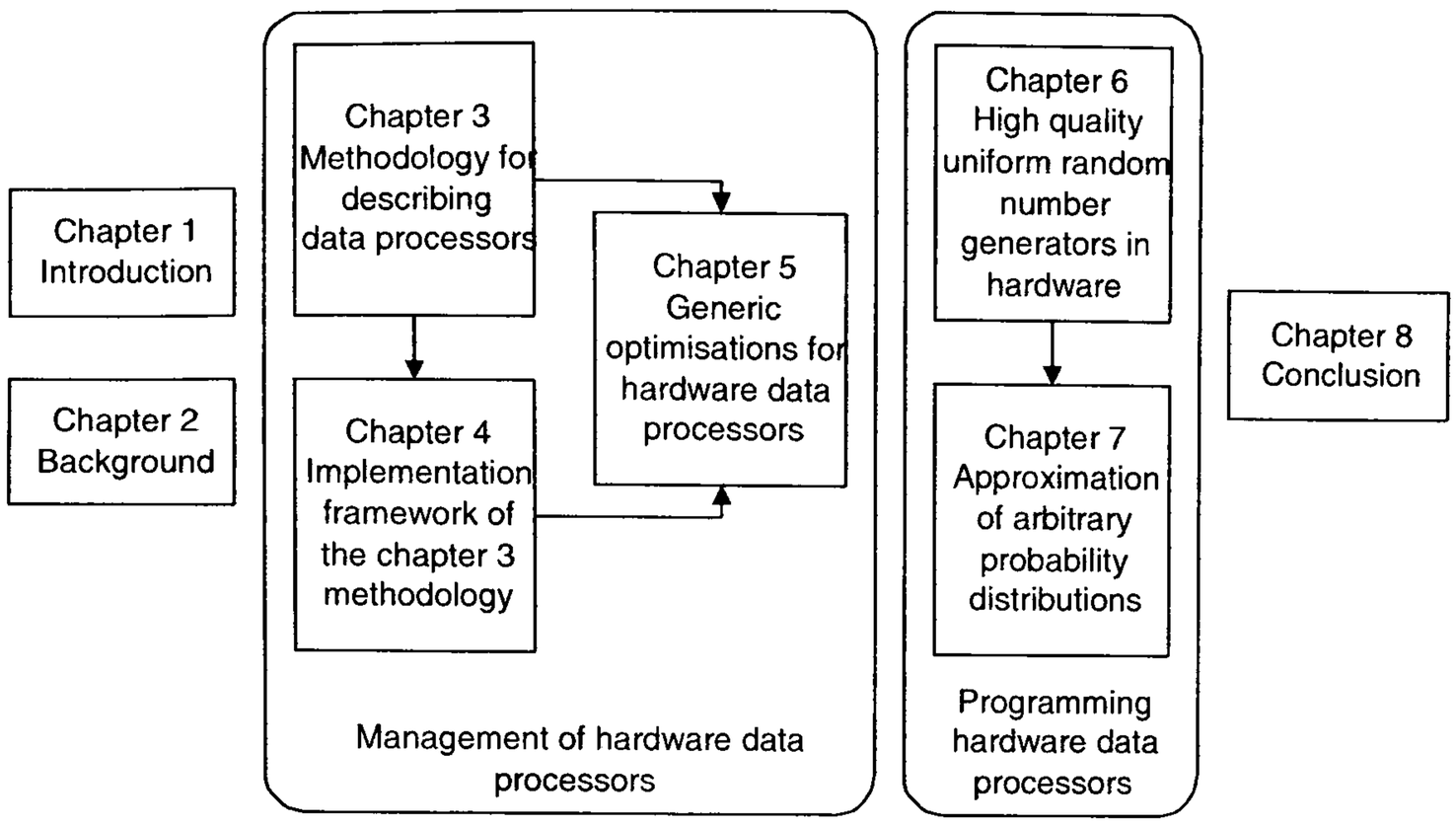


Figure 1.1: Illustration of the dependencies between the chapters of this thesis.

tweak distribution parameters at run-time. Our method is optimised for hardware implementation, and uses a novel technique for approximating target distributions. The resulting generator is much faster than equivalent software methods, and can be used to accelerate software applications that require non-uniform random numbers.

The greatest technical achievement involving reconfigurable hardware programming is the data processing framework presented in in Chapters 3, 4 and 5, which allows hardware data-processors to be created without needing to write a software component. The fact that any hardware data-processor immediately becomes available in multiple applications increases the value of any data-processor that is created, and also increases the potential utilisation of installed hardware devices. This kind of co-operation between hardware and software is critical if the idea of using FPGA based accelerators as general purpose – rather than application specific – compute resources is to become reality.

Figure 1.1 shows the dependencies between the different chapters in the thesis. Chapters 3, 4 and 5 are concerned with data processor management and the use of hardware from software applications, with each chapter building on the ideas and implementations of the previous one. Chapters 6 addresses uniform random number generation, and in Chapter 7 we use one of the random number generators developed in Chapter 6 as the source for producing uniform random numbers.

In the next two sections the main motivations for the work presented in this thesis are provided.

1.1 Hardware Acceleration of Software Applications

One of the contributions of this thesis is a framework for managing hardware accelerated platforms, which makes the creation and use of systems containing hardware accelerated software applications simpler and more flexible. This allows for hardware accelerated functions, such as the random number generators also presented in this thesis, to be integrated into applications without the application needing hardware specific communication code.

Almost since reconfigurable hardware such as FPGAs first became available, it has been possible to buy expansion cards for host computers such as PCs which typically contain an FPGA, some memory, and a means of transferring data between the host computer and the FPGA [4, 66, 206]. These cards traditionally contain the latest FPGA, have low production runs, and use expensive components, so these cards are usually expensive relative to the cost of a general purpose PC (Personal Computer). Users of such cards are often confined to two groups: researchers who use them for development and prototyping of general purpose FPGA algorithms and techniques, and developers of customised "black box solution" computers which perform demanding tasks such as video processing.

However, there is no reason why FPGA cards should not be considered a general purpose computing resource, outside of research equipment and dedicated solutions. There are many tasks performed on PCs which would benefit from offloading some processing tasks onto reconfigurable hardware. For instance, practically any task where computers are referred to as "workstations" rather than PCs is a potential candidate for hardware acceleration. Such tasks include video editing, computer-aided design, colour grading, numerical modelling, and 3D rendering, which could all benefit from a highly parallel data processor that can share the processing burden. Skilled workers' time costs more than hardware equipment, so improving these tasks with FPGA resources is likely to be cost effective.

Three reasons that FPGAs as a general purpose compute resource have not become common-place are:

1. previous generations of FPGAs were too small or too slow to compete with processors in many tasks;

2. FPGAs and FPGA based expansion cards are expensive compared to a standard PC;

3. there is no software support for applications to benefit from hardware acceleration, for instance methods for discovering and using hardware resources.

The first reason used to be true, but is not any longer: modern FPGAs are capable of performing complex tasks including floating-point calculations [145, 226]. The second reason is partially due to the historically high cost of cutting-edge FPGAs, and partly due to the low demand for such cards. However the need to use the latest FPGAs has reduced: low-cost devices, such as Spartan-3 from Xilinx and Cyclone from Altera, offer performance that often approaches that of the more expensive devices [7, 241], and the increase in size and IO

capabilities [244, 245] of modern devices means that less external components are needed to create a functioning hardware platform.

Nevertheless the cost of such cards may still be high even if the FPGAs are inexpensive, as the non-recurring development costs of creating the card must still be recovered. This results in a dependency cycle: cards are expensive because there is low demand, demand is low because there are no applications that can use them, no applications use them because none of their clients own the cards, and nobody owns the cards because they are expensive. By using the framework proposed in this thesis it is hoped that the cycle can be broken, by making hardware acceleration such as random number generation usable in many commonly used applications. Chapter 3, 4 and 5 describe our approach for managing reconfigurable hardware resources, which allows software applications to benefit from any available hardware acceleration, while Chapters 6 and 7 give two examples of the kind of hardware acceleration that can be used in the framework.

1.2 Random Number Generation for Simulation Applications

Simulations and Monte-Carlo based applications are an ideal candidate for hardware acceleration, and many such hardware accelerated simulations have already been developed [10, 42, 123, 130, 191]. Even more simulation methods are now becoming possible to implement due to the ability to instantiate many floating point operations in designs using modern FPGA families [218, 224]. However, one key requirement for such simulations that has received little attention is the random number generators used to drive these applications. In Chapters 6 and 7 two methods for random number generation in hardware are presented that close the quality and functionality gap between previous generators and the needs of this new types of long-running simulation.

There are two classes of random number used in stochastic algorithms: uniform random numbers, and non-uniform random numbers. Uniform random numbers usually provide the underlying randomness needed in a simulation, and where non-uniform random numbers are needed these are usually generated by transformation of uniform sequences. The key requirements for uniform random number generators are that they should exhibit near perfect statistical randomness, even over extremely long sequences: many applications consume more than 10^{12} or more samples in a single run [123, 139]. In software this is achieved by using large amounts of memory or by combining multiple generators [110], but both memory and logic are at a premium in FPGAs, so the same techniques are not appropriate.

Most FPGA applications requiring uniform random numbers currently use Linear Feedback Shift Registers (LFSRs) [75] and Cellular Automata (CAs) [195]. These generators both have efficient bit-level implementations, allowing for area-efficient implementations with high clock rates. However, for newer applications that consume many more samples

these techniques are no longer appropriate, due to low period and poor statistical quality. In Chapter 6 a method for generating very high quality uniform random samples in modern FPGAs is presented, which is appropriate either for supplying numerical processing units within an FPGA design, or for providing uniform random numbers for use in a software application.

Non-uniform random samples are also an important component in stochastic algorithms, for example the Poisson distribution is often used in queueing simulations [55] while the Log-Normal distribution is common in financial simulations [19]. The only non-uniform distribution that has received significant attention from the FPGA community is the Gaussian distribution [20, 123, 124], due to its importance in Bit-Error Rate modelling [71]. Although other distributions can be produced by creating custom designs, this will require work on behalf of hardware engineers for each new distribution and require hardware simulation designs to be resynthesised in order to slightly change the distribution of the random numbers it consumes. In chapter 6 a novel method for producing non-uniformly distributed samples is presented, which allows a single hardware design to produce almost any distribution. This method is also appropriate for generating non-uniform random numbers to be consumed by software, particularly when combined with IGOL to manage the hardware and configuration on behalf of applications.

Chapter 2

Background

The continually increasing performance of hardware presents a challenge to both hardware engineers and software developers. Both groups are finding it difficult to design, create and manage systems that are defect free, make maximum use of the resources available, and meet performance targets. In this section we survey some of the overall trends in modern software and hardware design, then examine some of the methods and tools used to design and manage reconfigurable hardware. Finally the issue of random number generation is considered, as this is a critical part of many hardware and software applications.

2.1 System trends

Software Software developers have for a long time assumed that CPUs will become faster with every generation, so that existing programs and systems become faster without any need to change the way that code is written. However, this is no longer the case; rather than processors executing more sequential instructions per second, the main increases in performance are now provided by multiple processor cores on a chip, or by using very larger clusters of cheap medium-performance processors [76, 106]. Taking advantage of this local and distributed parallelism has lead to the use of a variety of paradigms for exploiting parallelism such as threads [35], functional programming [25, 85], Remote Procedure Calls [9] and Web Services [175].

Another pressure has been the requirement for secure and robust code, particularly in the face of the Internet, with its wider accessibility. As systems grow in complexity, the ability of the designers and developers to catch all errors before releasing code is severely reduced due to the amount of code involved and the number of possible interaction patterns between components. Minor software errors could allow hostile code to enter systems, and possibly to gain complete control of the host [200].

One solution adopted by industry to manage the conflicting desires for fast development, high performance, robustness and security has been to move from code which is written for

a specific platform or instruction set to code which is targeted to an abstract platform, or VM (Virtual Machine). This has a number of advantages:

- Security and access control can be strictly enforced by the VM, even in foreign code of unknown provenance [86].
- Rich meta-data allows for transparent access of objects across process and machine boundaries [131].
- Support for migration of running processes between computers [102].
- Built-in support for coarse-grain parallelism and synchronisation primitives [67].
- Automatic memory management, error propagation and application isolation reduce the scope for bugs, and minimise their effect on the overall system [56].

The two most common examples of VM based systems are the JVM (Java Virtual Machine) [215] for Java [79], and the CLR (Common Language Runtime) [158] for the .NET framework. The JVM comprises a low-level stack based processor architecture, not unlike the architecture of a conventional concrete processor, but with intrinsic support for object-oriented features such as object construction and virtual method invocation. Although mainly associated with the Java language, there are also compilers that can compile many other languages such as LISP, Scheme and LOGO into executables for the JVM.

The .NET CLR adopts a similar level of abstraction as the JVM, but provides some higher level features, such as direct support for generics at run-time [45]. Unlike the JVM, the CLR was designed to be used with multiple languages, of which the three main languages are C# [231], VB.NET [159] and C++ [204], although there are others such as IronPython, F#, J# and C-Omega. One feature of the CLR not present in many other VMs is support calling into native code (i.e. code that runs directly on the underlying processor, rather than through the CLR), allowing executables to mix both native and CLR code. Most VMs include some support for integration with native code, such as the JNI (Java Native Interface) support provided by the JVM, but rarely in the tightly integrated way that the CLR allows. This support is particularly useful for migrating C++ based applications to the CLR, as it allows legacy components to be used without requiring the code to be rewritten.

There are many other examples of languages which compile to a specialised VM rather than to native code, such as PERL (Practical Extraction and Reporting Language) [222] and Python [220], and also of language independent VMs, such as the PVM (Parallel Virtual Machine) [73]. It is often not clear whether a given architecture should be classified as a VM, an emulator or a native instruction set architecture: in principal there is no reason that the x86 instruction set architecture [98] should not be considered a VM, as x86 executables can be executed on other architectures through the use of emulators[183], and micro-processors that directly implement the JVM architecture are available [238]. The main difference is the intent of the designers, as VMs are usually designed with portability and simplicity as the focus, while instruction set architectures are designed for efficiency and performance.

Hardware The number of gates available to hardware developers has increased every year since the introduction of the integrated circuit, following Moore's Law [162], and the main barrier to the performance of hardware designs now often lies in the design process rather than a lack of available resources [230]. Processor design teams are growing in size faster than the resultant processor performance, with the number of large ASIC (Application Specific Integrated Circuit) projects started decreasing every year [37].

FPGAs (Field-Programmable Gate Arrays) have provided one way of using the large number of gates available without requiring an equally large design effort for two main reasons. First, creating and producing an FPGA is simpler than designing an ASIC with the same area, due to the regularity of the gate-array design and the ability to transfer many components of the gate-array fabric design between different sized FPGAs [242]. Second, creating a design that targets an FPGA is in general cheaper and faster than creating a functionally equivalent ASIC.

The reason that designs targeting FPGAs are cheaper and faster to create is because they have a very regular structure which reduces the number of choices when creating the design, as well as reducing the number of potential problems found within ASIC design such as cross-talk [114]. This allows for much more automation of the design process, allowing small teams to produce complex designs.. The design process is also less risky, as any flaws in the design can be corrected after deployment using techniques such as reconfiguration over the Internet, which means less time needs to be spent on verification and testing.

Initially FPGAs had the disadvantage that they were small and usually only used to provide glue logic, but as the FPGA manufacturers have improved their designs, and the underlying ASIC processes have decreased in size, the gate count has increased significantly. This has led to some of the problems in the ASIC design process resurfacing, with talk of a "design gap" between the size of designs that FPGAs can support and the complexity of designs that teams of engineers can manage [230]. In order to address this design gap large amounts of work has been done in the field of hardware compilation.

2.2 Hardware Compilation

Assuming that there is a design gap between the amount of logic available, and the amount of logic it is possible for developers to manage, it is natural to look to the hardware compilers and tools and hope that new compilation technology and automated design techniques will allow engineers to effectively manage the large numbers of gates available. This section will examine some of the compilers and languages used to target FPGAs.

2.2.1 Traditional Hardware Description Languages

VHDL (VHSIC Hardware Description Language) [96] and Verilog [214] are two of the oldest HDLs (Hardware Description Languages), originally used to create ASIC designs,

and the majority of commercial FPGA designs are still written in one of these two languages. Both these languages operate at the Register Transfer Logic (RTL) level, so rather than using high-level constructs such as loops and sequential statements, designs are specified as connections between low-level components such as registers and logic gates.

Compilers for these languages are improving to make developers lives easier. For example the Synplify compiler now performs optimisations such as automatically pipelining designs, re-timing logic, and un-gating clocks [207]. This makes it possible for hardware developers to spend less time concentrating on architecture specific details, as low-level performance optimisation can be performed by the compiler. By using detailed knowledge of the target FPGA, such as gate and routing delays, the compiler can apply transformations that increase performance without changing the designs functionality.

2.2.2 High-Level Compilers

One direction taken to try to reduce the hardware design gap has been to compile high-level languages into FPGA designs. The goal is to allow hardware engineers to operate at a higher level than registers and gates, but there are many different approaches to the abstraction level provided.

Tools such as Handel-C [38] encourage a gradual transition from sequential C code to parallel C code through the incremental addition of "par" constructs which execute statements in parallel. All the control constructs found in C are still available, but there is a fixed mapping between statements in the source code and clock cycles in the eventual hardware design. This approach provides a middle-ground between RTL design and high-level design, as in order to create efficient designs the author of the code must consider how the high-level constructs will be mapped into registers and gates. The compiler does not attempt to extract any parallelism from the code, so any fine-grain or coarse-grain parallelism must be indicated in the code using par blocks. The Handel-C language is presented as a rapid application development tool, allowing the creation of complex hardware components in very short time periods.

Haydn-C [53] is an evolution of the ideas found in Handel-C, which removes some of the burden of RTL level optimisation. Haydn-C code is written as pipelined modules, each of which is written with a specific throughput and latency. For example, a task with five main steps might be written as a module with a throughput of one result per cycle, and a latency of five cycles, as this is the most natural way for the programmer to translate the specification for the task into code. However, once all the modules are combined together it may be found that too much area is being used, so some modules must be partially sequentialised to save logic resources. Alternatively, it might be discovered that with just five stages the critical path is too short, limiting the clock speed of the whole design, so the five steps must be split across multiple pipeline stages.

In Handel-C the pipeline would have to be re-written in order to adjust the latency or

throughput, but in Haydn-C these parameters can be adjusted automatically. By sequentialising and then re-parallelising the code, the compiler can adjust these properties, allowing area and performance to be satisfied without re-writing the modules in a design. This allows for easy design-space exploration, either under manual control, or automatically through the compiler.

SPARK [83] is a compiler framework which takes sequential C code and uses advanced compiler techniques adapted from software compilers such as code motion and peep-hole optimisations to create customised gate-level parallel designs. The SPARK compiler will automatically turn any sequential C code into a hardware design, but the results will not have the performance of a custom designed solution. Although the gate-level description is expressed using VHDL, the resulting code is not suitable for human editing, so it is difficult to tweak the output for better area or performance.

A Stream Compiler (ASC) [155] is a method for describing data stream processors, and is designed to allow high-performance pipelined designs to be specified using a high level description. ASC programs are written using standard C++ [204], and can use all the normal operators and statements available, such as arithmetic operations, conditional statements and variable assignment. The only real restriction is on the types of loops that can be used, as the system cannot process certain types of loop-carried dependencies [194]. The C++ description of the code is executed to create the design, but instead of directly executing the statements in the program, the statements and expressions are used to build a data-flow graph for processing in hardware. This data flow graph is compiled to a netlist using PAM-Blox [153], which can be compiled to a configuration bitstream using standard place and route tools.

2.2.3 FPGA Based Processors

A recent development has been the tight integration of traditional instruction processors and FPGAs. An example of this type of system is the Virtex2 Pro, which combines one or more PowerPC processors (implemented directly in silicon) with a large amount of reconfigurable logic [239]. These hybrid systems are ideal for SoC (System-on-a-Chip) solutions [74], as they remove the need for a separate processor to manage the FPGA logic. However, processors can also be instantiated within the reconfigurable fabric, albeit with lower performance than the directly instantiated processor cores.

Early examples of FPGA based processors include the DISC (Dynamic Instruction Set Computer) [232] and the Nano Processor [233]. These processors were designed to have very low resource usage (due to the very small amount of logic available in FPGAs at that time), and attempted to compensate for low clock rate and IPC (Instruction-Per-Cycle) rates by incorporating custom instructions. These custom instructions are determined by analysing the target application, then inserting appropriate custom instructions into the processor architectures. For example, in an image processing application the DISC proces-

sor is shown to achieve a 23 times speed-up over its default instruction set, by incorporating a custom instruction for calculating the mean of multiple pixels.

The Flexible Instruction Processor (FIP) [192] is an attempt to combine a custom processor with application specific custom instructions, to provide a high performance application specific processor. A core processor template is modified to include custom instructions, which are generated through static analysis of the source code. These custom instructions can replace sequences of instructions with a single instruction which executes in fewer cycles, so reducing the overall number of cycles taken to execute the application. However the FIP architecture also supports the collection of statistics at run-time, allowing the processor instruction set to be reconfigured in response to changing application characteristics [193]. Using run-time adaptation, a three-fold increase over a static design is shown in an AES (Advanced Encryption Standard) [57] processing application.

As well as flexible instruction set architectures, a number of FPGA based processors that are designed to achieve good performance even without creating custom instructions have been created. Where these are tied to specific architectures, such as the Micro-Blaze for Xilinx devices [243], and the NIOS for Altera devices [6], the processor architecture and instruction sets can be tailored to produce low area designs that provide good performance. There are also device independent designs such as LEON [112] and Open-RISC [115], which are not as area efficient as the architecture specific processors [149], but have the advantage of portability to other FPGAs and into ASIC designs.

Processors that implement Virtual Machine instruction sets are also available, such as JOP (Java Optimised Processor) [190] and LightFoot [238] which both implement the JVM (Java Virtual Machine) architecture. Such processors can provide good performance for embedded applications, with JOP achieving three times the processing rate of a 486SX processor running at the same clock rate.

2.3 Environments for Customised Applications

In this section an overview of systems for producing customised applications is presented. This includes systems for composition of applications from software components, as well as methods for combining hardware and software into a single application.

2.3.1 Ptolemy

Ptolemy [33] is a system for assembling heterogeneous processing blocks, where each Block represents a processing task to be performed at run-time. Blocks can be implemented in many different ways, such as local software processes, embedded software, and reconfigurable hardware. Blocks are connected used PortHoles, which provide a standard interface that allows data to flow in and out of each Block, no matter how the underlying functionality is implemented. Blocks are connected to each other via their PortHoles, and are then organ-

ised into Galaxies, with all the Galaxies belonging to the overall Universe. The universe represents the overall application, and the composition of Blocks within it captures all the processing tasks needed within that application.

Within each Galaxy the Blocks all belong to the same domain: this means that they all share a common infrastructure and data model. For example all the blocks in one domain might be software based, passing data blocks around as memory buffers, while another domain might be hardware based, where data blocks are routed around within the FPGA. To communicate between domains the framework supports Wormholes, which provide a way for domains to communicate with each other. In this way large systems can be assembled, using many blocks from multiple domains, without each domain needing to known how data from other domains is being passed between them.

FPGAs are supported within Ptolemy using a special domain called the Adaptive Computing System domain [171], which is also used to support DSPs (Digital Signal Processors). The ACS model is based on data-flow graphs, with support for detecting parallelism within the graph and optimising the bit-width of data-paths. These graphs are compiled into a pipelined FPGA design, which is loaded at run-time into the containing application.

The Ptolemy system attempts to capture every possible component of a system within the same framework, and in this goal it seems to succeed. Case studies are presented for the assembly, simulation and deployment of heterogeneous systems, in particular those containing DSPs and embedded processors [176], but for systems involving hardware such as FPGAs there are fewer examples. The integration of reconfigurable hardware uses the same model as DSP-based hardware, which restricts the ways in which FPGAs can be used to simple pipelined data-processing. It also provides little support for the packaging and deployment of systems once they have been developed and tested, which is important in systems with combined hardware and software due to the need to manage both application binaries and hardware configuration files.

2.3.2 System Generator

System Generator from Xilinx [246] is a highly integrated plug-in for the Matlab Simulink system [144], which allows the rapid prototyping, simulation and synthesis of hardware designs within the Matlab environment. The Simulink system is a Matlab feature allowing the creation of hierarchical graphs of processing and auxiliary components, allowing complex systems to be easily simulated. Components can be selected from a library of built-in blocks, or can be written using m-code (Matlab's language) or C++. Simulink has been used in the design of DSP and embedded processor based systems, with a key feature being the high quality graphical editor and strong simulation and debugging support.

System Generator builds on Simulink by adding a series of components for modelling FPGA based processing blocks, such as variable width mathematical and bitwise operators, memory blocks components such as block RAMs and FIFOs, and larger building

blocks such as FFT (Fast Fourier Transform) and LFSR (Linear Feedback Shift Register) blocks [240]. System Generator also allows algorithms built from these blocks to be tested using bit-exact functional simulations of the hardware components. Standard Matlab features can be used to feed data into and capture data from a graph, allowing graphing and analysis of results while adjusting block parameters [186].

While the simulation of bit-exact code is important, the major feature of System Generator is its ability to create synthesisable code from the Simulink description, allowing developers to move straight from the graphical simulation to VHDL code, complete with test-bench. Many of the blocks supplied with System Generator are generated as high-performance device-specific EDIF blocks, created using Xilinx Core Generator [240]. The blocks instantiated using Core Generator are optimised using device specific knowledge and relative placement information, and so ensure that components such as multipliers and dividers provide optimal performance once in hardware. The blocks generated by Core Generator are connected together using automatically generated VHDL, which can be synthesised to a hardware configuration.

System Generator is in common use, and as the strong links with Core Generator ensure high performance, many developers use it to provide the final version of their hardware, rather than as just a rapid prototyping tool to develop the algorithm [60]. However, the system does have some limitations, with feedback loops and data dependent control difficult to implement. However, where an application can be expressed using the System Generator model, the environment makes it easy to generate initial versions of designs. In addition, some hardware platforms such as the ADMX-RC2 [4] are directly supported by System Generator, meaning that once a hardware configuration has been produced it can be used directly from Matlab.

2.3.3 Chimera Port-Based Objects

The Port-Based Object is a software abstraction for designing and executing real-time software, with support for dynamic or run-time reconfiguration of applications [202]. The system was originally designed as the control system for CMU's Reconfigurable Modular Manipulator System (RMMS), a project to create reconfigurable robots [189]. RMMS demanded control systems which were capable of hard real-time performance while supporting the kind of control feedback and data processing needed for inverse kinematics, with the added constraint that the software must be as simple to reconfigure as the physical robots.

The system is based around objects with ports, with instances of objects connected together via their ports. This is similar to systems such as Ptolemy and System Generator, but below the highest level abstraction of data processing graphs it uses a different data and processing model. Rather than each object producing discrete packets of data which are then consumed by another object, each object continuously consumes input and produces

output, independently of any other object's port IO. This is based on port-automaton theory [201], whereby multiple concurrent processes operate on the shared state of the world, with updates to the world by one process being immediately visible to all other processes. This removes the need for complex scheduling and message passing between objects, and removes the possibility of deadlock.

Communication of the world state between objects is accomplished through a global state-table. This is a central rendezvous point for all objects that holds the last value produced from all the output ports of all the objects. As each object begins a processing cycle it is given the latest values from the table for its input ports, performs it processing, then the values from its output ports are stored back into the table. The table is protected by a mutex to stop concurrent accesses, so multiple threads and processors can share the same state table.

The port-based objects themselves are implemented as a combination of C source code to provide functionality, and a text based manifest to provide meta-information about what ports the object has. The Chimera framework will instantiate the objects as processes at run-time, using a small number of entry points into each object to execute the overall system. During reconfiguration the system can remove and instantiate running processes, changing the source of values within the state-table, effectively re-routing data between nodes in the graph.

2.3.4 Application-specific Acceleration using FPGAs

Many applications provide extension mechanisms through plug-ins, or use certain types of functionality through well-known APIs. This allows third-parties to provide functionality that these applications do not already contain. Plug-in architectures are found in many large applications, such as Microsoft Office and the Mozilla web browser, but these are not good candidates for hardware acceleration as such plug-ins usually request services of the applications rather than performing processing on their behalf. Of more interest are applications for media creation and presentation, as these typically require fast processing of large batches of data such as images, a field where FPGAs have already been successfully used [21, 111, 217].

Examples of media creation applications with plug-in systems include Adobe Premiere, Adobe After-Effects, Virtual-Dub [122], Adobe Photo-Shop, and The GIMP (GNU Image Processing Program). These typically use their plug-in system to allow special-effects and filters to be applied from within the application. Some media playback applications also include plug-in support, such as the Win-Amp music player and Windows-Media Player. These applications allow the application of filters in real-time to achieve effects such as noise reduction in video, or frequency adjustment of audio streams. The plug-in architectures of these applications are usually specific to the application, so a video effect plug-in written for one application will not work in any other application.

By using these plug-in architectures it is possible to integrate hardware into applications which have no explicit hardware support. An example of this can be found in [88], where hardware accelerated video processing is integrated into the Premiere video editing application. The Sonic is a hardware platform designed for video processing, including support for streaming of data between FPGAs over internal buses. As well as supporting applications that communicate directly with the platform, a plug-in for Premiere is also provided which allows Premiere to load and execute video filters through Sonic. A similar approach is also taken in [199], where a hardware accelerated Photo-Shop plug-in is developed which uses a VCC XC6200 card to accelerate image processing operations.

Both these applications perform the same type of processing, as the video processing performed in Sonic stores no data between frames, so is identical to repeatedly processing static images. However, because of plug-in incompatibility the Sonic cannot be used to accelerate Photo-Shop, and the VCC XC6200 cannot be used to accelerate Premiere. In order to allow these accelerators and applications to work together, a new plug-in would need to be created for each hardware accelerator. Even if a new application is written from scratch it is not easy to access the hardware acceleration, as the Photo-Shop and Premiere plug-in APIs require the application to provide significant hosting infrastructure. It is more likely that the application will be written to directly access the hardware platform, bypassing the abstraction layer provided by the plug-in wrapper, and requiring separate code paths for every supported platform.

Another way in which applications can access external functionality is through well-known API layers, such as the OpenGL [197] and Direct-X [15] 3D graphics APIs. In [205] an FPGA based 3D graphics renderer is integrated into the OpenGL graphics framework. The graphics renderer is implemented on an RC1000 accelerator card [206], but from the point of view of applications using the framework the FPGA looks exactly like a "normal" graphics card. This provides transparent acceleration, to the point where OpenGL screen-savers and even the 3D game Quake can use the FPGA as a 3D renderer. This demonstrates the advantage of non application-specific standards, as the FPGA renderer immediately becomes accessible from any application that supports the OpenGL standard.

2.4 Random Number Generation

Random number generation is at the heart of many processing intensive applications, such as simulations, optimisation problems and numerical approximation. Many different methods for the generation of random numbers in software are known, as demand for higher quality random numbers have led to the development of more complex random number generators. Simulations are also beginning to be implemented using FPGAs [10, 166, 203], also requiring high quality sources of random numbers. In this section a survey of the main methods for uniform and non-uniform random number generation is presented, followed by an investigation into the state-of-the-art for hardware random number generation.

2.4.1 Uniform Random Numbers

Random number generation is an important part of simulations performed using computers, and much time and effort has been spent trying to create pseudo-random sequences in software that have all the statistical qualities of a truly random sequence. As computers have increased in speed it has become possible to perform simulations that consume huge amounts of random numbers, which requires random number generators that maintain the appearance of randomness over much larger sequences. Here a summary of some of the common methods for generating uniform random numbers is presented.

The classic software random number generator is the Linear Congruential Generator (LCG) [110, 181]. This is based on a recurrence equation which creates the next value in the sequence based on the previously generated value:

$$x_{i+1} = (ax_i + c) \mod m \tag{2.1}$$

The choice of a and c is important, as some choices produce very poor quality sequences, or produce sequences that quickly repeat. A set of basic criteria for choosing m and a can guarantee a maximum period of m before the sequence starts repeating, and other criteria are known that maximise the quality of the sequence [110]. Extensive testing of the possible parameters have been performed, and the "Minimal Standard" generator [170] is recommended [181] for most applications, with $a = 7^5, c = 0, m = 2^{31} - 1$.

Although the LCG is simple to implement, it is known to produce a very poor quality sequence. Two specific problems are that the low bits of the sequence are less random than the high bits due to the multiplication, and that it has easily detectable serial correlations. This means that if consecutive tuples of k numbers are taken from the generated sequence, then used to plot points in k dimensional space, the numbers will not fill the space up uniformly, but will fall on $k - 1$ dimensional planes [65]. For example, if $k = 2$ a 2D plot of consecutive values would show regular lines forming from the sample points. Generators such as the Minimal Standard generator provide amongst the best possible quality for LCGs, but still have enough statistical flaws to make LCGs unsuitable for some applications [91].

A generator which retains the simplicity of the LCG but that provides better statistical quality is the Multiply-With-Carry (MWC) generator [78, 137]. This is based on the following set of recurrence equations:

$$s_{i+1} = ax_i + c_i \tag{2.2}$$

$$x_{i+1} = s_{i+1} \mod m \tag{2.3}$$

$$c_{i+1} = \lfloor s_{i+1}/m \rfloor \tag{2.4}$$

This generator uses the overflow from the previous calculation in the next calculation, and so reduces the sequential correlations found with LCGs, although the effect is still present [213]. The maximum period of MWC generators is still rather low, as typically m

is chosen to be 2^w, where w is the width of the machine word. MWC generators have a maximum period of 2^{2w}, so for a 32 bit architecture the maximum sequence length is 2^{64}. This is more random numbers than most applications will consume, but sequence length is also an indicator of potential quality, so it is desirable to have a period far in excess of the number of values that will be consumed. A suggested rule of thumb is that the sequence length should be much larger than the square of the number of samples taken [119, 187]. Larger values of m can be used, for example $m = 2^{64}$ is easily implemented using a 64 bit architecture, but when implemented on a 32 bit architecture this will require more than twice as many multiplies and additions per generated number. Even a period of 2^{128} may not be enough for some applications, particularly in massively parallel simulations, where it is important that independent instances of the random number generator do not produce overlapping sequences [47, 90].

To create random number sequences with higher periods it is necessary to use a generator with a larger state. The Lagged Fibonacci is a class of random number generators that uses combinations of two or more previously generated samples (rather than just one as with the LCG and MWC) to form the next samples [26, 27, 119, 135]:

$$x_{i+1} = x_{i-r} \bowtie x_{i-s} \mod 2^w \tag{2.5}$$

The operator $\bowtie$ can be almost any linear operator, but common choices are $\oplus$ (bitwise exclusive-or), $+$, $-$ and $\times$. The value of the larger constant r in concert with the operator determines the maximum possible period of generator. With the $\oplus$ operator the period is at most $(2^r - 1)$, while for addition it is $(2^r - 1)(2^{w-1})$. The maximum period can only be attained for specific combinations of r and s, which are derived from primitive polynomials.

The different operators are also known to provide different degrees of statistical quality, with $\oplus$ providing the least random sequence for a fixed value of r, and $\times$ providing the best. To attain an acceptable quality sequence with $\oplus$ it is necessary to use extremely large values of r, using large amounts of memory [47].

In an attempt to maintain the high speed attainable with $\oplus$, but to reduce the memory requirements, a number of modifications have been proposed [26, 146, 147] which use more complicated feedback patterns to improve randomness. These culminated in the Mersenne Twister [148], which has a period of $2^{19936} - 1$, and maintains a uniform distribution when hypercubes of dimension up to 623 are filled with sequences of samples, while consuming only 624 words of memory. This has become almost the default choice for software random number generation, as it is fast compared with other high quality generators, and has proven statistical properties.

2.4.2 Non-Uniform Random Numbers

As well as uniform number generation, there is also a need for random numbers which follow different distributions, for example the Gaussian, Poisson and Exponential distributions

used in many simulations [55, 89, 187]. Usually a two step approach is taken, where first one or more uniformly distributed numbers are generated, which are then transformed through some algorithm to match the desired PDF (Probability Distribution Function) [61].

A continuous probability distribution is usually described in terms of the Cumulative Distribution Function (CDF), a function that determines the probability that a sampled value will be less than or equal to a given point; the first derivative of the CDF is the Probability Density Function (PDF).

$$CDF(D,x) = \Pr(D \leq x) = \int_0^x PDF(D,y)dy \tag{2.6}$$

For example, the PDF and CDF of the exponential distribution are:

$$\Pr(E = x) = e^{-x} \qquad \Pr(E \leq x) = 1 - e^{-x} \tag{2.7}$$

Discrete probability distributions can also be described in terms of a CDF, but rather than a PDF have a Probability Mass Function (PMF). This gives the direct probability that a single specific value occurs, unlike the continuous PDF. In practice all continuous distributions can only be sampled as discrete approximations (due to the finite number representations used to store the samples), and discrete versions of continuously defined distributions can be derived by quantising the range of the distribution.

CDFs may be defined analytically through a formula, or implicitly through empirically derived data. Certain well-known analytically defined distributions, such as the Gaussian and Poisson distribution, are used in many applications because theoretical models are based upon these distributions. However, in many simulations of real-world systems these distributions are still used as approximations to empirically derived distributions, even when the measured distributions are known to be slightly different from the theoretical model. For example, when simulating a communications channel it is common to use a Gaussian noise model, fitted to the empirical noise distribution using the standard deviation [104]. If the empirical PDF could be efficiently sampled instead then the simulations could produce more accurate results.

There are many techniques which can used to create random samples from a specific non-uniform distribution. Some of these are distribution specific, for example the Box-Muller and Wallace method for sampling from the normal distribution [24, 223]. Other more general techniques allow potentially any distribution that satisfies certain constraints to be targeted, but must be manually customised to fit each specific distribution. These range from quite restrictive techniques such as the Ziggurat method [140], which can only approximate distributions with strictly decreasing densities, to very general purpose techniques such as CDF inversion, which can in principle be used to approximate any distribution.

There is usually a trade-off in quality, design time and performance, against the generality of a technique: creating a new distribution-specific approximation will usually require

a large investment of development time, or acceptance of a poor quality or low performance approximation. If a technique applicable to a family of distributions (e.g. the Ziggurat method) can be used, the process can be semi-automated and will result in somewhat predictable quality and performance, but only if the desired distribution belongs to that family.

The most general purpose technique in common use is CDF inversion, as this can be used to approximate any distribution where the CDF function can be inverted. Given the inverse of the CDF, CDF^{-1}, and a random variate $u \sim U(0,1)$, then $CDF^{-1}(u)$ will provide random samples from the target distributions. For example, the Exponential distribution has a CDF of $1 - e^{-x}$. This can be inverted to give an inverse CDF of $-\log(u)$, allowing sampling from the exponential distribution directly, although requiring the evaluation of $\log$ for every sample produced.

Even if the CDF cannot be analytically inverted it is often possible to find an approximation with sufficient accuracy for a task, for example by using a rational polynomial approximation, or a piecewise approximation with polynomials. The Gaussian (or Normal) distribution has no closed-form solution for the inverse CDF, and so is often approximated in this way. For example in the PPND16 [229] inverse CDF algorithm three high-degree rational polynomials are used to provide a transform accurate to 16 decimal digits. Approximations may also be used when an inverse function is available, but prohibitively expensive to calculate: while the exponential inverse CDF can be evaluated directly, a less computationally expensive approximation may be more appropriate if accuracy of the distribution is less important than speed.

Most of these CDF inversion approximation techniques either require extensive work for each new distribution or large amounts of RAM in order to hold pre-calculated data, but there are also automated techniques that attempt to produce an accurate inversion without large tables. In [92] a mechanical method is developed for the approximation of arbitrary inverse CDFs by adaptively splitting the domain into intervals and then approximating these intervals with Hermite polynomials. The technique is automatic, and has the advantage of providing guarantees on approximation error.

Another general purpose technique for sampling from arbitrary distributions is the ratio-of-uniforms method [108]. In this method points (u, v) are generated within a region of a plane, such that the ratio of the coordinates follow the desired distribution x, i.e. $PDF(\frac{u}{v}) = PDF(x)$. The region within which the points must be sampled is dependant on the target distribution and often has a complex shape, so the uniformly distributed points within the region are generated using rejection. This means that many candidate points within a larger section of the plane (for example a rectangle) are produced, and only those within the target region are accepted. To create an efficient random number generator it is important that the acceptance test is fast (on average) and that the proportion of rejected points is low. Methods for automatically applying the ratio-of-uniforms have been developed [128], which can create efficient approximations for many distributions. In the average case these methods are very efficient, accepting or rejecting most candidate points

with very few operations, but in a small proportion of cases complex functions must be used to determine whether to accept a point. The method is also not completely universal, as the distribution to be generated must have certain properties, commonly found in analytical, but not necessarily empirical, distributions.

An example of a distribution specific distribution approximation is the Box-Muller method [24], used to transform uniform variables x_a, x_b into two normally distributed variables n_a, n_b:

$$n_a = \sqrt{-2 \log x_a} \cos(2\pi x_b) \tag{2.8}$$

$$n_b = \sqrt{-2 \log x_a} \sin(2\pi x_b) \tag{2.9}$$

This provides a simple way of generating normally distributed numbers, but requires the evaluation of a computationally expensive trigonometric function for every output sample.

A modification to this process that eliminates the trigonometric calls is to offset and scale the input variables so that they are drawn from the range $[-1, +1]$, with the additional condition that $x_a^2 + x_b^2 < 1$. Pairs of uniform random variables x_a, x_b are generated until these conditions are satisfied, and then two normal variates can be generated [14, 109]:

$$r = x_a^2 + x_b^2 \tag{2.10}$$

$$n_a = x_a \sqrt{\frac{-2 \log r}{r}} \tag{2.11}$$

$$n_b = x_b \sqrt{\frac{-2 \log r}{r}} \tag{2.12}$$

Removing the call to sin and cos should speed the transformation up in the average case, but in the worst case many input pairs must be rejected before getting two that are accepted. However, the probability of accepting any pair is $p_a = \frac{\Pi}{4}$ (the ratio of the area of a unit circle to a unit square), so the probability of rejecting k or more consecutive pairs is $(1 - p_a)^k$, and the average number of rejected pairs is $p_a/(1 - p_a)^2$. This means that the probability of rejecting more than 3 pairs is only 1%, and the expected number of rejected pairs per call is 0.348.

A more general transformation algorithm that can be used to approximate any distribution with strictly decreasing densities (and through reflection about the mode, symmetric uni-modal distributions) is the Ziggurat method [140, 141]. This uses a set of thin horizontal rectangles to approximate the shape of the distribution PDF, shown in a simplified form in Figure 2.1, where half of the Gaussian distribution is covered with 8 rectangles. To sample the distribution one of the rectangles $a_i \in A_1..A_8$ is selected using a uniformly distributed integer from 1 to 8, then a continuous random number between 0 and x_i is generated. If the the point is less than x_{i-1} then it can be immediately accepted, as it is clearly under the curve. Otherwise a second uniform value between y_{i-1} and y_i is generated, and a test against the PDF is made. If the point lies above the curve then it is rejected, and the process is repeated, otherwise the point is accepted. The lowest segment is handled using special

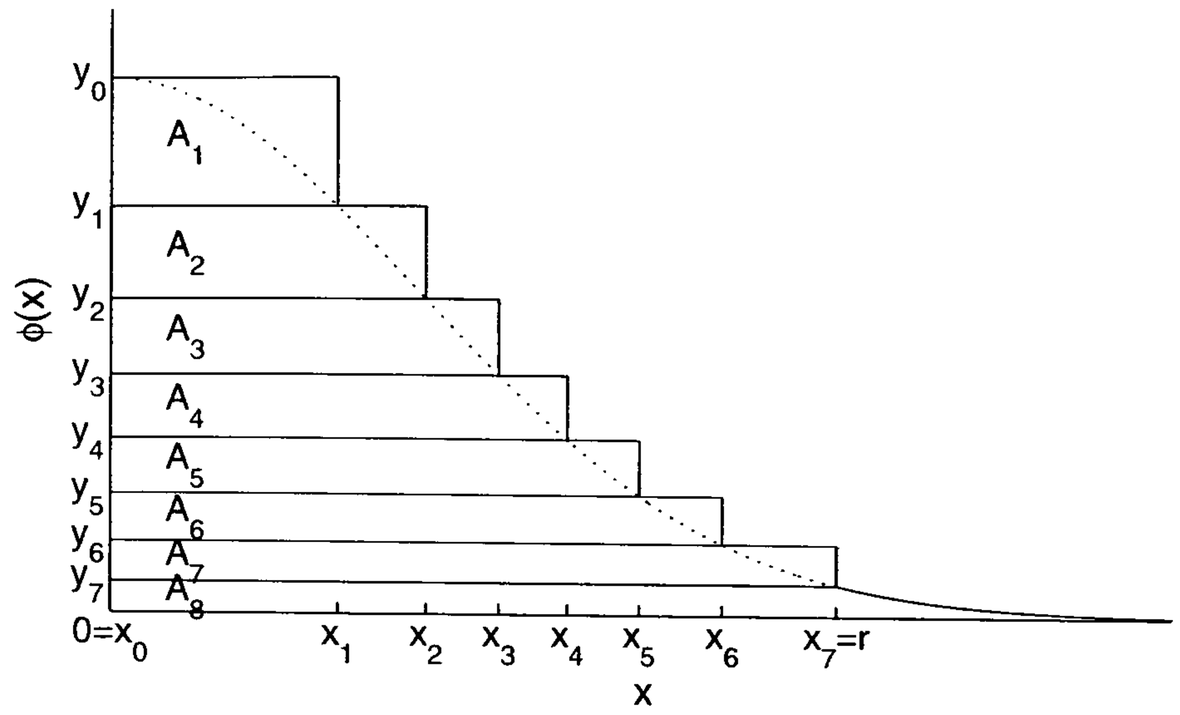


Figure 2.1: Example of the Ziggurat technique applied to the Guassian distribution. One of the areas $A_1..A_8$ is selected at random, and if a randomly generated point within the area lies under the curve it is returned, otherwise the process is repeated.

case code which can provide samples from the tail of the distribution.

The advantage of the Ziggurat distribution is that if a large number of rectangles are used then almost all samples are generated by just selecting one of the rectangles, generating a horizontal point within the rectangle, then performing a simple comparison. For the Gaussian distribution it is convenient to use 128 rectangles [141], which results in about 99% of calls being satisfied through this simple code-path. Only the remaining 1% of calls require complex function evaluation, when exponentials are used to test whether points lie under the curve, and logarithms are used to generate samples from the tail distribution.

One of the few techniques for generating non-uniform variates directly is the Wallace method [223], which can be used to generate normally (or exponentially) distributed samples without needing to transform some underlying uniform variate. This is achieved by creating a pool of numbers which are normally distributed, then applying transformations to the pool that change the individual values, while preserving the overall normality of the pool.

For example, the following two Hadamard matrices can be used to transform the pool:

$$A_0 = \frac{1}{2}\begin{pmatrix} -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 \\ -1 & -1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{pmatrix} \qquad A_1 = \frac{1}{2}\begin{pmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & -1 & -1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \end{pmatrix} \tag{2.13}$$

These can be efficiently implemented through a small number of additions, subtractions, and a division by 2. If $x_0..x_3$ are four values selected from the pool then the four transformed

values $x'_0 ... x'_3$ can be calculated using the two formulas:

$$x'_0 = t - x_0 \qquad x'_1 = t - x_1 \qquad x'_2 = x_2 - t \qquad x'_3 = x_3 - t \tag{2.14}$$

and

$$x'_0 = x_0 - t \qquad x'_1 = x_1 - t \qquad x'_2 = t - x_2 \qquad x'_3 = t - x_3 \tag{2.15}$$

where $t = \frac{1}{2}(x_0 + x_1 + x_2 + x_3)$. All values in the pool are transformed in this way, resulting in a new pool. The new pool contains a new set of values, but together they still form a set of normally distributed samples. After each transform the numbers in the pool can be output as random samples, until all have been consumed by the simulation, then the pool will be transformed again.

This method is very fast, and can be as fast as uniform number generation, but in its simplest form results in significant correlations between samples from consecutive pools. For example, if a very large number is present in one pool, then after the transformation the new pool will still contain the energy of that large number, but spread amongst four different samples. This results in a deviation from the normal distribution, as very large output samples are likely to be closely followed by an abnormally high number of large outputs. In order to reduce these correlations, it is necessary to transform the pool multiple times before using outputs [29], but unfortunately this reduces the speed advantage of the algorithm. However, even when the pool is transformed multiple times between outputs the algorithm is still fast, and can also be vectorised [28].

2.4.3 Hardware Random Number Generation

The increasing size of FPGAs is beginning to see the migration of some types of simulation and testing applications into hardware, most notably for the generation of test signals [10, 43] and Bit-Error-Rate (BER) simulations of communications channels [123, 124]. Although the standard software techniques are still available, in most cases these applications use different methods for random number generation that are especially efficient in hardware.

The first major difference is that uniform numbers in FPGAs are usually generated using bit-level Linear Feedback Shift Registers (LFSRs). These are variants of the exclusive-or based Lagged Fibonaccis described in Section 2.1, except that instead of using a large memory buffer to provide the circular feedback FIFO, registers or LUTs are used [3]. In the Virtex architecture LUTs can be configured into SRL16s [237], which provides a 1-bit wide FIFO of up to 16 stages using a single LUT. This allows up to 16 registers to be replaced with SRL16s, allowing a very efficient packing of LFSRs in the Virtex architecture [75].

Another common means of generating uniform bits is to use Cellular Automata (CAs), which consist of a linear array of simple communicating cells [234]. Each cell calculates

its state in the next cycle based on the current state of itself and its two neighbours, then all cells synchronously change into their next state. If the cells are following certain rules then the cells in the array will change state in a chaotic way. The best known example is the Rule 30 CA, so called because 30 is the value of the binary truth table used by each cell to calculate its next value. The next value of the cell at i is determined using the following formula:

$$cell_{i,t+1} = (cell_{i-1,t} \otimes cell_{i,t}) \oplus cell_{i+1,t} \tag{2.16}$$

The array of cells is usually wrapped at the edges, so in a w bit wide array $cell_{0,t} = cell_{w,t}$ and $cell_{w+1,t} = cell_{1,t}$. The evolution of the array of cells is chaotic, and can be used as a pseudo-random number generator. If adjacent cells are used to provide each output bit there are significant correlations between bits in the generated stream, resulting in serious statistical flaws. A commonly used solution is to use an array containing wk cells, and to only use every k-th cell to form the output stream [93]. By using larger values of k the quality is improved, but more area is needed to support the larger array.

LFSRs and CAs have been compared [142] both by using the Diehard [136] test for randomness, and by using random numbers to drive genetic algorithms. It is found that not only are Rule 30 CAs larger than LFSRs, they are also less random (according to the Diehard test), and run slower.

More complex CA based generators using four inputs to generate each new output have also been explored [195], as well as different 2D and 3D cell topologies. It was found that the more complex types of CA could pass Diehard, so this means that the generators provide at least a minimum standard of statistical randomness, and all were found to run at over 200MHz on a Virtex 1000, although no area metrics are reported. The tested CAs used 64-bit states, which means that the generators have a maximum possible sequence length of 2^{64}, but in practise the highest period discovered was under 2^{35}. This represents just 113 seconds worth of processing time at $200MHz$, and many of the sequences were actually much shorted than this. The chaotic nature of CAs is problematic when it come to predicting sequence length, as it is almost impossible to determine the sequence length for a given starting state without stepping through the entire sequence [48].

Non-uniform random numbers are also required in FPGA designs, but almost all effort has been concentrated on one distribution, the Gaussian distribution. One of the reasons for this focus is that FPGAs have been used in a number of simulations of the effect of noise in systems such as communications channels [123], hard disks [42] and radar signals [10]. Noise is often approximated as a Gaussian process, so these hardware simulations have driven demand for fast, high quality, Gaussian random number generators.

The Box-Muller technique has proved a popular choice for implementing Gaussian noise in FPGA, as unlike most other methods it has a regular structure, with no branches or loops, so a pipelined implementation can provide a guaranteed rate of two outputs per cycle. The main problem when translating it to hardware is the calculation of the log, sin and square

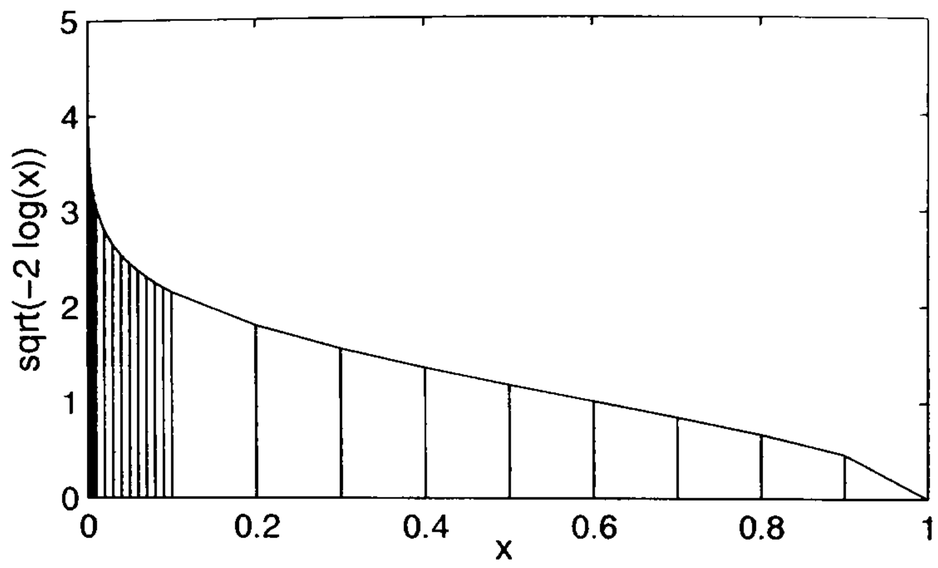


Figure 2.2: Non-uniform partion of the function $\sqrt{-2\log x}$ over the range 0..1. The range is split into ten equal sections, then within the leftmost section a further ten division are created and so on.

root functions found in Equation 2.9. Implementing high performance versions of these functions is not difficult, as they can be heavily pipelined: the difficulty is in balancing the transform accuracy and area required.

In [58] [20] the functions are approximated by using a non-uniform recursive quantisation of the functions, which uses a hierarchy of linear approximations to approximate a random variable with a PDF of $\sqrt{-2lnx}$, where x is a uniform random variate. This method splits the range of the function into 2^k segments (in the evaluated design $k = 4$) and then stores a linear approximation for the function over each of these segments. This provides an acceptable approximation for all segments except the one closest to zero, as the function heads to infinity as zero is approached. In order to approximate the function near zero, the segment is further split into 2^k segments, and another linear approximation is stored, then the process is repeated. Figure 2.2 demonstrates this scheme when 10 segments are used per hierarchy level, showing the segment division becoming finer as zero is approached. Figure 2.2 provides another view, using a log x-axis, showing the recursive division that allows the linear approximation to accurately track the curve even when the gradient becomes very steep.

The segmentation scheme can be efficiently implemented by storing the coefficients for each level of the hierarchy in a separate ROM. During each cycle a segment at each level is randomly selected, and then a chain of compares and multiplexors is used to choose the coarsest level that has not selected its first segment. Figure 2.4 shows how this chain is constructed in hardware when three levels are in use. Note that the chain can be pipelined at each multiplexor, so there is not a long combinatorial delay, even when a long chain of multiplexors is used.

Even with the non-uniform segmentation this Box-Muller method does not provide a high quality approximation, as the simple linear approximation still differs significantly from the true PDF, so the the Central Limit Theorem is used to improve accuracy. The

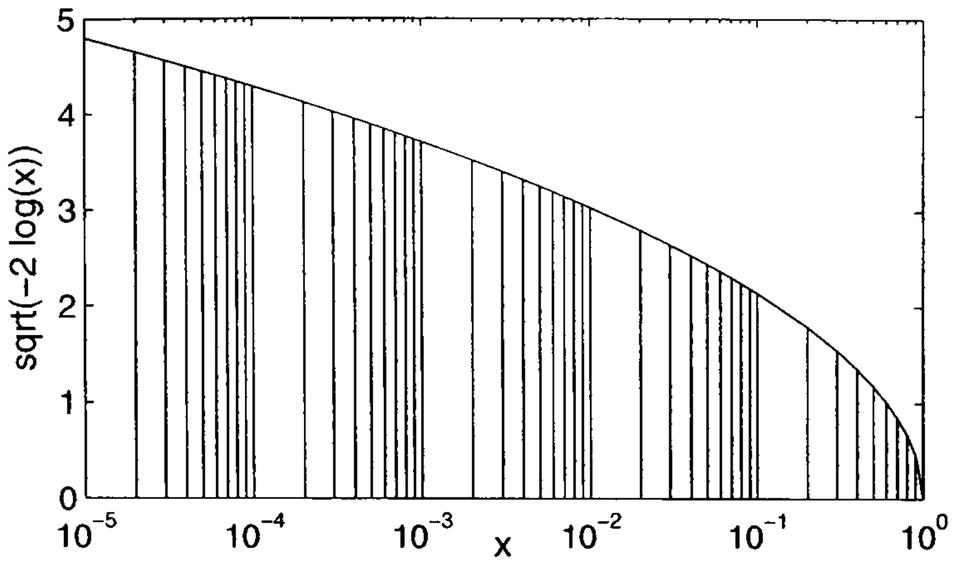


Figure 2.3: The same non-uniform partion shown in Figure 2.2, but using a log scale for the input, showing the recursive sub-division that provides finer steps as zero is approached.

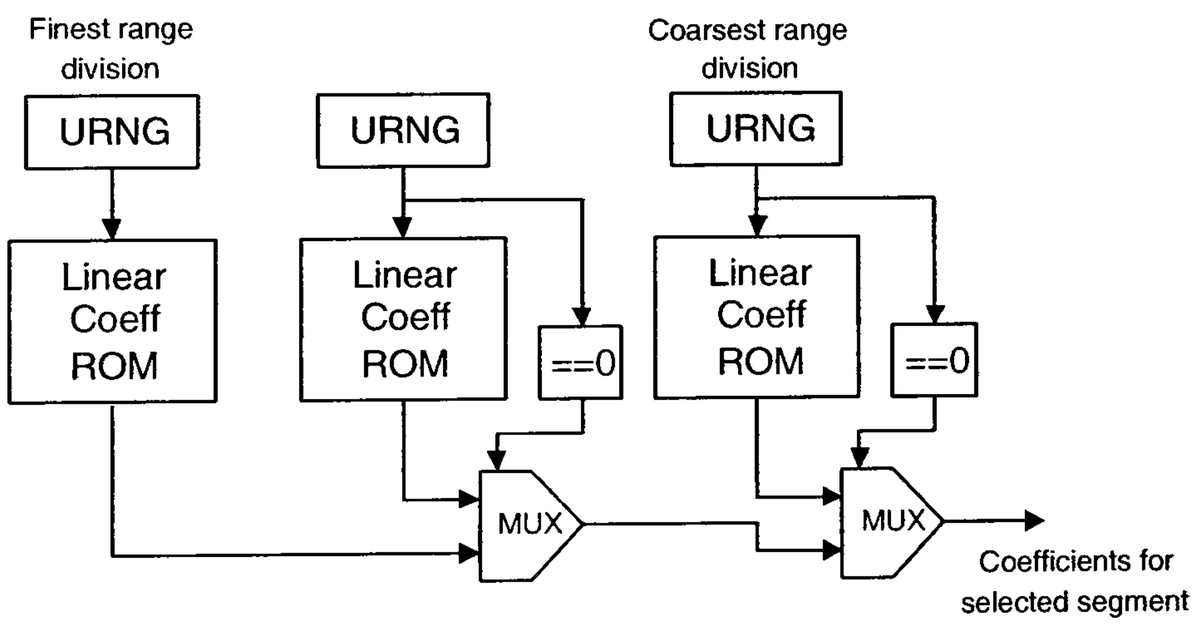


Figure 2.4: Coefficient selection circuit for non-uniform range segmentation of Gaussian generator.

Central Limit Theorem states that if any two variates are added together then the resulting variate will be closer to a normal distribution than either of the two original variates. In this case 4 Box-Muller stages are accumulated, which provides a relative error of 0.1% in the range [-4,+4] of the probability distribution. The range of [-4,+4] is the $\pm 4\sigma$ range for the N(0,1) distribution, and is the bare minimum starting range for useful Gaussian distributions; any less and the tails will not be accurate enough to be useful. Unfortunately it is not clear what is meant by relative percentage error, as the results seem to be derived from relative error scaled by 10^{-3}. This suggests that the maximum relative error is in fact 1000, a very large error.

Xilinx have produced a commercial Gaussian core [236], also based on the Box-Muller with Central Limit combination. According to the published details it uses LFSRs to index into ROMs to provide the approximation to the two factors of the Box-Muller functions, although it is not clear how the steep gradients near 0 are approximated. The quoted accuracy is 0.2% in the range [-4.8,+4.8] which appears to be better than that of the non-uniform segmented Box-Muller [58], although some doubts as to the accuracy of the core have been raised [123].

The most accurate Box-Muller based Gaussian is found in [123]. Here the functions from the terms of the Box-Muller equations are again approximated using non-uniform quantisation, but the method is much more accurate. The segmentation scheme uses a different method that provides higher accuracy near both 0 and 1 by using a symmetric division scheme about 0.5, and careful error analysis is used to control accuracy. This results in a higher quality distribution, and as a result only two samples need to be accumulated using the Central Limit Theorem. The distribution of the approximation remains accurate at values as large as ± 7, allowing simulations to accurately investigate events with very low probabilities.

Three Gaussian approximations which do not rely on Box-Muller have also been implemented in hardware. The first uses a binomial approximation to the Gaussian distribution [10], using the property that binomial distributions for large numbers of events are approximately normal. This means that by adding together 128 1-bit uniform random bits an approximation to the Gaussian is produced (although this technique could alternatively be viewed as an application of the Central Limit Theorem). The resulting distribution is accurate to $\approx \pm 6\sigma$, but the distribution is highly quantised, with very few fractional bits.

Another non-Box-Muller Gaussian random number generator is based on the Wallace generator [124] introduced at the end of Section 2.4.2. The random pool is maintained in a dual-ported block RAM, and as the transformation process is performed samples are also extracted to be output, resulting in a an output every cycle. This produces good quality results (in terms of gross PDF accuracy), of a similar level to Lee's Box-Muller, but using less than half the area. However, there is still the possibility of correlations, particularly as this generator does not discard as many samples as the recommended software version.

In [250] the Ziggurat method [140] is used to generate normal samples. The key strength

of the Zigurrat method is that 99% of the time only two table lookups, a multiply, and a compare are needed per variate. However, in the remaining one percent of cases a more complex equation must be calculated, involving the exponential function. The hardware implementation takes advantage of the differences in throughput, using heavily pipelined calculations for the common case, and an area efficient sequential calculation for the uncommon case. Although the Zigurrat method is usually only used for the Gaussian and Exponential distributions, it can actually be applied to any distribution that decrease monotonically from a single peak. The main change is in the function evaluated in the uncommon case, which needs to be customised to each distribution, so different hardware designs would be needed for each.

A distribution independent approach is taken in [151], where a table based approach is used to approximate the inverse CDF of the target distribution. The table contains regularly spaced samples from the inverse CDF to be sampled, which are used to provide a piecewise-linear approximation. Although this method is very general, allowing any distribution to be approximated, it requires large amounts of storage (262KB). Even then, the regular sampling pattern will not allow a good approximation of steep gradients and discontinuities. More complex approximations, for example quadratic or cubic interpolation, would allow the table size to be decreased, but require more logic.

2.5 Summary

This chapter has covered background material for the rest of this thesis. Section 2.1 presented some of the broad trends in computing, such as the rise of Virtual Machines and FPGAs that now complement standard CPU architectures and ASIC designs, followed by a survey of techniques for creating hardware designs in Section 2.2. In Section 2.3 some environments for hardware-software co-design was investigated, with an emphasis on the way in which hardware can be used to accelerate software applications. Finally in Section 2.4 a survey of random number generation was performed, examining both traditional software methods and the hardware methods used in FPGAs.

The first conclusion that can be drawn from this survey is that there is currently no easy way to develop hardware accelerators that can be used in many different applications, or to write applications that can access lots of different types of hardware. This forms the motivation for the IGOL Framework described in Chapters 3, 4 and 5. The second is that there is a need to develop new FPGA based random number generators that can provide high quality uniform and non-uniform random numbers (other than the Gaussian distribution), for use both as a component of hardware based simulation, and as a means of speeding up software applications. This provides the motivation for Chapters 6 and 7, where techniques for random number generation are developed, both as a way of accelerating hardware simulations, and so that existing software simulations can be accelerated through the IGOL framework.

Chapter 3

Analysis and Design

Before implementing any system it is first necessary to analyse the problem to be solved and then specify how any solution to the problem must behave. So this chapter will investigate the problem domain of hardware and software management systems and make some claims about how any such system must behave.

Firstly we look at the problem from a software engineering point of view, and try to identify the basic requirements. In particular: what kinds of hardware and software systems interact within applications, and what kind of application domains need to be considered? We then go on to create a system which takes these design considerations into account. First an overall system model is introduced, then a means of describing components within the system are defined.

The key contributions in this chapter are:

- A framework for capturing the structure and behaviour of hardware and software components
- A schema to allow information about components within the framework to be managed.

3.1 Analysis of Problem Domain

3.1.1 Target Applications

There are many kinds of computer application in the world, from basic desktop programs running on a single computer, to massive distributed systems running across thousands of computers across the world. In this thesis the focus is on applications that can be accelerated using reconfigurable hardware, which limits the scope to certain classes of application. In this section these classes of application are identified, and the performance goals and bottlenecks for each are explored.

Classes of Applications

Rather than classifying every type of application, we identify the activities or components that are used in hardware accelerated applications. These can be sorted into three different classes according to the main limit to performance in the system:

Data Bound The main bottleneck is the rate at which data can be passed through the system, so Input/Output and memory bandwidth are the key factors. Examples include image processing [21, 199, 217] and factorisation [107].

- Large volumes of data being processed.
- Frequent or continuous communication.
- Predictable communication pattern.
- Small or medium amount of state maintained.
- Small or medium amount of processing relative to amount of data.
- Main performance metric is data throughput.

Processing Bound The performance limit is the rate of calculation, so CPU speed or FPGA area/clock rate are the critical feature. Examples include ray-tracing [235], graph analysis [18, 95, 154], simulations [10, 43, 123] and vision processing [62, 143].

- Small or medium amounts of input and output data.
- Infrequent communication.
- Predictable communication pattern.
- Medium or large amount of state used during processing data (but not necessarily maintained between items of data).
- Large amounts of processing relative to amount of data.
- CPU or memory bound.
- Main performance metric is overall time to complete single job.

Latency Bound The performance limit is the latency of data-dependent IO operations, such as memory accesses or disk requests e.g. sparse matrix calculations calculations [226, 227].

- Small or medium amounts of input and output data.
- Frequent communication.
- Unpredictable communication pattern, often from multiple sources.
- Large amount of state maintained.
- Small or medium amount of processing relative to data.

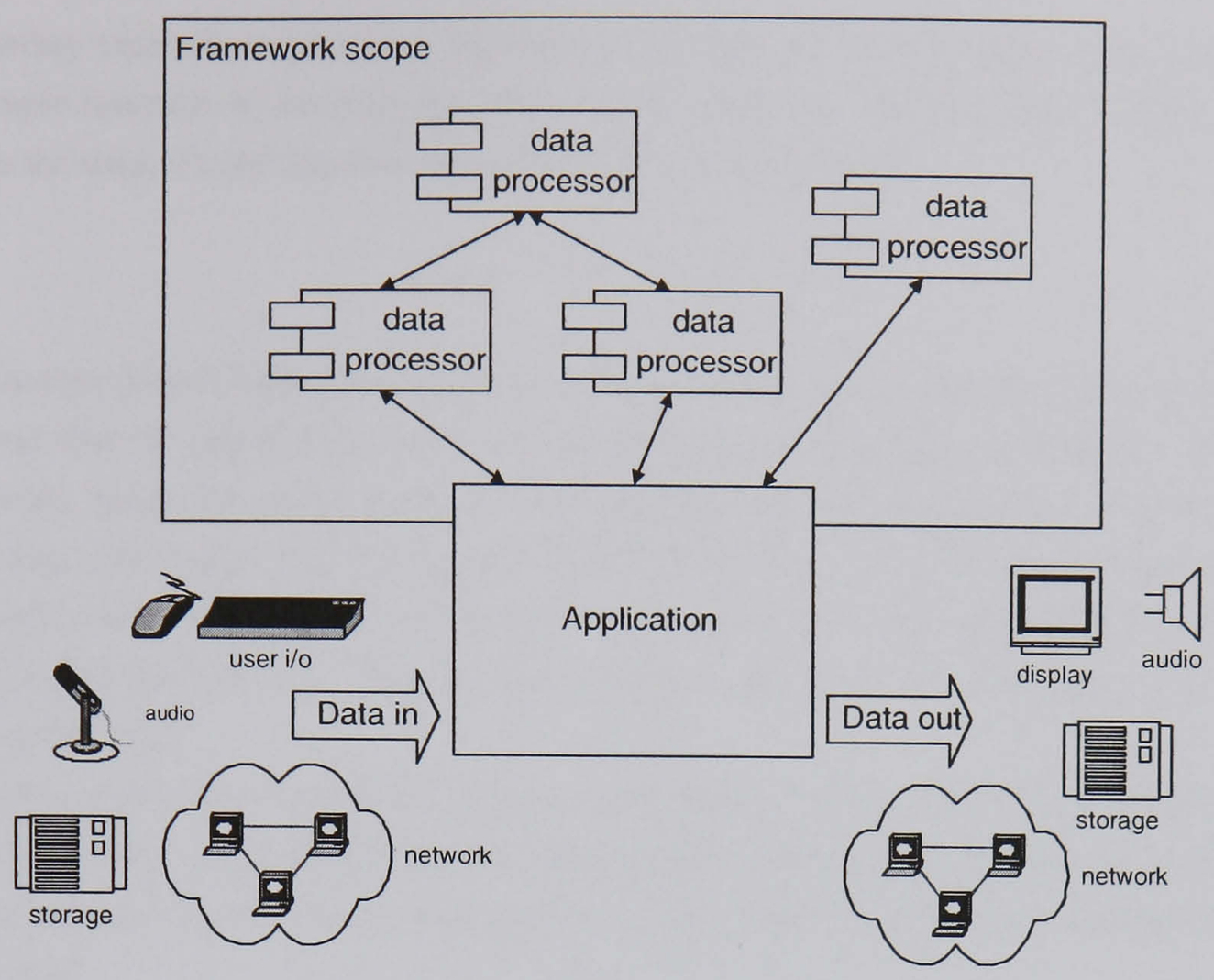


Figure 3.1: Conceptual model of data processing: data is sourced from somewhere, such as a file or network and is then processed by some application, after which the data are then sent to some data sink such as a display or another file. The framework is used within the application to provide data processing components, but it is up to the application to request and organise the data processors.

- IO or memory bound.
- Main performance metric is average time to service a request or complete a task.

Almost all data processing tasks fall into one of these categories, although some may belong to two, for example encryption can be data bound or processing bound, depending on the algorithm and architecture used. The sourcing and sinking of data are considered a task of the application, which will provide the GUI (Graphical User Interface), file or network capabilities needed. Figure 3.1 illustrates the conceptual model used in this thesis, where applications manage overall input and output, but defer to separate components in order to perform data processing tasks.

Estimating Performance

Focusing on just the data processing components, without considering aspects such as user interaction or IO, allows estimates of application performance and potential speed-ups to be made. This is particularly useful in the data bound and processing bound classes of applications, where parameters such as data and processing throughput are strongly influenced in hardware architectures by factors such as clock rate and communications buses.

In many cases the total processing time for a single processing task can be summarised as a simple function of: initialisation time, I; time taken to send input data, S; time taken to process the data, P; and the time taken to receive output data, R.

$$T_{total} = I + S + P + R \tag{3.1}$$

This assumes that all input data must arrive at the processing location before processing can start, and that no output data can be produced before processing has finished. This is an appropriate model for many hardware accelerated tasks such as image processing, matrix processing, encryption, ray-tracing and factorisation: first the hardware platform must be configured, then input data is transferred over a bus to the hardware platform, the active component of the hardware platform processes the data, then the output data is transferred back over the bus.

In some applications there are many items of data to be processed, or the total amount of data is too large to be stored and processed in one section on the hardware platform. This requires repeated transfer and processing of n data packets, but the initialisation cost only occurs once:

$$T = I + \sum_{i=1}^{n} (S_i + R_i + T_i) \tag{3.2}$$

If the input data for each stage depends on the output data from the previous stage then the execution time cannot be further reduced, due to the loop-carried dependency [194]. However, in many cases there is no dependency of input packets on previous output packets, or the dependency can be managed by caching on the hardware platform. An example of this situation is video processing, where individual frames can often be processed individually.

If there is no dependency between data items, and sufficient storage space is available on the hardware platform, communications and processing can proceed in parallel as a coarse grained pipeline:

$$\begin{aligned} T =& I + S_1 + \max(S_2, P_1) \\ &+ \sum_{i=2}^{n-1} \max(S_{i+1}, P_i, R_{i-1}) \\ &+ \max(P_n, R_{n-1}) + R_n \end{aligned} \tag{3.3}$$

Using pipelining the performance of the system is now limited by either the data transfer rate or the processing rate. The slower of these two indicates whether the system is data bound or processing bound. This pipelining technique is often used in graphics applications, such as real-time video display applications [164], as it allows the de-compression of the next frame to be performed while the current frame is displayed to the user, and compressed data for future frames can also be read from disk or the network at the same time. However, parallel data transfer and processing is not commonly used in FPGA based video processing, and all

of the FPGA based graphics applications referenced in this thesis use the simple transfer-process-transfer arrangement.

A useful simplification is to relate both processing and communications times to data size, and to assume that all data packets are the same size. This is appropriate for many (but not all) data bound and processing bounds tasks such as image processing, as both processing and data transfer times are directly proportional to pixel count [161]. Communications on many hardware platforms are also provided through a bus, such as a PCI [172] or USB [219] connection, which means that sending and receiving cannot be performed concurrently. This leads to to an approximation of total processing time:

$$T = I + n \times max(dp \times c, 2 \times bw \times c) \tag{3.4}$$

where n is the number of data items (e.g. video frames) to be processed, c is the size of each data item (e.g. number of pixels), and dp and bw provide the scaling factor for processing and data transfer speed with data item size.

To give an example of how these parameters interact in a real data processing situation we will now consider some performance parameters derived from four different versions of a video processing task. Table 3.1 shows the values of I, dp and bw for four different implementations of a generalised convolution operation [68]. The convolution operation has a number of parameters which are specified at the start of processing and remain constant for all frames processed. The four different techniques used are:

Generic Software A C++ implementation of the convolution operation which holds all parameters in variables at run-time.

Specialised Software A C++ implementation which uses templates to fix the convolution parameters at compile-time. This allows certain code-paths within the convolution to be eliminated, but means that when processing is initialised the compiler must be invoked to compile the run-time specified parameters into the executable.

Generic Hardware A pipelined Handel-C implementation of the convolution operation for use in an RC2000 platform. The design allows all parameters to be varied and has to used many full multipliers. The data transfer rate is determined the PCI bus used by the RC2000 platform.

Specialised Hardware The same pipelined Handel-C code is used, but the convolution parameters are known to the compiler at compile-time. This allows multipliers to be replaced with faster and more area efficient constant-coefficient multipliers, and for many multipliers and data-paths to be completely eliminated. The simplification of the design's logic and the reduction in overall congestion allows a higher clock rate, reducing the time taken to process each pixel. However, in order to integrate the coefficients at run-time it is necessary to compile the design at run-time, requiring a full place and route cycle before processing can start.

Implementation	I (s)	dp (ns)	bw (ns)	Notes
Generic SW	0	200	0	No initialisation costs
Specialised SW	4	75	0	Setup costs due to C++ compiler
Generic HW	0.1	20	4	Complex logic reduces clock rate
Specialised HW	60	8	4	Full hardware compile needed

Table 3.1: Parameters for four hypothetical implementations of an image processing filter in hardware and software. The specialised versions are compiled to support a specific set of parameters values, while the generic version can support any parameter values. The three constants are: I the initialisation time in seconds; dp, the time in ns process each pixel; and bw, the time taken to transfer each pixel. These values are used in the approximation of total processing time $I + n \times \max(dp \times c, 2 \times bw \times c)$.

Table 3.1 shows the performance characteristics of these four different implementations. These figures were derived from proof-of-concept code used during initial benchmarking when designing the IGOL system.

Graph 3.1 shows the total execution time of these different implementations as the number of frames being processed increases. When very few frames are being processed, the generic software version has the highest performance due to its ability to start processing data immediately, albeit at a low speed. If hardware is available then the generic hardware version overtakes the software at the three of four frame mark, and the hardware initiation costs are irrelevant by the fifth or sixth frame. Due to its large startup cost the specialised hardware cannot compete with the other methods unless more than 20000 frames are to be processed, equivalent to a total processing time of around 100 seconds. However, if it is known that a large number of frames will be processed then the specialised hardware can provide over twice the processing rate. The specialised software is only useful if no hardware is available, in which case it becomes useful if more that 120 frames are processed.

The results obtained through real experiments closely track those predicted by the simplified model shown in Equation 3.4, with a difference of less than 1% observed between predicted and observed execution times seen for the generic hardware and software implementations when more than four frames are processed. The prediction of the specialised software and hardware is less accurate as the time taken in compilation and the exact resulting performance (such as clock rate reported by hardware synthesis tools) is not certain, but predicted results are still within 5% of observed results once a period of more than $4I$ has passed.

This experiment demonstrates three points. First, simple models which reduce data processing component performance to a few parameters can be used to provide a good estimate of their resulting performance, and although this model may not be applicable in all cases, it works well for modelling image processing applications. Second, initialisation time can be an important factor in overall execution time, so it is necessary to choose implementations based on the amount of data to be processed, rather than simply choosing the implementation with the highest raw processing rate. Third, specialisation of implementa-

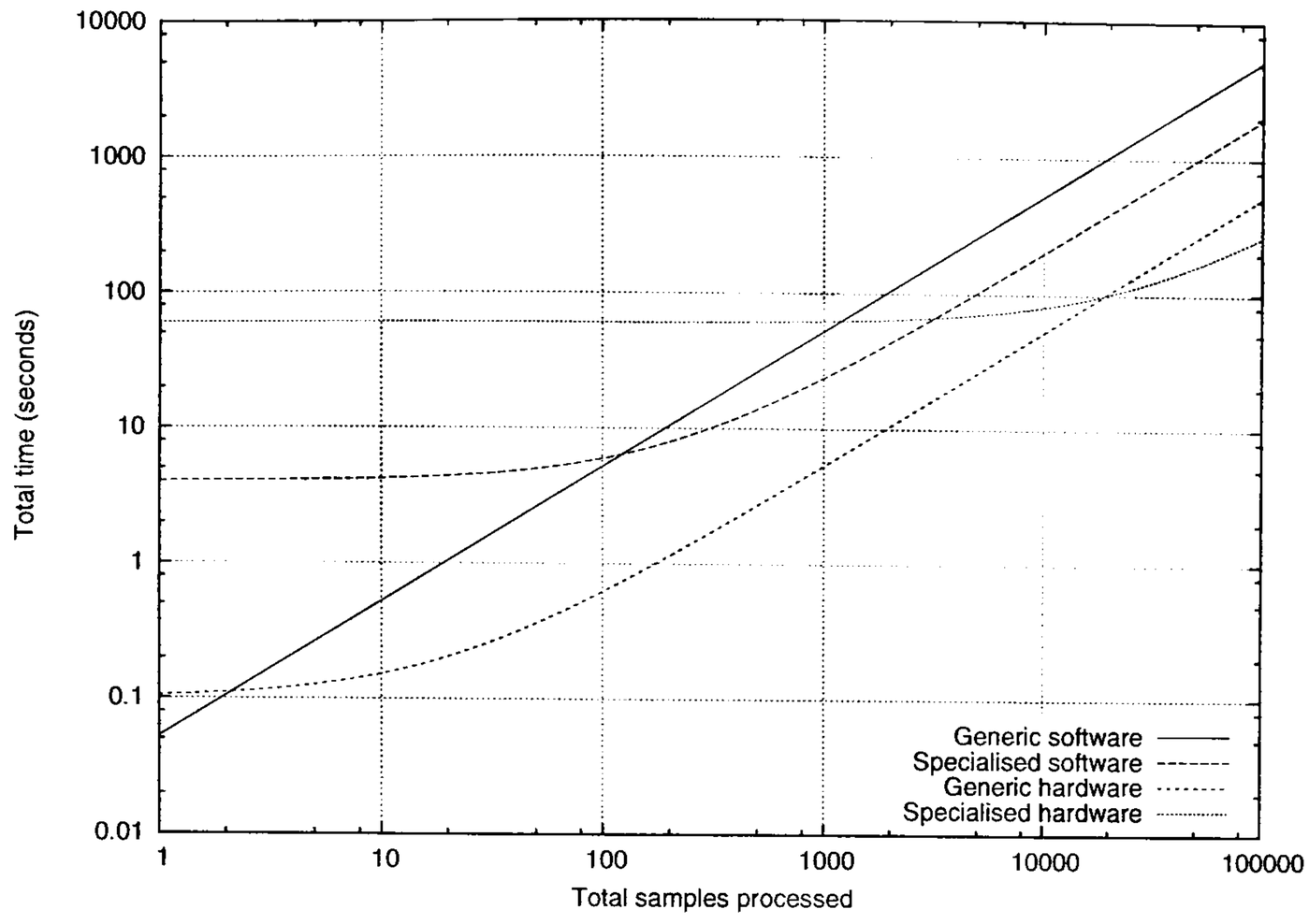


Figure 3.2: Comparison of convolution filter using software and hardware platforms with using generic and parameter specific versions.

tions at run-time can be used to decrease total processing rate in both software and hardware, as long as the number of frames to be processed is high enough to amortise the relatively long initialisation times required.

3.1.2 Hardware-Software Communication Models

Hardware accelerator platforms are attached to software hosts using many different mechanisms, such as the PCI bus [206, 66], and as pseudo-memory devices attached to the processor [126]. When such an accelerator is used to perform processing, it is necessary to use device specific API (Application Programming Interface) methods to communicate with the platform, both to transfer data back and forth, and to communicate with and control the active component of the accelerator. The active component may be an FPGA, DSP, or general purpose processor, and depending on the platform may be purely reactive [126], or may be able to perform many tasks such as initiate data transfers from host memory or raise interrupts [4]. The wide variety in platform architectures, and the different methods used to communicate with them, means that performing the same task will require a platform specific sequence of API calls.

Beyond the platform-specific API calls used to transfer data, there is also another level of communication between software and hardware, which is the communications protocol used to synchronise and organise processing. This typically requires state-machines that are understood by both hardware and software, and supply meaning to the data that is trans-

ferred between the host and hardware platform. For example, an image processing protocol requires a means of transferring both bulk data such as arrays of pixels, as well as smaller properties such as the width and height of the image. As well as providing meaning to the data passed back and forth, these protocols are also critical for controlling when processing starts and stops: many hardware resources, such as memory banks, cannot be accessed by hardware and software at the same time [22].

These two layers of communication, the device specific communications mechanism and the protocol operated over this mechanism, can be made independent of each other, as it is possible to use the same protocol with many different platforms. This requires a method for hiding the device specific access methods behind an abstraction layer, for both the software calls made by the application and the hardware access methods used within the hardware platform. An example of this type of system is the Data Stream Manager (DSM) [39] and Platform Abstraction Layer (PAL) [40] system provided by Celoxica. The DSM system provides a platform independent abstraction over software-hardware communications, by establishing a channel-based data-transfer metaphor. One or more uni-directional blocking channels can be setup, which can then be read from or written to by both hardware and software, acting as both a synchronisation and data-transfer mechanism. The DSM system is available on many hardware platforms, and allows identical code to be used in both hardware and software.

Although the DSM system provides portability among many different architectures, it achieves abstraction at the expense of performance, as transferring data via the channel abstraction means that dedicated data routes within platforms cannot be exploited. For example it is often possible to DMA data directly from host memory to memory banks on the hardware platform, without requiring FPGA intervention [206], but this is impossible under DSM. It is also often the case that data must be stored locally in memory banks before the FPGA can start processing, but under DSM the data must be streamed in over a channel, then manually stored to a memory bank. This is likely to be less efficient than direct DMA, as it requires more data-paths to memory to be instantiated within FPGA logic. Data transfer speed also becomes limited by the clock speed of the design, as the FPGA can only transfer one data item from stream to memory bank once per cycle.

The problem of managing hardware accelerators is very similar to that found in distributed object systems such as RPC [169], CORBA [198] or DCOM [64], which attempt to make remote method calls look the same as local calls. The solution used in these systems is to use a *proxy*, which is responsible for implementing the method interface locally, and a *stub*, which executes the method as a local method call on a remote system [64]. The arguments passed to the proxy are transferred to the stub through a mechanism called *marshalling*, whereby the proxy serialises the arguments into a channel, which the stub then deserialises to re-create the method call arguments. The proxy and stub do not need to know how the channel moves the data between them, and the channel might be implemented using many different types of network protocol.

Because the method call originally made on the local host has the same structure as that of the eventual method call on the remote host, it is possible to automatically generate the code needed to implement the proxy and stub from the definition of the method interface, either by generating source code that must be compiled [64], or by directly generating object code at run-time [131]. However, this is not the case when interfacing software and hardware, as it is not generally possible to implement the same interface in hardware and software. For example, pointers are critical in an efficient software based data processing interface, but do not exist in the same sense in hardware.

From this discussion there are four main abstraction levels when accessing hardware accelerated functionality from an application, which can be considered as similar to a networking protocol stack [212]:

Functional Layer The interface used by applications. This directly reflects the task to be performed, and concentrates purely on communicating what must be done and the data needed to do it, with no concern for how the task will be performed. For example, this might consist of a single function with three arguments pointing to in-memory buffers containing images, two holding the input, and one that will hold the output on function return.

Protocol Layer The protocol or state-machine used by the software and hardware to synchronise the transfer and processing of data. Using the example of image processing, this will take the in-memory image structures and transfer them to the hardware platform, first by transferring the size of the images, then by transferring the pixel data, until the FPGA can start processing.

Channel Layer The communications abstraction used in the protocol to send and receive data, for example a uni-directional channel based metaphor [39], or block transfer via DMA [22].

Device Layer The underlying device specific mechanism used to comunicate and transfer data between host and hardware platform. This might consist of calling functions in a DLL (Dynamic Link Library) in order to initiate DMA transfers [4], or reading and writing to a specific region of memory [126].

In principle these four layers are independent, and can be changed independently, but in practise some levels will be more tightly bound. For example, given a protocol that allows image processing to be performed in hardware, it is likely that many functional layers (applications) could make use of it, or given a channel abstraction, there could be multiple device-level implementations of that channel. However, the protocol and channel layers are likely to be more tightly bound, as properties of the channel (such as the synchronisation mechanism) will influence the design of the protocol.

In existing hardware accelerated applications there is usually no separation between these four levels, as the device specific calls are embedded in the application source code [88,

205]. This limits re-use, as the hardware accelerated functionality is only available through direct calls to the device API. The sequence of calls needed to achieve this will be design and platform specific, and possibly undocumented, making it difficult to integrate the hardware into other applications that could benefit. The ability to port designs between hardware platforms is also limited, as both the application and hardware design must be rewritten in order to take into account the differences in device API calls, even if the protocol between software and hardware remains the same. By using a four layered system it should be possible to achieve isolation between the different levels, allowing hardware designs to be used across multiple platforms and from within many applications.

3.1.3 Requirements

In light of the previous sections, the requirements of the system will now be defined, including the scope of problems that it is (and is not) intended to solve. The overall mission statement can be summarised as:

> A framework for specifying, accessing, implementing, deploying and testing data-processors, where data-processors are components implemented in software or hardware that perform processing tasks on behalf of a software application. Applications can request an instance of a data-processor by specification at run-time, which will automatically be provided by instantiating the fastest component (software or hardware) that meets the specification.

Further requirements for the framework are:

Simple Deployment Hardware and software components that implement tasks should be able to be deployed by copying the required software or hardware executables to the target system, then presenting the component's meta-data to the system. The component will then become immediately eligible for use in any applications that can make use of the component's functionality, even if the application is already running.

Automated Testing It should be possible to verify that a given component meets all specifications that it claims to implements, both structurally (i.e. the type of interface it exposes), as will as functionally. This eases both initial development and testing of components, as well as allowing the functionality of components from external sources to be automatically tested.

Layered Hardware Access In order to encourage portability of hardware designs between platforms the four layer hardware protocol stack will be used. As well as helping designs to target multiple platforms, this allows protocol-level software to be re-used for different designs that implement similar or identical functionality. For common tasks (e.g. image processing) this will mean that hardware developers do not need to

write any software in order to test and deploy components, as the required standard protocol can be specified in the component's meta-data.

While no restriction is placed on the types of processing that will be supported within the framework, there is an emphasis on efficiently supporting certain types of task over others, particularly those that can be implemented using hardware accelerators. Such tasks typically belong to the processing bound and data bound classes identified earlier, such as image processing [21, 199, 217], simulation [10, 43, 123, 185], and numerical computing [107, 226, 227]. These applications allow simple interfaces to be used between application and the data-processor, often following the simple pattern of providing inputs, waiting for processing, then receiving the results, which are ideal for interacting with hardware accelerator cards. Tasks which require frequent or complex interactions between application and data-processor will be less efficient in hardware, as there is often a significant fixed overhead for initiating any communication between host and software, independent of the amount of data transferred [41].

Component based systems can be classified according to the level at which component and application are bound together, and the number of places that components can be sourced from [208, 209]. The system proposed here belongs to the highest-level class, as it is both *open* and *dynamic*. The system is open, as components from any source can be incorporated into applications, whether the component was designed with the application in mind or was designed in isolation. The system is also dynamic, as applications can be upgraded in the field or at run-time simply by deploying better components. Another property of component systems is the degree of composition that is supported, i.e. the ability to describe components that are implemented in terms of other components. In this regard the system described here only directly supports *atomic* components, and provides no explicit support for describing *composite* components. However, there is nothing to stop components explicitly instantiating sub-tasks through the framework, effectively acting as a sub-application of the main application.

3.2 Framework Design

In this section a framework designed for the management of hardware and software components is presented, which allows applications to use data processing components in an implementation independent way. This uses a three tiered approach, with a layer of abstraction between applications and the components which provide data processing functionality. This allows the data processing component to be changed during execution in order to optimise performance, such as switching from a software to a hardware implementation at run-time.

3.2.1 Units of Abstraction

The first unit of abstraction in the system is the *Operation Instance*. An Operation Instance is a component which provides access to data processing functionality for applications. Each Operation Instance is associated with an Operation Specification that determines what task it will perform. The Operation Instance does not do any data processing itself, but manages one or more Operation Implementation which perform the processing on its behalf.

An *Operation Specification* is a meta-data description of the behaviour and responsibilities of Operation Instances and Operation Implementations. The specification may range from weak, where there are almost no restrictions on the behaviour of an individual Operation Instance, to strong, where the specification has very precise requirements.

An *Operation Implementation* is a component which implements the functionality described by an Operation Specification. Operation Implementations are not accessed directly by applications, but are managed through Operation Instances. In order for an application to create an Operation Instance there must be at least one Operation Implementation that implements the requested Operation Specification.

Figure 3.3 shows the relationship between these components of the system. Operation Instances and Operation Implementations represent software or hardware components, while Operation Specifications are mainly passive descriptions, although in some cases the Specification may contain a functional specification, which can then be used to provide an Operation Implementation.

Operation Specifications

Operation Specifications provide a description of the structure and functionality of a component that can be instantiated by applications, describing the way in which data can flow in and out of the component, as well as what processing will be performed. The structure of components is described using ports, a common abstraction method for components [46]. Ports are either input, output or parameter ports, and each port has an associated data-type. Functionality is described by either attaching a source code implementation or textual description to the specification, identifying an Operation Implementation that provides a reference implementation, or by giving the specification a unique identifier such as a GUID (Globally Unique Identifier) [117].

Operation Specifications may be sub-typed [210], which means that a new component description is derived from an existing one. The structural and functional properties of the base specification will be inherited by the new specification, which can also add further restrictions or properties. For example, a "Video Effect" specification might require that a components supports one video input and one video output, while a "Video Average" specification derived from it will inherit the same port restrictions, but with the additional requirement that the functionality performs an average operation.

Applications can use Operation Specifications to request Operation Instances at run-

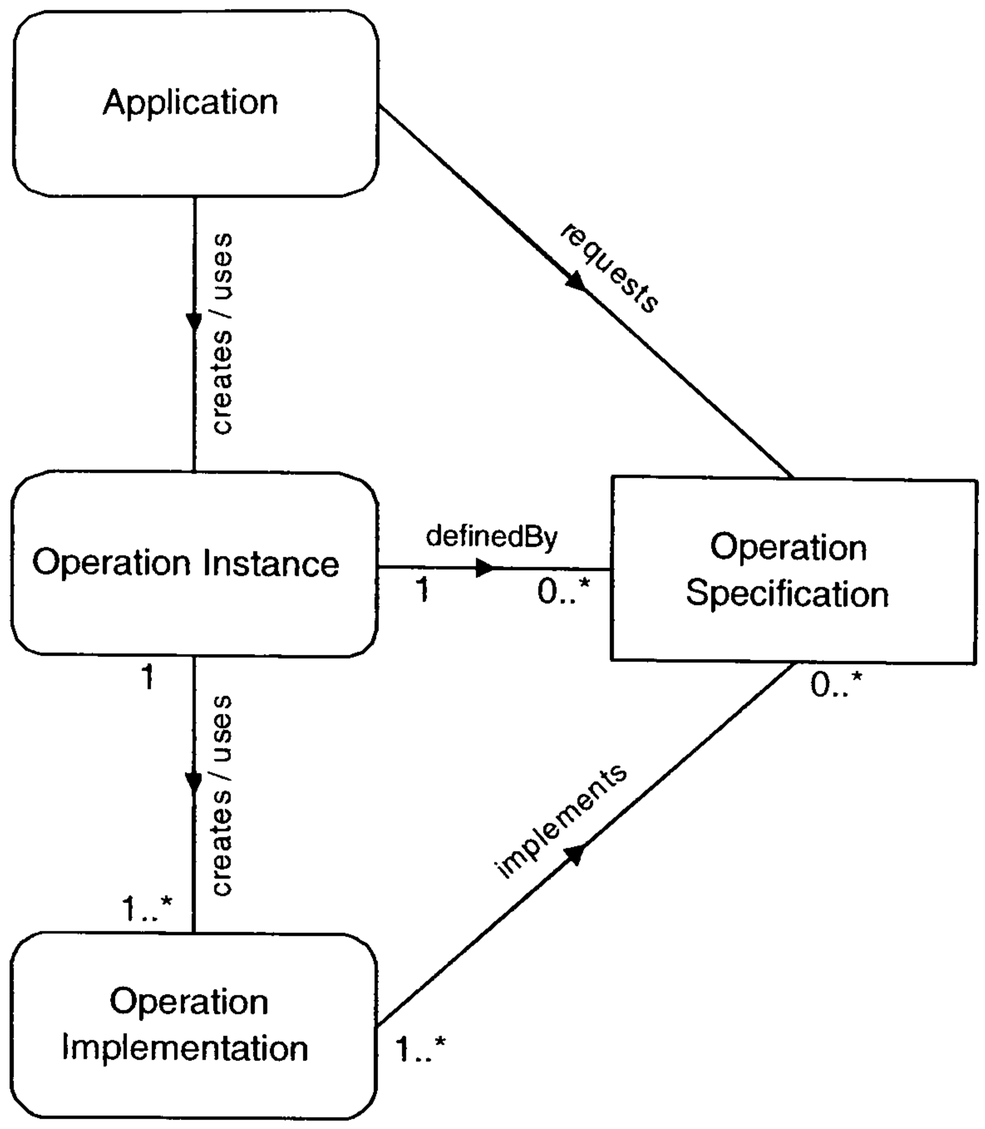


Figure 3.3: Relationship between applications and the components in the framework.

time, and the instantiated component will have the same structure and functionality as the specification.

Operation Implementations

An Operation Implementation is a component which is able to implement the functionality of an Operation Specification at run-time. The component might be implemented using software or hardware, the only requirement is that the component meets the requirements of the requested Operation Specification. Components advertise their abilities through an associated manifest, which lists the Operation Specifications which it can implement (of which there might be more than one). As with Operation Specifications, each Operation Implementation has its own GUID, and any Operation Implementation implicitly defines an Operation Specification with the same GUID. This allows applications to force selection of a specific Operation Implementation for testing or benchmarking, and ensures that every Operation Implementation implements at least one Operation Specification.

It is often the case that an Operation Implementation is required to accelerate an Operation Specification used by an application, but the full specification cannot be implemented. In software it is possible to handle multiple cases, such as many different data-types, by including multiple code-paths. Hardware devices are more restricted in the amount of code paths that can be supported, as an increase in the amount of logic often slows down the en-

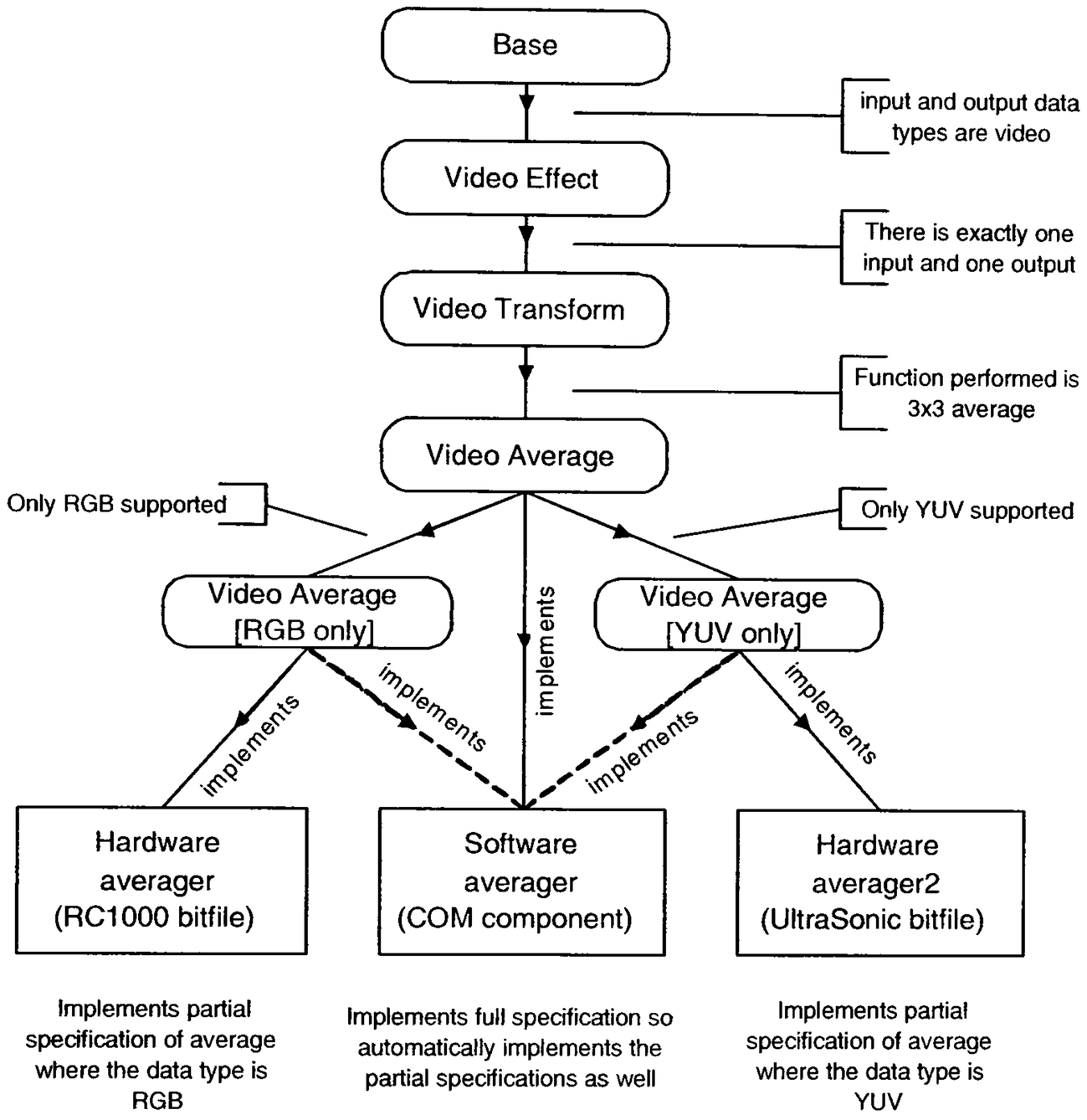


Figure 3.4: Example of how operation specifications can be related via extension to add more restrictions, and how implementations support a subset of the whole operation specification via a partial specification.

tire device, even if only small parts of the design are active (for example, see Section 3.1.1). So it is useful to be able to implement only a sub-set of the full specification, for example by only supporting particular combinations of parameters.

Another reason for not wanting to provide a complete implementation of the specification is that it simply takes more design, implementation and testing on the part of the implementer. One of the goals of this framework is to allow applications to be incrementally improved: in particular, optimising systems by providing faster implementations of data-processing bottlenecks. If analysis of a system determines that the slowest part of a system occurs only when an Operation Instance is used in a certain way, there is no need to provide a completely new implementation that is faster for every combination of parameters: only the particular mode of operation causing the bottleneck need be accelerated.

The solution is to allow Operation Implementations to derive a new Operation Specification that specifies certain restrictions on the original, such as only supporting a single data-type, or only handling a limited range of parameters. The resulting component provides a *Partial Implementation*, and can only be selected by the Operation Instance when the restrictions are met. This method is similar to techniques such as partial specialisation of templates in C++ [204], where a class can be optimised when parametrised with a set of compile-time parameters.

Figure 3.4 shows the specification and implementation chain for an averaging filter. As well as the general purpose software implementations, there are two device specific hardware implementations for the RC1000 and the UltraSonic. The two hardware versions provide partial implementations of the Average operation, reflecting the specific case which was causing the bottleneck in the application they were written for. Note that the software implementation automatically implements the two specifications created by combining the full specification with the restrictions of the partial implementations; any implementation which implements an operation specification automatically implements all partial versions of that specification.

Operation Instances

The Operation Instance provides isolation between applications and implementations. Although it would be possible to use a system where applications talk directly to components that provide implementations, there are a number of advantages to having an intermediate layer:

Uniform interface for client application developers Having a facade in front of implementations allows client applications to use a uniform interface no matter what the implementation interface looks like; applications should never need to know anything about what lies behind the Operation Instance interface.

Backwards and forwards compatibility The Operation Instance can map between clients and implementations using different versions of the interface protocol.

Reduced complexity for implementation developers By moving common functionality out of the implementation component and into the Operation Instance, the development of each implementation component requires less code and effort.

Additional services for Operation Instances Additional layers providing services can be inserted between client and implementation without any involvement of either party. This allows for automatic optimisation, as shown in Chapter 5.

The Operation Instance is only active when performing management tasks such as switching between Operation Implementations, or when providing services on their behalf, such as managing hardware data transfer. During data processing most calls from the application will be forwarded through to the implementation with very little or no processing, so the effect on overall run-time is negligible.

It is important to understand that this framework is not designed to be an all encompassing component system such as Darwin[133]. There is very little support for composition of data-processors using the common component and connector model [196], although the system does use the port abstraction common to many of these systems. The goal of this system is to allow accelerated data-processing components to be easily integrated into new and existing software applications, and efficiency and practicality are two of the most important features: elegant systems that are difficult to use or understand are unlikely to ever be used.

3.2.2 Modelling Components

In order to describe Operation Specifications and Operation Implementations it is necessary to have a corresponding language and semantics that allows such meta-data to be described and processed. The requirements for the meta-data system are:

Simple Syntax Meta-data descriptions should be easy to write and parse, both for humans and computers.

Extensibility Both the system and third-parties should be able to safely attach arbitrary information to specifications.

Safe Merging It must be possible to take meta-data from multiple independent sources, and to merge them in a way that leads to no conflicts.

Structural Descriptions Descriptions of the structural features of components such as input and output ports must be supported.

Functional Descriptions A method of describing how components should behave is needed, both for specification of the behaviour of Operation Specifications, and to describe how the functionality of Operation Implementations can be accessed.

There exist many languages for specification of software and component based systems. The most commonly used of these are languages for describing distributed object systems, such as IDL (Interface Definition Language) used by DCOM [15] and RPC [169]. These languages use a C-like syntax to describe interfaces as functional calls, but extended with attributes to identify extra properties of the function calls, such as which parameters are inputs and which are outputs. These languages are too complex for the task of describing meta-data due to their C-like syntax and types, and they also operate at a lower-level than the proposed framework. Rather than considering high-level data-flow, these languages are concerned with the detailed mechanism for passing blocks of data around, which is not the intent of the meta-data system.

There are many Architecture Description Languages (ADLs) used in software engineering to specify component systems. These include Darwin [133], UniCom [196], and SOFA/DCUP [178]. These languages typically cover many aspects of software design, including both specification of component structures, as well as how components will be connected together at run-time. The syntaxes of these languages is usually quite simple, but contain many clauses and sections (mostly to deal with complex component binding tasks), making them non-trivial to parse. Ultimately these languages are attempting to tackle a much more difficult problem than that addressed here, and so introduce too much complexity.

No existing languages that could be leveraged to describe component meta-data were identified, so instead a new language is provided, which meets the five requirements outlined above. The basis for the language is RDF (Resource Definition Framework) [87], a very general method for describing relationships, originally defined as a basis for the Semantic Web [17]. The main concept used for data representation in RDF is the statement triple, which consists of a subject, a predicate and an object. For example, the triple (*Othello,author,Shakespeare*) can be used to represent the information that Othello was written by Shakespeare. Information within RDF is represented by sets of these triples, and URIs (Uniform Resource Identifiers) are used to uniquely identify objects, allowing information from multiple sources to be merged together. RDF has been used in a number of applications, such as the XMP (Extensible Metadata Platform) format for describing media files [1], the ATOM syndication language [100], the FOAF (Friend-of-a-Friend) vocabulary for representing relationships between people [32], and as the data storage mechanism used by the Mozilla Web-Browser to contain information about plug-ins [216].

The RDF model is ideal for the meta-data language, as the language syntax is very regular and easy to manipulate, there is scope for almost infinite extensibility by introducing predicates and identifiers with globally unique names, and due to the ability to reduce all statements to triples it is very easy to merge different sets of meta-data together. RDF does not directly address the need for a structural and functional data description, but this can be provided by creating a schema for RDF [31].

Meta-data can be written directly in one of the standard RDF languages such as RDF/XML [13], RDF/N3 [16] or N-Triples [81], but these languages are more verbose and

support more features than are required for this application. A simpler way of writing specifications is provided by the Operation Specification Syntax, which is presented in Appendix A.

The port structure of Operation Specifications is described in the meta-data language by using relations to connect the main operation specification's identifier to a sub-identifier representing the port. Information about the port, such as data-type and name, is then attached to the port's identifier. An example of an Operation Specification for Motion Estimation is given below:

```
// A GUID is used to uniquely identify the specification
C3D446CF-67CF-4699-B23D-21A12C7AB6A0 {  // in the world
  type = opSpec ;   // This is an Operation Specification
  name = "Motion Estimator" ; // The user-friendly name
  inputCount = 1 ;   outputCount = 1 ;
  input = image { dataType = video ;  }
  output = motion { dataType = vectorField; }
  function = f1 {
    platform = "native" ;    format = "C#" ;
    source = "
      // source code for reference implementation in C#
    "
  }
}
```

This specification can be used by applications to request an instance of the motion estimator operation at run-time, and will be referenced by all components that provide Operation Implementations that implement the specification. An example of an Operation Implementation's manifest data is shown below:

```
EF661623-53B3-43c3-8BBD-2D5D180C24AC {
  type = opImpl ;   // This is an Operation Implementation
  name = "RC1000 Motion Estimator" ; // User-friendly name
  function = f1 {   // Information for instantiating
    platform = "rc1000/xc2v1000e" ;  // the component
    format = "bitfile" ;
    path = "http://example.org/motEst.bit" ;
  }
}
```

Functional specifications are attached to Operation Specifications either by specifying a reference implementation, or by including pre and post-conditions in the specification.

These provide verification through comparison or constraints, but do not attempt to capture the algorithm as a formal specification.

Formal semantic specifications are an area of active research, and specification languages such as Z [174, 179] attempt to capture the functional behaviour of tasks, providing a method for specifying, verifying and reusing components. Although Z has been used outside academia, it is mainly restricted to niche areas such as safety critical applications [103, 105], and documentation tasks such as standards and operating systems [101, 163]. Formal specifications have had little impact on main-stream programming, where test based methodologies such as extreme programming are more common [12, 132, 150, 167], and this is the approach used in IGOL Operation Specifications.

Reference implementations can be attached either by embedding source code code directly into the specification, or by identifying an Operation Implementation that should be treated as the reference implementation for the specification. Using an existing Operation Implementation is appropriate when parts of a legacy application are being moved into hardware. The original software implementation can be packaged as an Operation Implementation, and can be used by the application until the equivalent hardware replacement is developed, as well as allowing for functional testing by the engineer developing the hardware version.

Embedded reference implementations can be provided in a number of languages, such as C++, C# and Handel-C. For each language a standard run-time environment is defined which the embedded source code will operate within. Sand-boxed languages such as C# and Java are preferred as they allow code contained within specifications to be executed without posing a security risk [69]. The emphasis in the reference implementation should be on defining functionality, rather than providing an efficient implementation.

An alternative method for providing a functional specification is to define pre and post-conditions for the component, providing a specification through constraints [70, 125]. These are expressed by specifying C# expressions or functions that will be attached to the ports of an Operation Instance at run-time. During execution the constraint expressions will be used to check that the outputs and state of Operations Instance ports meet the specification. Constraints can also be applied to the input ports, which can be used to ensure that applications also behave in the correct way. The constraint checking functions are instantiated as objects, associated with a single Operation Instance, so state can be maintained within the constraint object in order to ensure temporal conditions, such as inter-frame effects that should be seen in video processing applications. Constraints can be used when it is easy to check that a solution is correct, but not to determine how to determine that solution. For example, writing the post-condition for an integer factorisation task is very easy, but implementing the factorisation task efficiently is not. Constraints are also useful when there is no single "correct" answer to a task, such as non-uniform random number generation. There are a potentially infinite number of correct results that a random number generator could provide, but a constraint based specification can apply tests for randomness to the sample

returned and still determine whether the specification is met or not.

Constraints are not intended to be attached to Operation Instances during ordinary execution, so assertions may be complex and computationally expensive if necessary. Only when requested, typically during testing or debugging, will constraints be attached and evaluated, and this must be specified at instantiation of the Operation Instance. The Operation Instance returned when a non-constrained version is requested does not need any support for intercepting function calls, so there is no overhead introduced in released applications. When a constrained Operation Instance is requested a specialised version is instantiated, which automatically compiles the constraint objects contained in the specification using the runtime compilation features of the .NET framework [158]. During execution the Operation Instance then notifies the constraint object as the application calls methods on the Operation Instance, allowing it to detect constraint failures.

The constraint mechanism resembles the Aspect-Oriented Programming (AOP) paradigm [49, 177]. An AOP approach to applying specifications in the form of pre and post-conditions and other assertions using AsmL has been applied in the .NET framework [152] which allows assertions to be applied arbitrary objects without their support. However, the approach proposed here is much more limited, and rather than weaving [52] the constraints into the Operation Instance object without its knowledge, it is the object itself that is responsible for inserting the constraint object into the Operation Instance control flow. The constraint model is also imperative rather than declarative [228], and assertions may contain loops, side-effects or any other feature of the C# language. While a declarative model has some potential advantages, this is an active research problem, and current work in the area concentrates mainly on enforcing interaction patterns between method calls [152]. Imperative constraints are easy to write, and allow complex post-conditions to be expressed concisely without requiring complex logic statements. For example, specifying that the samples of a Gaussian noise generator must pass a χ^2 test [173] using 32 equal sized buckets is simple to express imperatively, but difficult to express declaratively (at least under contemporary declarative specification systems).

The intent of including functional specifications into Operation Specifications is to allow automated testing and debugging of components that purport to implement some Operation Specification. However, reference implementations and constraint based specifications can only be tested if there is test data available. Data sources to provide input data can be specified in the Operation Specification, by providing a link to a source file such as video file or image file, which can be loaded at test-time and processed using the Operation Instance. When it is inconvenient to store the data in a file, for example when using a custom data format, or when very large amounts of data must be sourced, a programmatic data source can be used. This identifies an Operation Specification that has no input ports, and can be used to provide the data needed during testing.

When using constraint based specification a data source is sufficient by itself, as post-conditions on the output ports of the Operation Instance will check for correct operation, but

if only a reference implementation is specified then more information is needed. As well as a source for the input data, a source for the expected output data can also be specified, along with an Operation Specification that identifies a two input comparison function. The comparison function can do either a direct binary comparison (the default), or allow results that are within a certain tolerance of each other. For example the IEEE standard for DCT accuracy allows deviations from the exact solution, as long as the errors remain below a specified level [97].

The specification system provides three ways in which a functional specification can be included in an Operation Specification:

1. Reference implementation, input data source, and comparison method.

2. Constraint specification and input data source.

3. Input data source, output data source, and comparison method.

While these three methods may lack the purity and elegance of the formal declarative systems found in specification languages, they make it possible to provide a working functional specification for Operation Specifications.

3.3 Summary

In this chapter we have examined the problem of how to manage systems containing reconfigurable data processors. We define a model which consists of three components:

- Operation Specifications, which are component specifications that allow applications to request concrete components at run-time.

- Operation Instances, a component used by applications to access functionality at run-time. These do not implement the functionality themselves, instead they will instantiate and manage multiple Operation Implementations on behalf of applications.

- Operation Implementations, components which provide the required functionality in hardware or software. These are not accessed directly by applications, instead they are managed on the application's behalf by an Operation Instance.

We have also defined a system of meta-data for describing Operations Specifications and the Operation Implementations provided by software and hardware components. Using this we can store information about different operations and their relationships, and by creating a system of meta-data that only uses declarative statements, information from many sources can be merged without creating conflicts within the meta-data.

Chapter 4

IGOL: Implementation and Services

This chapter describes the IGOL framework, which is the implementation of the ideas described in the previous chapter, supporting the idea of abstract data processors with multiple implementations in software and hardware. IGOL stands for *Imaging and Graphics Operator Libraries*, but the framework has many applications outside the realm of video and image processing, such as audio processing and random number generation.

The main contributions of this chapter are:

- An implementation of the core components needed to support the abstract data processor framework.
- A set of plug-ins for third party applications, demonstrating the use of IGOL operations in other plug-in environments.
- A device abstraction API, used to provide a standard mechanism for communication with FPGA devices from software.

The chapter is organised as follows: Section 4.1 describes the core components of the framework, consisting of interfaces, a registry for meta-data, and the operation activator; Section 4.2 provides an overview of the third-party applications supported by IGOL; and Section 4.3 outlines the device abstraction API, as well as some of the devices it supports.

4.1 Framework Implementation

The IGOL framework is built around a set of four core features, which work together to provide Operations Instances for applications, by selecting and instantiating from available Operations Implementations. The three main components are:

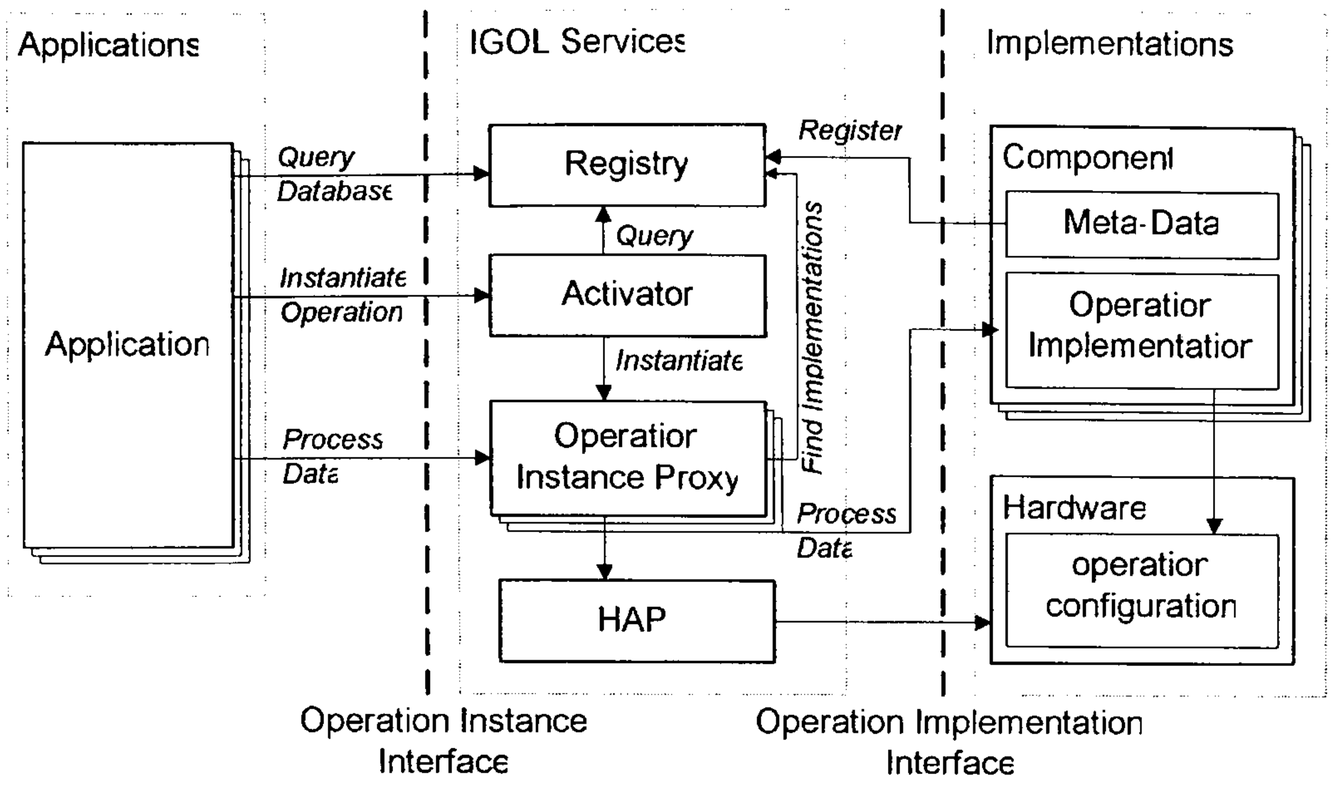


Figure 4.1: Overview of the main components in the IGOL architecture, consisting of applications which use functionality, implementation components that provide functionality, and the IGOL services which act as a matchmaker and intermediary between the two.

Interfaces The IGOL framework defines a number of core interfaces, which define the way that applications communication with IGOL services, and how the IGOL services then communicate with implementation components. There are two levels of interfaces, as applications never communicate with Operation Implementations directly, only through an Operation Instance supplied by IGOL.

Registry The Registry is a data-store which holds information about IGOL Operation Specifications and components that are able to provide Operation Implementations. Information is added to the data-store when components are installed on a machine or through communication with external sources over the network, and can then be queried by any interested party, such as applications and other parts of IGOL.

Activator The Activator is responsible for connecting requests by an application for an abstract Operation Specification to a concrete Operation Instance. The activator manages the instantiation and configuration of proxy objects that provide the Operation Instance interface used by applications.

HAPI IGOL provides a device independent mechanism for communicating with hardware. This is used internally by IGOL to discover and use devices at run-time, but may also be used independently of IGOL by other applications.

Figure 4.1 shows the overall architecture of the system, which is split into three main sections. Applications are shown on the left, and these only interact with services provided by the IGOL framework, allowing them to perform actions such as discovering what kinds of Operation Specification are installed in the machine by querying the Registry, or instantiating an Operation Instance via the activator. On the right are shown the Components that

provide Operation Implementations, which use the IGOL services mainly to add meta-data to the Registry when initially deployed on to a machine. When an Operation Instance is requested at run-time, the framework uses information in the Registry to determine which components can instantiate a compatible Operation Implementation. Applications and implementation components have no direct communications, and only interact via the services provided by IGOL. Operation Instances access hardware devices through the HAPI interface, which provides a common interface for different hardware platforms.

4.1.1 Common Interfaces

A key feature of components based systems is that there are well-defined interfaces that allow components to interact [196]. The IGOL framework uses two levels of interfaces, the higher-level of which is the structural and function specification mechanism provided by Operation Specifications described in Section 3.2.2. However, Operation Specifications are described in terms of input and output ports, and say nothing about how these mechanisms work at run-time. In IGOL this lower interface level is defined and implemented using COM (Component Object Model) interfaces, which define the way in which ports described in Operation Specifications in terms of memory buffers and function calls.

COM is a software architecture that allows component systems to be built through the use of objects that support one or more well defined interfaces, where each interface represents some task or aspect of the objects functionality, and applications access these interfaces at run-time through interface pointers. Interface pointers also control object lifetime through a reference counting mechanism, allowing interface pointers to be safely passed between components, as the object providing the interface pointer is only destroyed when all interface pointers have been released. COM interfaces follow a binary standard for function calls, based on C++ virtual method calls [23], and the structure of the interfaces is defined using a C-like language called IDL (Interface Definition Language). Each interface is identified by a GUID (Globally Unique Identifier) [117], which identifies both the structure of the interface, as well as providing a functional identity [182], though it includes no functional specification.

COM is used in IGOL as it is has wide support across many languages and platforms, and because it offers an efficient calling method that introduces almost no overhead when a function call through an interface is compared to a normal function call [23]. Many languages are able to use COM interfaces, such as C++, C#, Java, JScript and VBScript. Any language that is able to use COM interfaces is able to access IGOL services and process data using IGOL Operation Instances, and languages that are also able to implement COM interfaces can be used to create Operation Implementation components. COM is most commonly found on the Windows platform, where it is used to provide operating system services, but implementations of the COM framework are also available for UNIX [82], and the compatible XPCOM system used in Mozilla is available on almost all contemporary

platforms [216].

Three COM interfaces are always exposed to applications by Operation Instances:

IOpInstance This is the key interface for operation clients, as it represents an actual Operation Instance which can be used to process data. This interface does not itself allow data processing, but it allows basic information and state to be queried, and acts as the base from which more specific functionality is discovered.

IOpParameters Exposes information about parameter ports and their current state. This allows applications to discover the number and type of parameters at runtime, as well as to alter the values.

IOpDataTransform Provides a basic bulk data transformation suitable for the majority of data processing jobs. The IOpDataTransform interface is retrieved from the generic IOpInstance to allow data to be transformed. More specialised transfer can be handled by implementing other interfaces which are more efficient for certain cases, but this interface is always available. This ensures that all applications can use Operation Instances, even if they do not support an task specific interface for data transform.

An Operation Instance may expose more interfaces than this set, but applications can always rely on at least these three interfaces being present. For example, Operation Instances in the current version of IGOL expose an informational interface that allows applications to determine what kind of component is currently providing the Operation Implementation, so that this information can be shown in a GUI status display. However, applications must not rely on this interface being exposed, and must use the COM interface querying mechanism at run-time to discover if it is supported.

The three interfaces exposed to applications are designed to be easy to use from applications, even if this requires more complexity within the Operation Instance component. For example, the interfaces are all exposed in both early-bound (high performance) and late-bound (scriptable) versions [23], as this makes the interfaces easy to use from scripting languages such as JScript. The interfaces also include many additional utility calls, such as the ability to specify the value of a parameter port using a filename, which is then automatically read and parsed into the appropriate data type, such as an image. Other parts of the interface allow the same task to be performed in more than one way, and provide composite methods that wrap common sequences into a single method. This actually goes against standard interface design doctrine [208], as the set of methods provided by an interface should be minimal and orthogonal. The guidelines are not followed in this case because moving complexity into the framework and out of applications is more important than purity of interface design; many applications will be written to use the interfaces, but only rarely will the interfaces need to be implemented.

The interface used by Operation Implementation components is not the same as that exposed by the Operation Instance, as many methods called by the application, such as

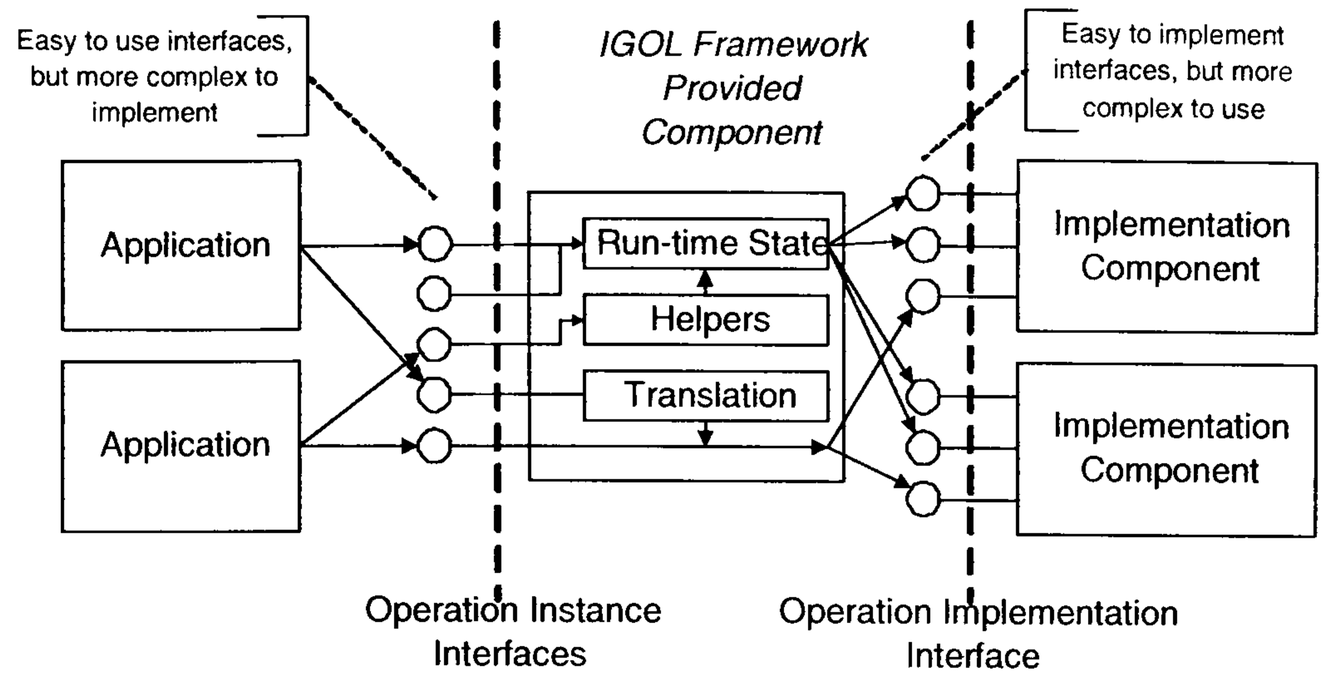


Figure 4.2: Diagram of relative interface complexity at the levels between applications, Operation Instances, and Operation Implementations. Applications access a set of interfaces that are easy to use, while implementations implement a set of interfaces that are easy to implement. The framework provides the mapping between the two interfaces, translating and forwarding calls as necessary.

queries for run-time state, can be handled directly and so are not needed. It is also desirable that implementation components should only have to implement a very minimal and simple interface, reversing the requirements used when designing the interfaces that applications use. For this reason components providing Operation Implementations only need to implement a single interface, called **IOpImplementation**, which includes only the minimum set of methods needed to support data-processing. The Operation Instance is responsible for translating between the two interfaces, translating and forwarding calls as necessary. Performance critical methods such as those used during data-processing to pass input and output data have direct analogues on both interfaces, and so pass through the Operation Instance with a negligible effect on performance.

Figure 4.2 shows how the two interface levels work together. The key feature of this architecture is that all the complexity is concentrated in the components supplied by IGOL, rather than in applications and components created by users of the framework.

4.1.2 Registry

In Section 3.2.2 a system for describing operations using meta-data was described. This provides a common vocabulary for applications, components providing Operation Implementations, and the IGOL framework components, but a means of publishing and discovering this information is needed. The *Registry* is a service provided by the framework that forms a central point for storing and querying meta-data about Operation Specifications and Operation Implementations.

The Registry is a COM component that exposes two main interfaces, one for adding and removing information to the database, used to merge and remove meta-data supplied by

components when they are added and removed to the system, and another for querying the information held by the Registry. Applications can query the Registry at run-time to discover the available Operation Specifications, for example to populate a GUI with a selection of available video effects. The Registry is also queried by other parts of the IGOL framework services, such as the Activator and Operation Instances.

Information within the registry data-store is held at three nested scopes, which provide isolation of meta-data. For each scope the registry holds a set of data introduced by that scope, which is combined with all data from the containing scope to provide a merged view of all the knowledge within that scope. The three different scopes used by the registry are, from outermost scope to innermost scope:

System This contains information shared within the local computer. All processes on the local machine can read the information at System scope, but cannot change it unless they are in the Administrator group.

User This contains information that is only visible to the current user, and is persistent between processes and login sessions. When queried, the User level data is presented as a merged view with the System data. Users are free to change information in their own User scope, but are unable to change or read information from User scopes for other users.

Process The Process scope contains all the persistent data from the User and System and Global scopes, but also has a store of information that only the registry within the local process can access. When the process terminates this information is discarded.

The different scopes allow information to be selectively shared within the system, and provides isolation between users. For example, a non-administrator cannot make persistent changes to any scope except for the User scope that belongs to them. This allows administrators to control what is published within the machine, isolating users from malicious or incorrectly implemented components that another user has created. During the development process users can modify information within their own scope, then only publish the release version to System scope, preventing the problem where users can see and use operations which are still being debugged.

The registry is implemented as an in-process COM component [23], which can be instantiated by any component which needs to access the Registry. Under standard COM activation policy, every time a component is instantiated the interface returned will be for a unique object. However for the registry it is necessary to have just one instance active in each process, as this will eliminate consistency problems where changes made in one instance do not propagate to other instances. In order to ensure that only one instance of the Registry exists in each process, it is created using the Singleton design pattern [23, 72]. On creation of a Registry instance, a check is made for any already existing instance, and if one

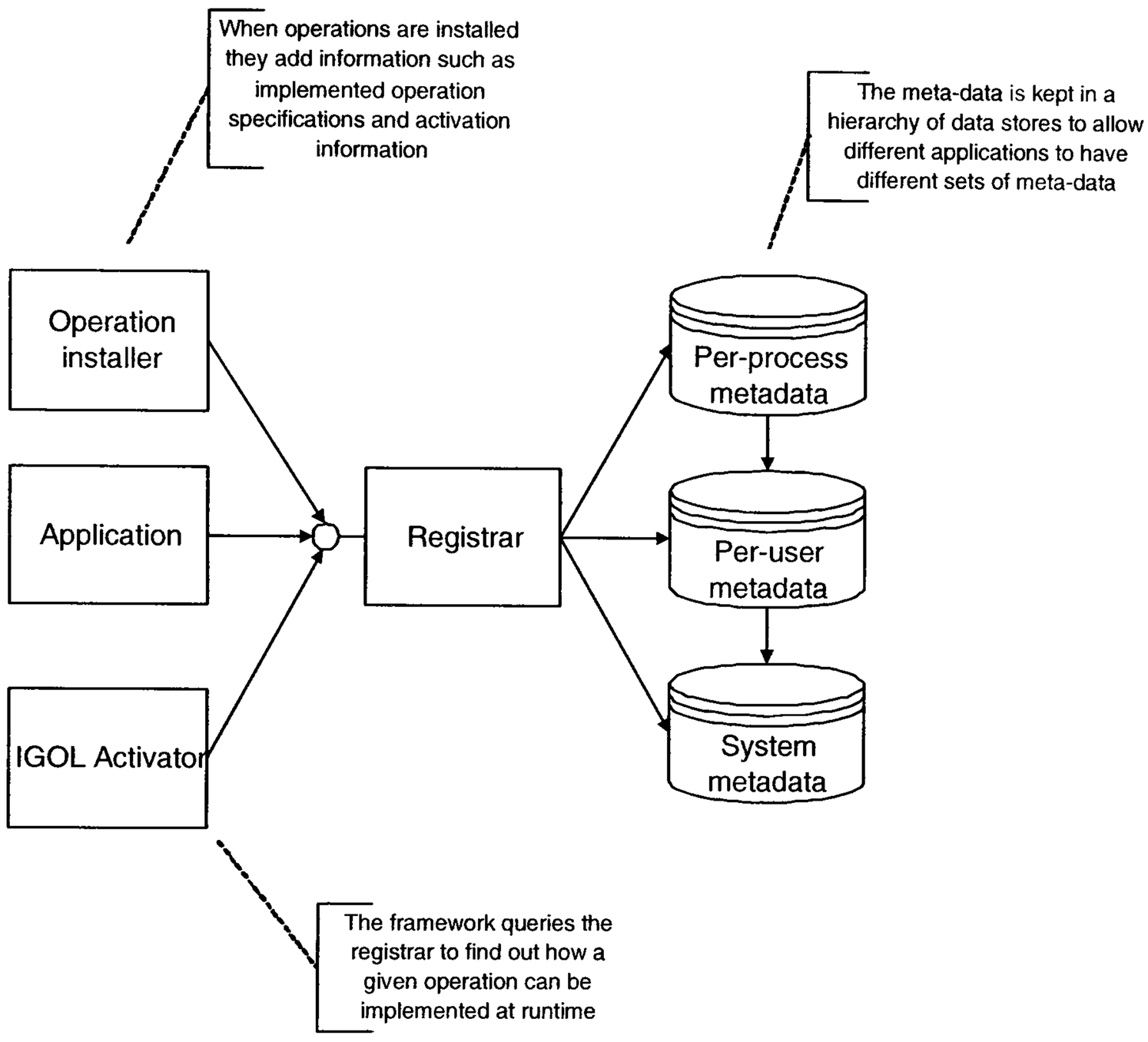


Figure 4.3: The Registry provides access to data stores at a number of nested scopes. Anybody can read data from containing scopes, but only administrators can write data to stores below User scope.

exists this will be used. Otherwise creation goes ahead, and the newly created Registry will be the sole instance.

The Registrar storage mechanism is implemented using a custom database format, which maintains a binary representation of the meta-data triples. This format acts as both the in-memory representation, and the persisted format stored in files. The database is held using a large contiguous memory block managed using a garbage collection scheme, so that when meta-data is removed from the database the space can be reclaimed. When data is added to the system the triples are stored either in free space, or if the database is full then more space is allocated at the end of the block.

The system level and user level persistent databases are stored in a file, which is then memory mapped by registrar components. Registrar instances in different processes all map the same file, resulting in the memory pages being shared between processes. All Registrars can simultaneously read from the mapped storage, but when modifications are made the system is locked using a system wide mutex. Changes are made by one Registrar in separate copy of the database, then once completed it signals all other Registrar instances to switch to the new data. Process level databases are maintained in the same format, but using a memory block that is not backed by a file, so when the Registrar is destroyed no information remains.

4.1.3 Activator

The Registry contains information about Operation Specifications and Operation Implementations that can be used to implement them, but in the IGOL architecture applications only communicate directly with Operation Instances. Applications acquire Operation Instances at run-time through the *Activator*, which turns requests from applications for an Operation Specification into an Operation Instance that can be used to process data. The activation and execution process uses the following steps:

1. The application prepares a request to be passed to the Activator detailing its requirements. The main component is the GUID (Globally Unique Identifier) of the Operation Specification to be instantiated. This may refer to the implicit Operation Specification of an Operation Implementation (see Section 3.2.1), allowing the application to force a specific implementation to be used for testing or benchmarking. The application can also attach arbitrary properties to the request, using a binary in-memory version of the Operation Specification language. These properties can be used to influence the choices the framework makes, for example by providing a guess of the total amount of work that will be performed by the Operation Instance or specifying a partial specification that fixes the value of parameters. Any of these properties can be marked as "required", meaning that the framework must either understand and honour the request, or fail to create the Operation Instance. All other properties are considered as hints, and the framework is free to ignore them. Once the request has

been prepared it is passed to the Activator.

2. The Activator prepares a list of all implementations that are stored in the Registry, including partial specifications, along with performance meta-data such as the initialisation time and processing rate for each implementation. This list is organised into a schedule that dictates the optimal implementation to choose under different circumstances (described in more detail below).

3. An Operation Instance is instantiated to provide the interface to the application. The Activator will choose the specific component to instantiate based on the applications request. For example if the specification requests that constraints from the Operation Specification be applied, then an Operation Instance that supports hooks for constraints must be used. The Operation Instance will then be passed the Operation Specification that it is implementing, and a schedule of possible implementations that it can use. Using the structure described in the Operation Specification, the Operation Instance creates the appropriate data structures needed to support input, output, and parameter ports.. The Operation Instance is now completely constructed and can be returned to the application.

4. The application uses methods on the Operation Instance to set parameters and configure any options, and then starts processing of data. At this point the Operation Instance will be forced to instantiate an Operation Implementation (if it has not already prepared one), by working its way through the schedule of possible implementations until an Operation Implementation can be successfully instantiated. It may not always be possible to use the fastest implementation due to resources (such as hardware accelerators) being unavailable at run-time. The fastest implementation may also have a very long initialisation time due to a compilation or specialisation step, in which case the Operation Instance can instantiate and begin initialisation of that component, but must also instantiate an interim Operation Implementation that can be used to process data until the faster version is ready to use.

5. During execution the Operation Implementation monitors the state of the system, and may attempt to switch the implementation currently in use in order to optimise performance or to maintain correct behaviour. For example, if the active implementation only supports a partial specification and the application changes a parameter to a value that is not supported under the partial specification, then the implementation must switch to a different implementation that can support the new parameter value. Other changes that may cause implementation switches are: new components registering meta-data with the Registry; hardware resources that were previously unavailable becoming free; and the initialisation of a faster implementation finishing (for example the compilation of a specialised implementation). As these events occur the schedule

will be amended if necessary, and the active implementation may be changed as a result.

The schedule created by the Activator tells the Operation Instance how to move between implementations under different conditions. The list of known implementations $S = s_1...s_n$ is formed, along with a list of associated performance metrics $p_1...p_n$, determined from the benchmarking meta-data in the Registry. Some implementations may also depend on certain conditions being true, for example if they are partial specifications, are waiting for an initialisation process to complete, or rely on hardware resources being available. The current set of conditions can be represented as a vector of boolean variables $C = c_1...c_k$, which will either be true or false at any point in time, and changes in the environment occur when $C' \neq C$, i.e. one or more of the conditions has changed. Each implementation s_i has a set of required conditions $R_i = r_1...r_k$, where each r_i corresponds to a condition c_i that must be true. Some implementations do not have any conditions, so $R = 0$, while others may depend on multiple conditions, such as requiring that both a specific parameter value being set, and that a specific hardware platform is available.

Under any set of conditions C_t it is possible to determine the set of implementations that can be used:

$$\{s_i \in S : C_t \wedge R_i = R_i\} \tag{4.1}$$

The best implementation can then be found by choosing the implementation with the best performance p_i.This will behave correctly when two implementations for the same hardware platform are available, as both will be depend on the same condition bit. If the condition bit is set at run-time, indicating that the hardware is now available, then the faster of the two will automatically be selected.

This method is efficiently implemented in software by using bit-wise operations to manipulate multiple condition bits at once. If the schedule is stored in a linked list, sorted by performance, then this allows the best implementation to be selected in time proportional to the number of implementations in the schedule. New implementations can also be inserted into the schedule in linear time (due to the scan through the list). If an implementation added to the list introduces a new condition bit c_{k+1}, then entries already in the schedule will always have $r_{k+1} = 0$, so it is not necessary to add more storage to support them.

The types of condition supported by the Operation Instance are:

Availability of Hardware Resources Hardware platforms can be exclusively locked by processes [4, 22], so even if hardware is installed, an Operation Instance may not be able to access it. Changes in the condition are detected by polling the device API to determine when a device becomes free, as if a third party application is using the hardware there is no other way to detect when the exclusive lock is released. IGOL Operation Instances co-operate to tell other processes that hardware has been released, by signalling a system wide synchronisation event, allowing almost instant detection of the condition change when the component releasing the hardware uses IGOL.

Partial Specification Restrictions The restrictions specified in a partial specification, such as a restricted range of a parameter, are associated with a new condition bit in the schedule when the partial implementation is added. When parameters are changed the Operation Instance checks the restriction, and if it has changed updates the associated condition bit.

Completion of Component Initialisation Some components require significant initialisation times to allow them to compile a specialised component, or download a resource. When such an implementation is added to the schedule, a new condition bit is also attached, and when the initialisation is complete the condition bit will become true.

The Operation Instance monitors the set of condition bits and if there is a change it will determine whether an implementation switch is indicated. If an implementation switch is indicated because a condition became false, then the implementation *must* be switched, as the current implementation can no longer by used. However, if the switch is indicated because a condition became true, then the Operation Instance can choose whether or not to switch. Under these circumstances the selection algorithm used in IGOL will not switch straight away, and will wait to make sure that the condition does not change back again within a short period of time (the delay is determined by the current processing rate) before moving to the new implementation. This is because applications often change many parameters at once, so it is better to wait until the whole batch of changes has been made before switching.

In this section we have seen how the core of IGOL is constructed using three core components: Common Interfaces, a shared Registry, and an Activator. By having well defined interfaces coupling between components in the framework is decreased, increasing independence between the implementation details of applications, the framework and implementation components. The interfaces are also designed to make both applications and implementation components easy to write, by moving complex operations within the Operation Instance. The Registry provides a single point for publishing and discovering information about operation specifications and implementations. Finally, the Activator is responsible for satisfying application requests for operation instances with the fastest implementation possible, by creating a schedule of currently possible implementations and instantiating the fastest one.

4.2 Plug-In Support

One of the successes of IGOL has been its integration into a wide range of existing applications, making hardware accelerated functions available in them all. These applications are mainly oriented toward video processing, but also include audio processing and general purpose video processing. Here we briefly mention some of the applications and APIs in which IGOL has been used, and identify characteristics of applications that benefit the most from the framework.

4.2.1 Applications Supported by IGOL

The two main areas in which IGOL has been successfully employed are random number generation and video processing. A summary of supported application areas and the plug-ins used to support them is now presented.

Random Number Generation

High Quality random number generation is a critical part of any simulation or stochastic algorithm, and as is shown in Sections 6.3.5 and 7.3.3, FPGA based random number generators can offer a significant speed-up over software based implementations. The speed at which random numbers can be moved from a hardware generator into host memory is limited by the bandwidth of the bus between software and hardware, for example the 32-Bit/66MHz PCI bus used by the RC2000 and UltraSonic platforms can provide a peak of about 60 Million 32 bit random words per second. This is roughly the same speed as software random number generation using the Mersenne Twister [148] uniform generator, or twice the speed of a Ziggurat Gaussian generator on a 2.66GHz Athlon. In spite of the low relative speed-up due to the PCI bus bottleneck, it is still worth using hardware to generate random numbers for software applications, as the cycles that the processor does not spend generating random numbers can be used to process data.

Numerical simulations usually generate random numbers by linking to a library that provides good quality random numbers, such as the the Boost C++ template library [59], or GSL (The GNU Scientific Library) [77]. The Boost library is a template library, and so is compiled directly into the application it is used from, making it difficult to integrate hardware functionality without recompiling every application, but GSL can be built as a shared library. This allows IGOL support for random number generation to be inserted without recompiling applications.

The main problem found when accelerating applications via GSL was that the main GSL functions for generating random numbers only return one random number on each call. The bandwidth for single word transfers from hardware to software is very low due to the setup costs for each transfer, regardless of the amount of data that is actually transferred [41], so demand based transfers from hardware result in a very low random number generation rate. Fortunately random number generation is a process with no input data, so it is possible to generate random numbers before they are requested by the software application. The GSL interface to IGOL takes advantage of this by prefetching random numbers in large batches, then storing them in a large memory based FIFO. As random numbers are requested through GSL a single word is returned from the FIFO, making the cost of each random number generation that of a single memory access and pointer increment. The FIFO is refilled using a separate thread that is automatically created when the GSL random number generator instance is created, which runs in parallel with the application's main thread.

Using the hardware uniform random number generator scheme presented in Chapter 6 on

an RC2000 hardware platform in a 2.66GHz Athlon results in a generation rate of 48 MSamples/sec, compared with 31 MSample/s when the GSL built-in Mersenne Twister is used. Although this is only a 1.55 times speedup in generation rate, the host CPU utilisation when generating random numbers in hardware is only 10%. When the random numbers are actually used in a calculation this results in a larger speed-up, as not only is the generation rate higher, but the amount of CPU time dedicated to random number generation is lower; in a monte-carlo integration to calculate Pi [181] it was found that the application ran 2.3 times faster using the hardware accelerated GSL compared to the standard software GSL.

Video Processing

Video processing is a popular target for FPGA based hardware accelerators, due to the large amounts of pixel-level parallelism and pipelining possible and the relatively narrow fixed-point arithmetic that is required [62, 88, 143, 217]. Hardware accelerated video processors provide components that are needed as building blocks in applications, such as optical flow, as well as video filters that are directly useful, such as special effects that can be selected and applied to a video stream by the user of a video editing application such as Premiere. The IGOL framework allows hardware accelerators to be used in a wide variety of applications, by providing a single plug-in for each that allows all IGOL video processing to be used from within that application. A brief summary of the applications that have been adapted in this way follows:

DirectShow DirectShow [160] is the main streaming media framework for the Windows platform, and is used behind the scenes to provide support for video conferencing applications, DVD players and VOIP (Voice Over IP) applications in Windows. The main abstraction in DirectShow is that of a filter, which is a COM component with one or more pins. These pins represent points where data can either enter or leave the filter, and by connecting filters together by their pins a data processing graph can be assembled. This has a very natural mapping to the port based model of IGOL Operations, and provides a high performance video processing environment, explored further in Chapter 5. Figure 4.4 shows an example of an IGOL Operation being used from within the DirectShow environment.

IGOL Operations are also exposed through the DMO (Direct-X Media-Object) API [160], a simplified version of the DirectShow interface. This removes much of the complexity of implementing a DirectShow filter, but decreases the control that the filter has within the graph of filters; a DMO filter is not required to manage the movement of data from input to output (unlike a DirectShow filter), but cannot employ advanced techniques such as using multiple threads to parallelise processing and IO tasks. The DMO interface is exposed to allow applications that support DMO filters but not full DirectShow filters, such as the Buzz modular audio synthesizer [211]. When applications support both DirectShow and DMO interfaces, IGOL is exposed through DirectShow in to achieve the maximum performance.

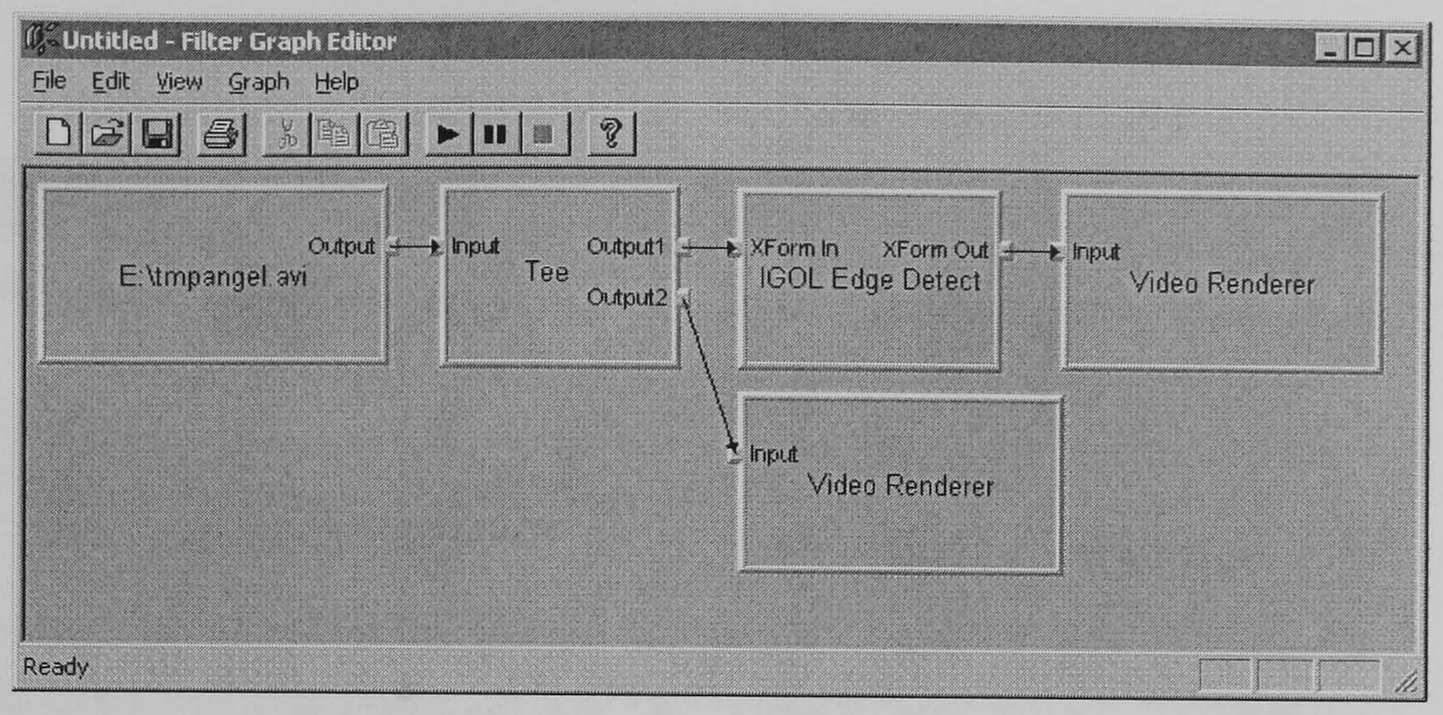


Figure 4.4: Example DirectShow filter graph. On the left is a source filter, which is reading frames from a file. This feeds into a Tee filter, which just sends duplicate frames to both its output pins. One of these pins is connected to a video renderer, allowing us to see the original video stream, while the other is passed through an IGOL edge detection filter, and then onto another video renderer to display the processed video. When this graph is played the original and processed videos will be played side by side.

Direct-X Image Transform The Direct-X Image Transform architecture [127] is an image processing API used by Microsoft to allow image processing and special effects to be used within applications such as Internet Explorer. It is implemented using COM interfaces, but with a very different communications pattern to that of DirectShow, with an emphasis on applying time-varying effects to static images, rather than to video (although it supports both). The interfaces are also designed to be used from scripting languages such as JScript, allowing the use of script-driven special effects in web pages. Figure 4.5 shows an image being processed by hardware in Internet Explorer, controlled by JScript. Figure 4.6 shows a live web-page being processed using an IGOL filter.

Premiere Adobe Premiere is a video editing application produced by Adobe, and is commonly used in the media-creation industry. Premiere has a proprietary plug-in API [34] that is used to integrate special effects that can be applied during editing, and many third-parties create and sell plug-ins. Premiere Plug-ins are installed simply by placing the plug-in DLL in a specific directory, which Premiere scans at start-up, and the integrates any plug-ins it finds into its GUI for selection by users. IGOL operations behave like any other plug-in from within the GUI, with full support for interactive adjustment of Operation Instance parameters through the plug-in API.

VirtualDub VirtualDub is an open source video processing application, often used to batch edit large amounts of video, while applying filters such as noise reduction and frame resizing. VirtualDub has a proprietary plug-in API [122] which allows a similar level of

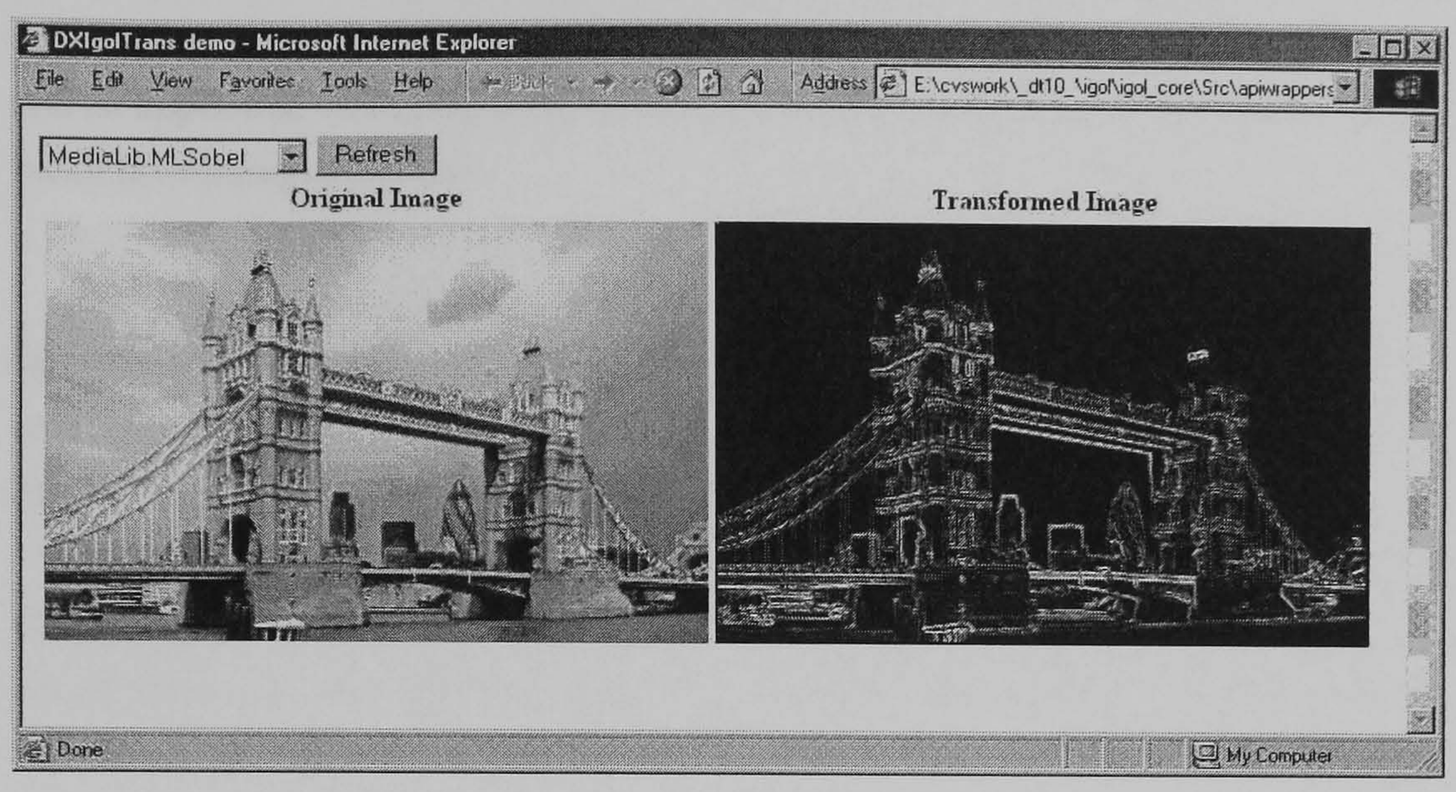


Figure 4.5: Image processing in a web page using the Direct-X Transform plug-in

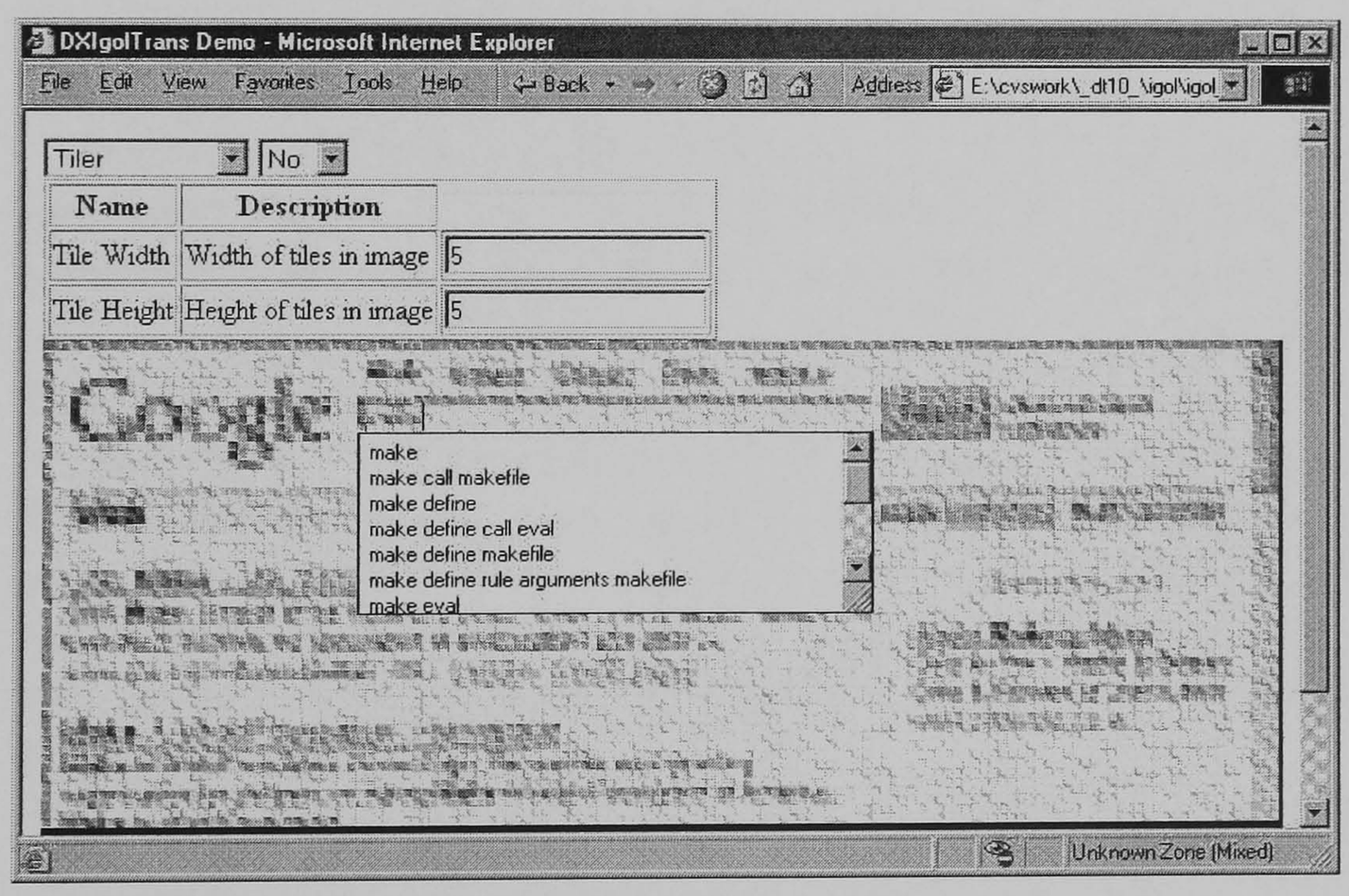


Figure 4.6: Applying image processing to a live web page using the Direct-X Transform plug-in. The contained web page is fully functional even though it has been distorted, and can be navigated to other pages.

functionality to that of the Premiere API, but uses a different method for the interactions between application and plug-in. As in Premiere, plug-ins are installed by placing a DLL in a specific directory, and will then be accessible from within the application. IGOL operations can be applied to video, have full parameter support, and can also be scripted by the built-in batch processing language.

Any image processing Operation Specification known by the IGOL Registry is available through all four of these applications and APIs, and the optimal hardware or software based accelerator will be selected in each. The performance of DirectShow based acceleration is explored in Chapter 5.

An issue that was discovered when integrating with the plug-in based application such as Premiere and VirtualDub is that the applications expect each plug-in DLL to expose a fixed set of video processing filters. However in IGOL the number of Operation Specifications that could be instantiated may change whenever meta-data is added or removed from the registry. If a single plug-in DLL is used to expose all Operation Specifications that IGOL can provide, then this must be mapped to the application's plug-in architecture by exposing a single "IGOL Operation" plug-in, which appears as a single choice to the user within the application. In order to select a specific Operation Specification the user must first add an instance of the generic "IGOL Operation" plug-in, then use the plug-in parameter user-interface provided by the application to select which Operation Specification the plug-in will instantiate. This two level process does not follow the standard model for plug-in integration into the Premiere and VirtualDub GUIs, and is confusing to the user.

In order to allow IGOL Operation Specifications to appear as first-class filters in Premiere and VirtualDub a system of small forwarding DLLs was used. Each forwarding DLL is bound to a single Operation Specification by embedding the associated GUID into the DLL, and then placing it in the application's plug-in directory. The application sees each forwarding DLL as a plug-in that exports a single video filter, and displays them in its GUI as separate filters. When the application attempts to instantiate the plug-in, the forwarding DLL simply passes all requests onto the main DLL that implements the plug-in to IGOL mapping, using its embedded Operation Specification GUID to tell the main DLL which implementation to select. This introduces no performance penalty over having a single DLL, and the only disadvantage is a small disk-space penalty for the forwarding DLLs in the application's plug-in directory.

It is necessary to create these forwarding DLLs as and when new Operation Specifications are added to the IGOL Registry, and this is achieved by attaching notification hooks to the Registry. An application plug-in such as the Premiere to IGOL mapper informs the Registry that it needs to be notified when meta-data is added or removed, for example when new components are installed. The hooks are installed by creating a COM component with the correct interface and then informing the Registry of the GUID used to create it. If a hook is installed then after the Registry has added new information it will instantiate the hook ob-

ject, and tell it which Operation Specifications or Implementations have been added. The hook object can then perform tasks such as creating and installing forwarding DLLs.

Audio Processing

Audio processing can be a computationally expensive task, particularly when many concurrent floating-point audio streams must be processed in real-time, but so far little audio processing work using FPGAs has been presented, except for audio codecs for embedded devices. Audio processing is supported within the framework, but no examples of accelerated audio processing have been developed. However, support for running hardware audio filters from within real applications is useful as a debugging tool, for example when audio components for SoC (System on a Chip) designs are being developed.

IGOL support has been provided in three audio processing applications: WinAmp, Premiere and DirectShow. The DirectShow support is provided by the same plug-in that allows processing of video, as it supports processing of many different data types. The Adobe Premiere support is provided by an audio-specific plug-in, independent of the video plug-in, as the two types of media use a different interface. As with the video plug-in, IGOL Operations appear as individual filters within the GUI, and parameter ports are fully mapped to GUI parameters within the Premiere. WinAmp is a popular MP3 player which supports a plug-in architecture to allow processing of audio data [168], so that effects such as stereo separation or bass boost can be applied before the sound reaches the speakers. An IGOL plug-in is provided which allows audio filter operations such as chirp and adding noise to be applied to the audio.

No speed-up was achieved in any of these applications, as all the audio operations implemented in hardware already achieved greater than real-time performance in software. Even when audio data is batch processed, the bottleneck was found to occur in the ability of the applications to source and sink the audio data, rather than in the ability of the CPU to process the data.

UNIX Console Processor

Am application written for use as part of the IGOL SDK is a command line application that uses the UNIX console model to process data. This application reads data from its input stream, feeds it through an Operation Instance, then writes it to its output stream. The Operation Specification to use is specified on the application's command line, and can be any Operation Specification that has one input port and one output port, or a single output port. A number of data formats are supported for the input and output streams, such as video and audio file formats, but it is also possible to simply stream the data through the operation as raw binary, without any translation.

The binary mode can be used to test hardware designs when the data format to be processed or produced does not correspond to any existing data format. A simple com-

mand line application can be written to generate appropriate test data, and another written to test the output from hardware. This provides a useful test bench for the hardware design without writing any software to transfer data to and from the hardware platform, or needing a file format to hold test data.

4.2.2 Characteristics of Successfully Accelerated Applications

While almost any application or API can be interfaced to the IGOL framework, not all will benefit to the same degree. Some applications may achieve much greater speed-ups than others when hardware is used, and some may even be deccelarated. The level of integration of IGOL into the application also varies, with some supporting IGOL as a highly integrated feature, no different to any other plug-in, while in others the presence of IGOL will be obvious, as it appears or behaves differently to other plug-ins using the API. Here some of the common characteristics of applications that have been successfully accelerated using IGOL are summarised.

The first observation is that the more information that the application makes available to the plug-ins it instantiates and uses, the more optimisation IGOL can perform on its behalf. This is best demonstrated in the DirectShow architecture, which allows plug-ins to discover significant amounts of information about overall data-flow within the application. This can be used by the IGOL architecture to perform optimisations such as sharing hardware between multiple plug-ins, and routing data flowing between plug-ins via hardware channels (this approach is evaluated in more detail in Chapter 5). However, most applications do not make such information available, as the main intent is to isolate the plug-in from as many unnecessary side-effects as possible.

Another important characteristic is that in order to achieve the maximum speed-up in hardware the application or API should allow large batches of data to be processed at once. This is important as the overhead for each data-transfer to or from hardware is high relative to that of a function call in software. APIs and applications that ask plug-ins to perform many small tasks, or to operate on many small items of data, are likely to be impossible to accelerate using hardware, as the time spent communicating with hardware is longer than that of the processing time of a processor based implementation.

In some cases the effect of these very "chatty" APIs can be hidden through software caching techniques, such as the system described above for the GSL random number plug-in. However, in many cases APIs use a synchronous call mechanism, meaning that input data is passed into a function call, and the expected return is the output data corresponding to that input. With a synchronous call mechanism it is impossible to hide the latency of the hardware communication calls, as these must happen during the API function call. For example, it was found to be impossible to accelerate the PhotoShop application under some conditions, as the application requested the synchronous processing of very small blocks. The time required to process these blocks in software was less than the time taken to transfer

to hardware, so in these cases the processing was automatically delegated to a software implementation. Only blocks large enough to allow the transfer overhead to be outweighed by the faster processing speed are processed using hardware.

APIs that use synchronous calls to process small amounts of data can be accelerated, but only if the API provides support for pipelining of data packets. This means that the application or API must be able to provide input packets without immediately requiring the corresponding output packet. Once the output data is ready the application can collect it and must be able to tell which input packet it is derived from. The DMO (Direct-X Media Object) filter interface is the only synchronous API supported by IGOL which can operate in this way, allowing multiple input packets to be presented to a filter, then using separate calls to retrieve the results. Pipelining is also possible in the DirectShow API, but in this case the IGOL plug-in directly manages data-flow through threads under its control, rather than through synchronous function calls on behalf of an application thread.

4.2.3 Summary of Plug-Ins

The IGOL framework is supported by a wide selection of plug-ins, which allows hardware and software components to be accessed from many different applications. Due to the extensible nature of the framework many different types of data processing such as Video effects, audio filtering, and random number generation are successfully supported. Hardware accelerated components are automatically presented through all compatible plug-ins, so for example a hardware accelerated video processor can automatically be used through Premiere, DirectShow, PhotoShop, VirtualDub and the DirectX Transform and DMO APIs. No changes need to be made to hardware components in order to support each application, and applications automatically support all available hardware devices.

4.3 Device Support

Reconfigurable hardware platforms must be accessed from host computers through a device specific API (Application Programming Interface), which is different for each type of platform. Due to the different capabilities of hardware platforms, these platform specific APIs vary significantly in the number and types of operations that the software can perform, but there are common tasks that must be performed: different platforms may have single or multiple FPGAs, but the tasks of configuring the FPGA remains the same independent of the number present.

In IGOL a device independent API called HAPI (Hardware API) is used to provide a uniform interface to all hardware platforms, allowing common operations such as configuration and transfers to memory banks to be performed in a device independent way. However, HAPI does not attempt to hide the differences between hardware platforms, and allows any configuration of FPGAs, memory banks, and other platform resources to be exposed

to applications. Applications must still understand the configuration of each hardware platform in order to use it successfully, but can discover and use all platforms through a single uniform interface.

The advantages of HAPI are:

- Applications and components that use one type of hardware can be ported to different types of hardware by re-using the code that performs common tasks, and only adding code where differences between platforms must be addressed. Two of the most important tasks, FPGA configuration and transfers to and from memory banks, can be performed through HAPI using identical code for many devices.

- Hardware is accessed through a uniform API that provides a single consistent interface for discovering and using hardware. Where device specific APIs provide multiple methods for performing a task with different performance characteristics, such as different methods for transferring data to memory banks, the application can use just a single method. Behind the scenes HAPI will choose the device specific call that provides the best performance.

- Simulated devices can be used that allow the performance of different hardware platforms, even those that do not yet exist, to be evaluated. Devices can also connect to software simulations of hardware logic, allowing testing and debugging of the interaction between hardware and software components without requiring the presence of actual hardware.

4.3.1 Unified Interface

Between any processor and a peripheral there are always (at least) two interface levels. The first is the obvious physical interface that enables the movement of data between the two, such as PCI [172] or USB [219]. The second interface level is the software API that allows programs running on the processor to "talk" to the peripheral in some way. At the API level the functionality of the peripheral is exposed, meaning that each class of peripheral needs a specialised interface: graphics cards, hard disks and sound cards can all be connected via the same physical interface, but the API presented to software is very different in each case. In some cases the API becomes a very abstract view of the peripheral, tightly integrated with the operating system and bearing little resemblance to the underlying device, for example the directory and folder metaphor used for storage in most modern systems. Software can access almost any type of storage device using this single API without needing to know how or where the storage is. In other cases the API is device specific, with little abstraction beyond mapping data transfers with the device to software function calls.

There are at the moment no standards at all for FPGA peripheral APIs; plug-in cards containing FPGAs have been around for almost as long as the FPGA concept itself, but no standards have evolved. One argument is that every device has different features and

capabilities, so it would be difficult to create one API that covers all of them, unless just a small subset of common features were chosen. However, this assumes that the API will try to be a high-level interface, attempting to hide most of the device details away behind a single abstract device model. If instead the API provides a low-level abstraction, then each device's specific features can be exposed. This means that applications still need to be able to recognise each type of device, but common features between devices can be accessed in the same way. It also makes each device accessible through a single point of access, removing the need to deal with the details of each device specific API's conventions for linking, error handling and parameter ordering.

The aims for creating a unified API for accessing FPGA devices from software are:

Single Point of Access All devices are available through the same mechanism: applications only need to link to and support one API.

Single Error Model All devices expose exceptions through the same mechanism.

Complete Feature Coverage The unified API should expose all platform features that are available if the native device API is used.

Single Device Abstraction All devices expose their features in the same way. Each device instance will have a different feature set, but all those features are accessed in the same way.

Low Overhead The added cost of going through the unified API compared with using the native device API directly should be as low as possible.

Even though a low-level interface has the drawback that each device still needs to be explicitly supported by an application, there is now the possibility to share code when two devices support common features. For example a task common to most FPGA devices is configuring the FPGA from either a file or a memory image of the configuration bitstream. The task is functionally equivalent across most native device APIs, but an application must re-implement the configuration task for each device, as each native API will have a different calling convention. Using a unified API such common tasks can be implemented once, then reused on all devices that support such a feature. Another example is devices which allow direct transfers to and from memory banks: the task is always the same, take a memory buffer on the host and transfer that data to or from the device's memory banks, but the exact details of performing this task vary significantly across APIs.

Another advantage for application developers of using a unified API is that it can remove extraneous or confusing options exposed by the native API. It is often the case that the native API will offer multiple ways of performing a task, with sometimes unclear performance advantages and penalties. A good example is memory transfers, where there is often one API call for performing small reads and writes, and another for transferring larger blocks, with different performance characteristics depending on the data transfer size.

By hiding the multiple ways of performing a task, behind a single method in a unified API, the complexity can be removed from the application and hidden behind the HAPI device interface. Developers gain a much simpler interface, while still taking advantage of the performance provided by the native API behind the scenes. For the memory access task the unified API will consider each call separately, and map it to the most appropriate native task based on platform specific information and run-time statistics.

A single API designed to support multiple native device APIs also provides a convenient way to create device simulations or emulations. One kind of device that can be provided is to map calls to the hardware device to a software simulation, rather than real hardware. This offers the ability to test applications against devices without actually needing a real instance of the device, ideal for debugging communications when the hardware is not yet ready. It also allows fictitious devices to be used, so that benchmarks for potential hardware platforms can be executed and evaluated.

Design

One goal in creating a unified API is to support any and all features that a hardware device might have. This presents a problem, as it is impossible to examine the feature set of every native device API that currently exists, let alone those that do not yet exist. However, the features of some common FPGA platform device APIs can be broadly classified into the following features [4, 11, 22, 51]:

Configuration The overall setup of the device including FPGA configuration, clock setup and internal data-path routing. While most applications only do this task once at startup, others alter device configuration throughout their run-time.

Communication Communicating with the code that is executing on the FPGA, including passing of parameters and synchronisation. The exact mechanism may be through dedicated synchronous channels built into the device, or through more ad-hoc asynchronous channels operating over general-purpose IO provided by the device API.

Data Transfer Transfer of segments of data from the host to the device or vice versa. The target on the device will often be a memory bank on the device, using a dedicated transfer channel independent of the FPGA, but might also transfer data to and from the FPGA directly. Data transfers involving memory banks also often require locks to be held by the host to prevent concurrent access by host and FPGA.

Notification Some devices offer a way for the FPGA to notify the host that an event has occurred, by triggering an asynchronous callback to the host CPU. Notification is distinct from Communication, from the hosts point of view, as it not directly controlled or initiated by the host. Notifications could potentially happen at any point during execution, rather than only when the application makes a call to the device.

If these tasks are simplified further, then all four correspond to a form of data transfer. Configuration, Communication and Data Transfer are host-initiated transfers, while Notification is initiated by the device. Any task that a device API can expose must ultimately be reducible to an underlying explicit data transfer, otherwise the task could not operate over the bus between CPU and hardware platform. The concept of reading and writing data is used as the central abstraction of HAPI, with the same pair of functions used to perform almost all tasks.

Each device in HAPI is presented to applications as a pseudo memory-mapped device, with a 64 bit address space within which applications can perform data transfers to and from the device. Within this address space all the features of the device such as FPGA configuration and memory banks are placed at different addresses, with each feature accessible by reading or writing data at that address. This model could be considered just a high-level abstraction of the underlying physical interface, where devices appear at different IO addresses on the data-bus, but in reality the two are not really related; all calls are still routed via the device specific APIs, so underneath the device driver will be doing all the raw data transfers.

Although each device feature is mapped into the address space, not all platform features can operate in the same way. Some data transfers can only happen in one direction, while others may block if the device is not ready. To support these differences the address space is split into distinct regions with different characteristics. Usually each device feature would sit in its own address region, but where appropriate more than one feature could be placed in a single region, or a single feature might be split across two or more regions with different characteristics. How exactly device features a spread across the device's memory space and which regions they are placed in is a choice made by the creator of the HAPI to native API translation layer for each device.

Each address region's semantics is determined by five characteristics:

Readable An address region with read access supports host initiated data transfers from the device back to the host.

Writeable Indicates that the host can initiate data transfers to the region. Regions may be writeable without being readable, for example a device to host communications channel.

Requires Lock Address regions that have locking semantics need to be locked before the host before data transfers can be performed.

Can Block Data transfers to or from this region may block if the device is not ready, for example if the address represents a synchronous communications channel. Regions which do not have this property cannot block, i.e. data transfers either start immediately or fail.

Can Notify The address region may be the source for notifications. Any region can potentially be a notification source, and regions may act as both a normal read/write region as well as supporting notification.

This five characteristics are the only properties of regions that can affect the semantics of the region from the point of view of the API: more complex behaviour such as regions which are sometimes readable must be exposed as a region with the Readable property, with the API returning errors in situations when the device is not in a state where the region can be read.

In principle these characteristics could also be incorporated into the memory mapped metaphor, by placing status flags at known points in the address space. This would allow absolutely every HAPI device task devices to be performed with just two functions for read and write. However, this would make applications that use HAPI more complex, and would slow down the interface as multiple calls into HAPI would be required rather than just one.

Regions which have locking semantics require that the host acquires a lock on the region before starting any data-transfers. This is necessary to prevent concurrent access to a device feature by the host and FPGA. Locking and unlocking regions is important, but requires some work on the part of the application. To allow simpler use of lockable regions the API allows three different modes of locking operation to be set on the region:

Manual Lock The application is responsible for acquiring and releasing the region's lock before accessing it. If a data-transfer is initiated to the region when the lock is not held, the data-transfer will fail.

Auto Lock Any time the application starts a data-transfer to the region, the lock status will be checked. If the region is not currently locked then the lock will automatically be acquired, then released after the data transfer has completed. If the region was already locked then it will be left locked after the transfer completes.

Sticky Lock As with Auto Lock, the region will be automatically locked before a data-transfer (if not already locked), but the region will remain locked after the transfer completes. So no matter the lock status before the data-transfer, the region will always be locked after a data-transfer to the region.

These three modes allow the application developer to trade-off simplicity of access against performance. The Auto Lock is the simplest method to use, but has the disadvantage that if multiple transfers to different addresses within a region are needed, the lock will be acquired and released multiple times. Lock operations often have a significant overhead, requiring a round-trip to the device over the physical bus, so if many small transfers are needed the cost of the lock operations may become significant. The Sticky Lock provides the convenience of not having to explicitly lock the region, while removing the overhead of repeatedly acquiring and releasing the lock. However, the application must remember to explicitly release the lock at some point, otherwise the FPGA will never gain access. Manual

locking offers the most control, but also requires the most work. As well as offering potentially the best performance, it also allows applications to use communications protocols whereby the acquisition and release of the lock are used as a communication mechanism as well as providing mutual exclusion.

Notification regions provide a way for devices to initiate communication rather than the host. For example devices with interrupt support will map the interrupt to a notification region on the abstract device. When a notification occurs the application receives a callback containing information about where in the device address space the callback initiated, and any optional data associated with the notification, for example an interrupt number. An important point is that applications will only receive notifications for a given region if they explicitly request them. This is necessary both due to the practical considerations of how to initiate a callback into an application's code without being passed an entry point, and because applications must factor the re-entrancy introduced by notifications into the code.

Notifications can happen at any time: there is no attempt to delay them until any API call in progress has finished. Applications that are interested in notifications must be explicitly thread-aware to accepts notifications that may happen while a data-transfer is in progress, or while the application is doing something non-hardware related. If a notification occurs and the application has registered its interest then the callback occurs immediately on an independent thread: if no callback has been registered the notification is simply dropped. It is up to the application to organise itself so that when the callback is executed either all the work needed to respond is done immediately (if the amount of work to be done is small), or the notification is communicated back to the main thread or a worker thread.

The device API is implemented using COM, as this provides a high performance interface, and makes the hardware devices accessible from many different languages [23].

Summary

The HAPI abstract hardware API makes all devices accessible through the same interface, even though the feature set exposed by each device will be different. This makes it easier for programmers to port applications between devices, as all the devices share the same API semantics and use the same method calls. A memory mapped metaphor is used to provide access to features within each device, requiring just two types of operation to access all functional, reads and writes.

4.3.2 Real Devices

A brief overview of the devices currently supported by HAPI is now presented, and an explanation of how their feature sets are exposed using the HAPI memory map metaphor,

RC1000

The RC1000 is a relatively old hardware platform [22], with a feature set and device API designed to be easy to use. The main features of the platform are:

- Single FPGA
- Four 2MB memory banks with locking
- Two synchronous uni-direction communications ports between host and FPGA
- Two clocks with configurable clock rate

Memory Banks The RC1000 memory banks must be locked before use, and each bank has its own lock status, so the host and FPGA can each own different banks concurrently. The banks also form a single contiguous addressable space within the RC1000, so a single transfer with HAPI must be able to write to all the banks. To allow this the four banks are exposed as four adjacent regions within the device, and each region has its own locking status. Writes that overlap more than one memory region will still perform per-region memory locking according to the HAPI model, but the actual data transfer using the native API will be performed in one call.

The native API offers two ways of accessing the memory banks, DMA and memory mapping. The DMA access method provides high throughput transfers, at the cost of a startup penalty for each transfer, while the memory mapped interface allows low overhead random accesses to RAM banks, but with lower throughput than DMA. When using the native API the developer has to choose between the two methods to get the best speed, but when accessed through HAPI, reads and writes will automatically be mapped to the most efficient method. In the HAPI interface a break-point for data transfer size is checked, and all transfers smaller than this size using are performed using memory mapping, and all larger transfers using DMA. The value of the break-point depends on the revision of the RC1000, the driver version, and the motherboard of the computer, so the HAPI device wrapper determines the break-point by automatically performing IO benchmarks when it detects that the system configuration has changed. This provides a measurable improvement of around 5% in bandwidth for data transfers near the size of the break-point when the RC1000 is used in different machines.

Synchronous Channels The two synchronous channels act as an atomic message passing service between the host and the FPGA: the Control channel is used by the host to pass data to the FPGA, while the Status channel is used to pass data from the FPGA to the host. Either party may initiate a transaction on a channel at any time, but will be blocked until the other party also starts a transaction on the same channel. Once both parties have started a communication transaction the message is atomically passed, and the API calls on both sides return.

On the host side the Status channel could be implemented as a notification, but this does not follow the semantics of the original API, as by design the host application must actively read the channel. On the FPGA side the developer expects that when a write to the Status port completes that it is because the software made an explicit read transaction. If the Status port were mapped to a Notification then the software application could choose not to register for that notification, so the channel semantics would be broken. For this reason the Status port is implemented as a Readable region with the CanBlock flag set. Similarly the Control port is implemented as a Writable region with CanBlock set.

Configuration Configuration of the FPGA is exposed as a memory region that represents the static configuration of the device. Writing to this device configures the FPGA, automatically selecting the fastest configuration mode depending on the capabilities of the card. Using the native API configuration is a write-only operation: once the FPGA has been configured the information about what was used to configure the device is discarded, but with HAPI the static configuration of the device can also be read back at a later point, by reading from the same memory region.

RC2000

The RC2000 [4] is a minimalist device by design and provides a very low-level native API for the platform. The main features of the platform are:

- Single FPGA
- Six banks of 8MB, accessible only by the FPGA
- Memory mapped communication between host and FPGA
- Asynchronous interrupt from FPGA to host

The design of the RC2000 API is very different to the RC1000. Where the RC1000 limits the architect to just a few communications patterns, sacrificing some potential performance in favour of simplicity, the RC2000 takes the opposite view. There is only one very low-level communications channel, and it is up to the engineer to implement their application specific protocol on top of this. This offers the application architect great flexibility and the highest potential performance, but basic communications primitives must implemented by the engineer.

Memory Mapped IO The main method of communication between host and FPGA is over a memory mapped bus (distinct from the memory mapped API metaphor provided by HAPI). From the host's point of view this is simply an addressable region which can be read and written to: what each address corresponds to is determined by the configuration loaded into the FPGA at that time. The FPGA is responsible for examining each data transaction

arriving over its local bus, deciding what the target address represents, and then performing the corresponding action on that target. This makes it simple to expose using HAPI : a single address region with the same size as the FPGA address bus is provided within the HAPI address space, allowing direct reads and writes to the FPGA bus. The FPGA has a lot of control over this memory mapped region, so very little can be assumed about it. The flags CanBlock, Readable and Writable are applied to the entire region, as any of these might be true depending on the configuration. Data transfers to the FPGA can be performed using either DMA or memory accesses, so the same technique used for RC1000 memory transfers is used to determine the best method to use at run-time.

Interrupts The RC2000 interrupt is a very simple notification mechanism which provides no information other than that an interrupt has occurred. This is mapped to a notification address region on the device wrapper. When the interrupt is fired it passes a notification event to the application using a zero length data payload; if more information is needed by the application to process the interrupt, it must initiate normal data transfers to retrieve any associated interrupt payload data.

Within the device wrapper the interrupt is implemented by using a worker thread to wait for notifications provided by the native API. Once an interrupt occurs this thread is used to make the asynchronous callback into the application. Between interrupts the thread blocks on an operating system mutex, so CPU resources are only consumed when an interrupt occurs.

UltraSonic

The UltraSonic [51] has the most complex architecture of the three platforms considered here. Rather than a single processing element it can have from one to sixteen FPGAs installed using daughter-cards called Pipes. Figure 4.7 shows the overall architecture. The main UltraSonic board is a PCI backplane which provides a configurable set of data-paths to control and connect the Pipes together. The Bus controller uses the shared pipe bus to read and write to individual pipes, allowing the host to configure each pipes and the route for data entering and leaving the UltraSonic.

The Pipeflow bus is a unidirectional bus that connects all the pipes together in a circular chain. The backplane routes this path so that no matter how many pipes are added the chain is unbroken. This provides a fast internal bus for pipelined data transfers between adjacent pipes. There is also a shared bus which allows point-to-point or multi-cast data transfers between pipes, although only one pipe can access it at one time.

On each Pipe the FPGA is responsible for the routing and processing of these data flows. Figure 4.8 shows how the buses are connected internally. The low-level data-flow tasks are implemented by a hardware IP core called the Pipe Router. This handles the tasks of routing data within the Pipe according to the current configuration, manages access to

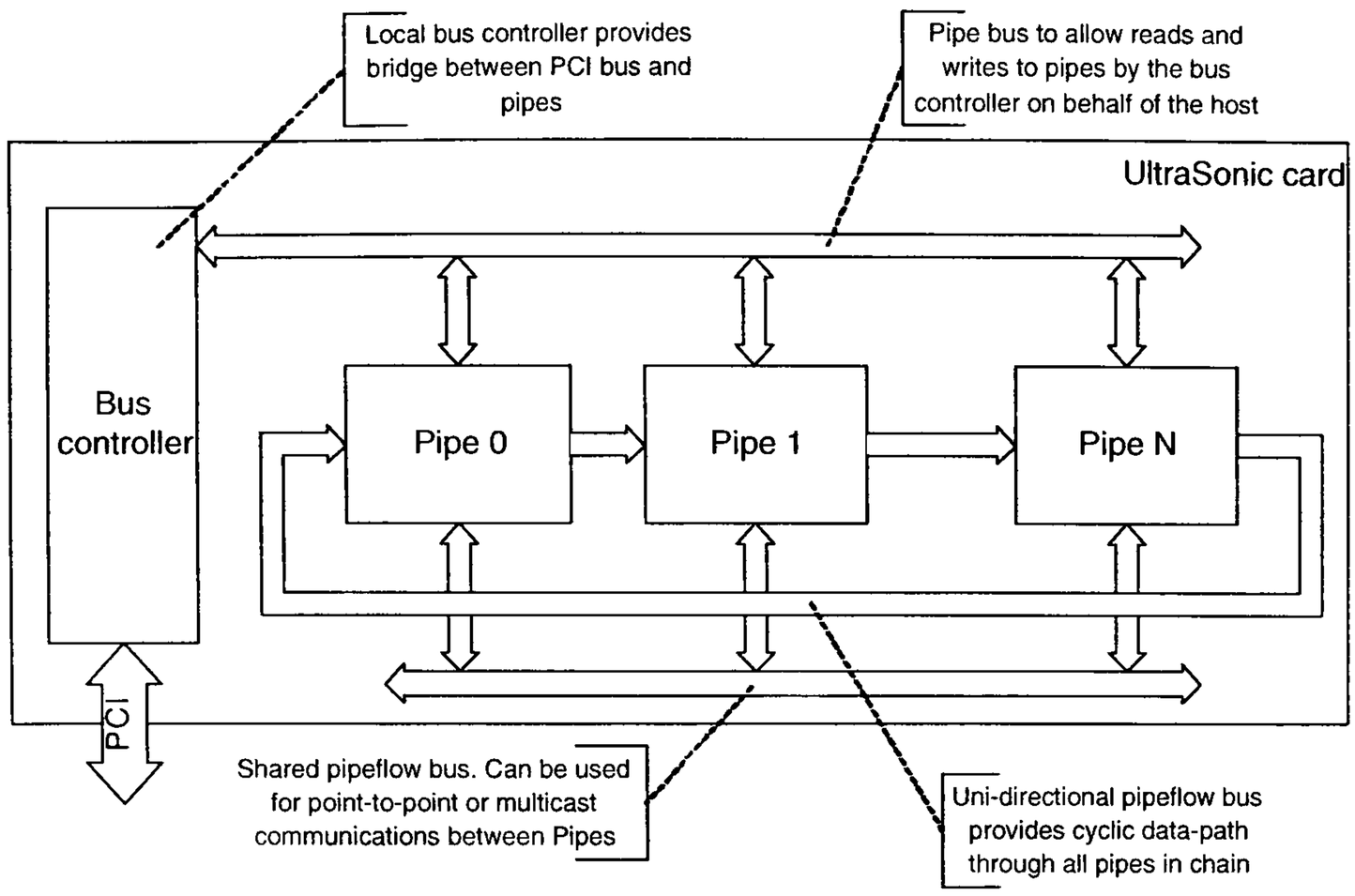


Figure 4.7: Data-flows within the UltraSonic backplane card.

the two memory banks on each pipe, and implements the pipe bus protocol. To create a configuration bitstream the Pipe Router library is linked with the user design containing the data processing logic. Once configured the Pipe Router can be controlled using registers to specify how data will flow within the Pipe. For example data coming into the user logic from pipe flow in can be sourced from the pipe to the left, the global pipe flow, or from the local memory bank.

Exposing the UltraSonic This complex organisation presents some challenges when adapting it to the HAPI model, as it is not obvious what HAPI should expose to applications. Depending on how many pipes are installed in the backplane, the capabilities of the card vary; it is not obvious whether an UltraSonic with one installed Pipe should be considered to be the same as an UltraSonic with two pipes.

The approach taking in HAPI is to split the use of HAPI into two usage models, based on how many Pipes an applications uses, and how the used Pipes are related in the backplane. This provides two ways in which the UltraSonic is exposed by HAPI:

Contiguous Pipes A device representing a fixed number of contiguous pipes. When an application requests the device the UltraSonic is scanned for a free range of pipes long enough to accommodate the requested length. The device is composed of a fixed number of address regions, each one of which represents a pipe in the allocated range.

All Pipes The device represents the entire UltraSonic, composed of 16 address regions,

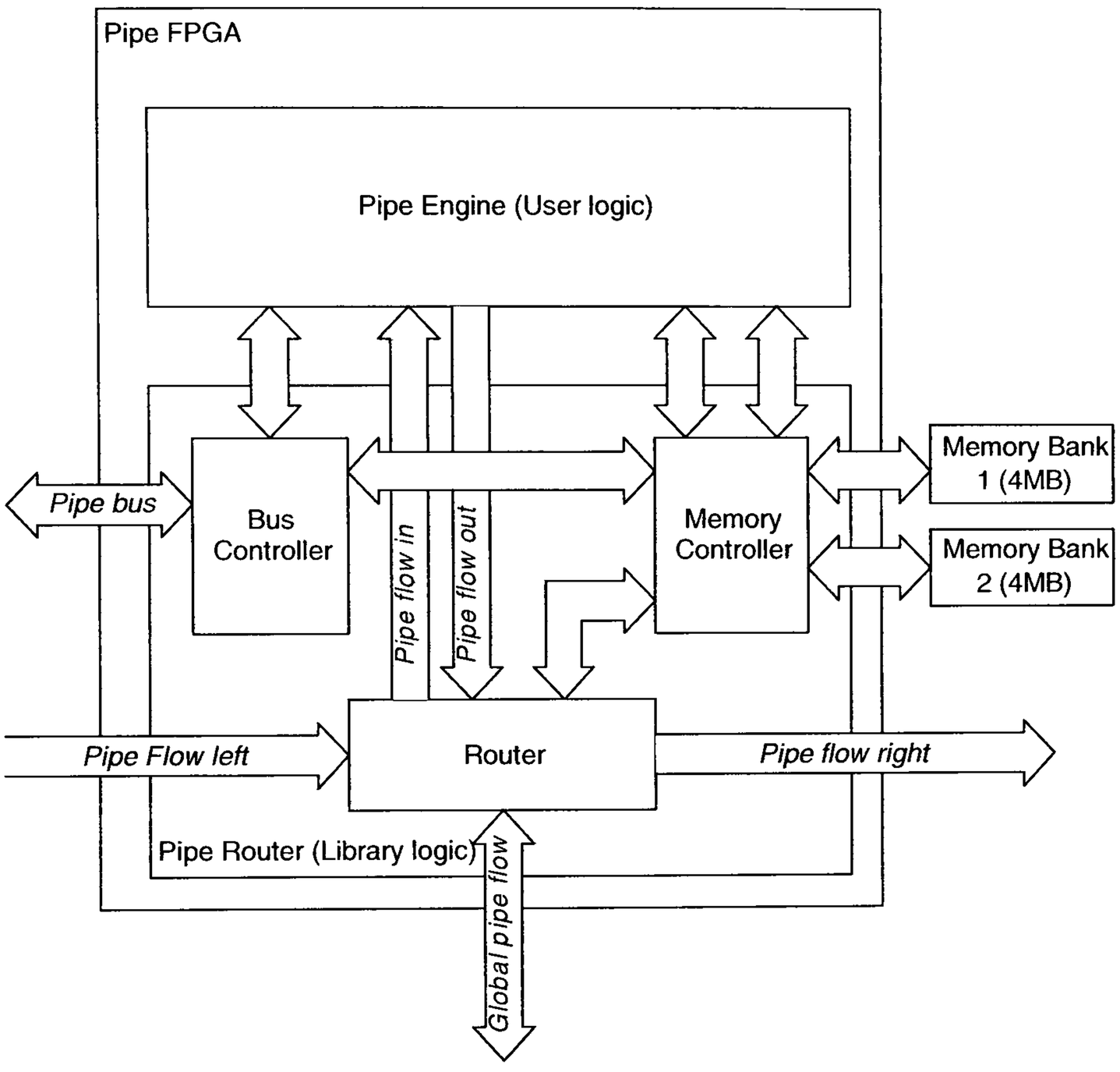


Figure 4.8: Data-flow within a single UltraSonic Pipe.

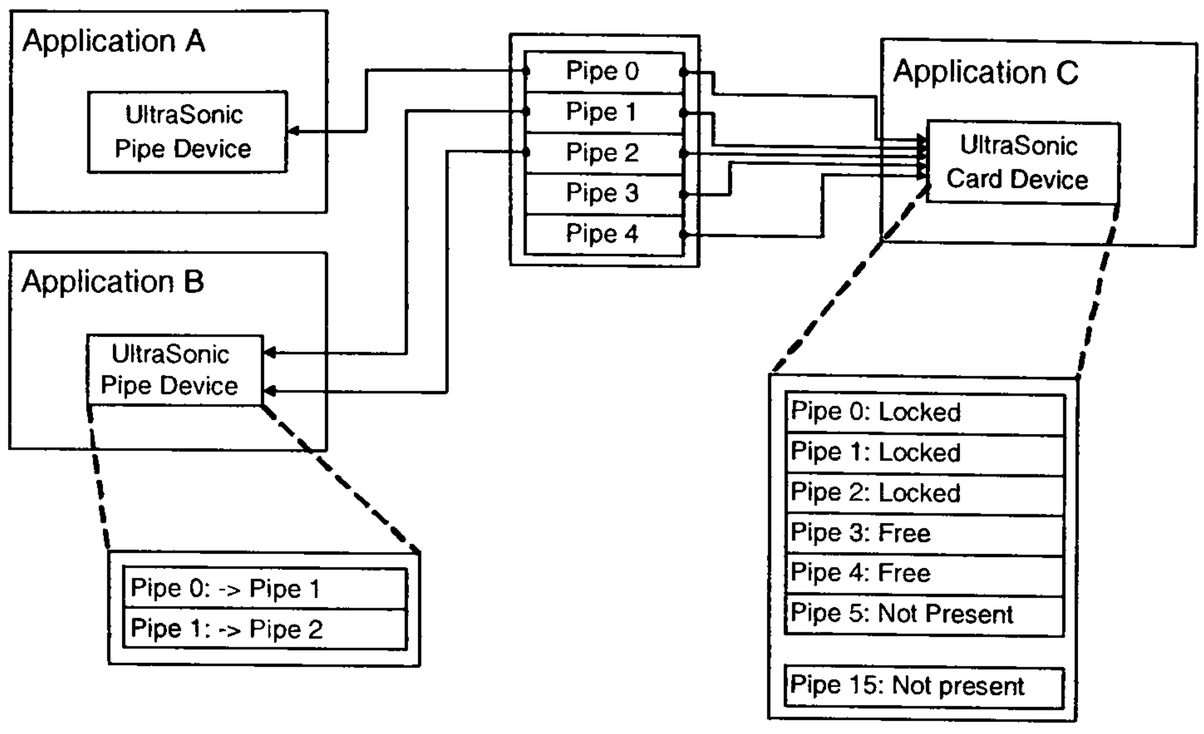


Figure 4.9: An example of multiple applications sharing access to a single UltraSonic device. Application A has requested a single pipe, and has been allocated physical pipe 0. Application B has requested two contiguous pipes, and has been allocated the next two pipes, although from its point of view the exact physical index of the pipes is not known. Application C has opened the device in Card mode, so it can see all the pipes installed in the UltraSonic. Three of the pipes are shown as locked, as applications A and B are using them, but two pipes are still available for it if it decides to use them. The card device that application C has opened also has address regions for pipes 5 to 15, even though there are no corresponding physical pipes present. It is up to the application to make sure it doesn't use pipes which are either not present or not currently owned.

as will as a master table containing information about which of the pipes is actually present, and whether they are currently available. It is up to the application to consider the set of available pipes and decide how to manage them.

Considering contiguous Pipes as a device in their own right provides a simple model to applications, while using the UltraSonic board in its entirety allows more complex configurations to be used, as long as applications are willing to micro-manage the board. Figure 4.9 illustrates these different operating modes.

4.3.3 Simulated Devices

An advantage of the HAPI interface is that there are no limitations on what sits behind the device interface: from the applications point of view it does not matter exactly how each read and write to the device memory map is implemented. This allows the device back-end to be implemented using software, rather than calls to a hardware platform through a device API.

HAPI includes support for software emulation of devices, which are configured by loading a DLL, rather than a bitstream. The DLL can contain a software emulation written using a language such as C++, which provides the same functionality as the hardware design, but

through sequential code. The software device can also be configured with a software simulation of the parallel hardware design, for example a bit-level cycle-accurate simulation compiled using Handel-C. In both cases the software device can contain a timing model, allowing estimates of the wall-clock processing time taken by an application to be made, even if the software emulation runs at a much lower speed than the real hardware.

By allowing hardware developers to use software devices during the development process we gain three advantages:

- More of the hardware development process can be completed without actually needing the real hardware to be present on the development machine. When hardware resources are scarce or have not yet been built it allows almost all of the integration testing to be performed using just software.

- Unit tests for the hardware can be written that exercise the whole design, rather than just those parts that do not perform IO. A common problem when testing hardware is that the code used during simulation tests is actually different to that used in the actual hardware, so tests that succeed in simulation may fail in the eventual hardware.

- Software devices can be mocked to up to explore the possibilities with different hardware platforms. "What if?" questions about the exact configuration of the platform such as changing the number of memory banks or increasing the number of FPGAs per card can be explored without needing to actually fabricate the hardware.

4.4 Summary

In this chapter we have presented the IGOL framework, and system that allows for the use of hardware and software data processing operations from within applications.

The core components of IGOL are: COM interface definitions, providing isolation between components with the system; the Registry, which acts as a centralised data store for operation meta-data and as a means for IGOL to make operations automatically available in applications; and the Activator, which allows applications to retrieve operation instances without needing to know how the are implemented.

We also demonstrated plug-in support for a wide range of applications, each of which is able to make use of any IGOL operation, be it implemented in hardware or software. This means that any of these applications can automatically take advantage of any hardware implementation available, without even being aware that IGOL or the FPGA device exist.

The abstract device API provides a uniform interface for FPGA devices within IGOL, removing the need to program each device using its own native API. This makes programming hardware a much more consistent experience, and allows some of the code that controls the device to be shared. Having an abstract API also allows us to implement emulated and

simulated devices, allowing the development and testing of software that talks to hardware devices, even if no physical device is present.

Chapter 5

IGOL Performance

5.1 Introduction

In this chapter we benchmark the performance of IGOL for video processing in DirectShow. As well as looking at the basic performance of the UltraSonic and RC1000 compared with software, we also examine the effects of two generic optimisations: segmented processing, which splits frames into segments, and overlaps data transfer and processing; and device sharing, which is a collection of techniques allowing multiple operations to share a device.

The key contributions in this chapter are:

- Segmented processing of frames, allowing pipelining of data transfer and processing to reduce total frame processing time. This can increase performance by up to 42%, and typically increases performance by at least 15% on the RC1000, without requiring any changes to the hardware configuration files.

- Placement of connected operations into adjacent pipes on the UltraSonic, allowing data to be streamed between them using a dedicated hardware data-path. This means that frame rate stays almost constant as more filters are added, providing a speedup over 9 times that of the equivalent software when 3 filters are used.

- Device sharing of the RC1000 by using reconfiguration to move filter operations through the device, while the data being processed remains on the card. This can provide a speedup of up to 4.5 times the speed of software when processing high definition video.

These optimisation techniques require no explicit support from either the application or the underlying component implementation, and are applied transparently by the framework. The optimisations can be safely applied to many hardware implementations without the knowledge or support of the hardware implementation, and the framework uses component meta-data to ensure that the optimisations are not applied to ineligible components. So

the optimisations are generic, in the sense that they do not depend on what a component implementation does, and can potentially provide a speedup for any application.

5.2 Test Setup

Here the performance of IGOL using DirectShow is examined. This is the highest performance environment currently supported by IGOL, and also allows the most optimisation by the IGOL framework. DirectShow is mainly used for the viewing of media in real-time, rather than as a batch processor, so here the ability of IGOL to apply real-time video filters is investigated.

The tests are based on the application of video filters to a video sequence and displaying the results onto the screen. The incoming video stream is read from a compressed file, decompressed, filtered using one or more IGOL filters, then displayed. The decompression stage adds a fixed overhead to the process, dependent mainly on the size of the frame.

The tests are performed using four video files with different frame sizes:

64x64 : Control This size video stream is unlikely to be seen in the real-world, but it is used here to highlight the relationship between per-frame and per-pixel costs. Except where it provides interesting results or demonstrates a point it will largely be ignored, as the numbers are often unrepeatable or unreliable. The video is compressed using HUFFYUV, a fast lossless video codec.

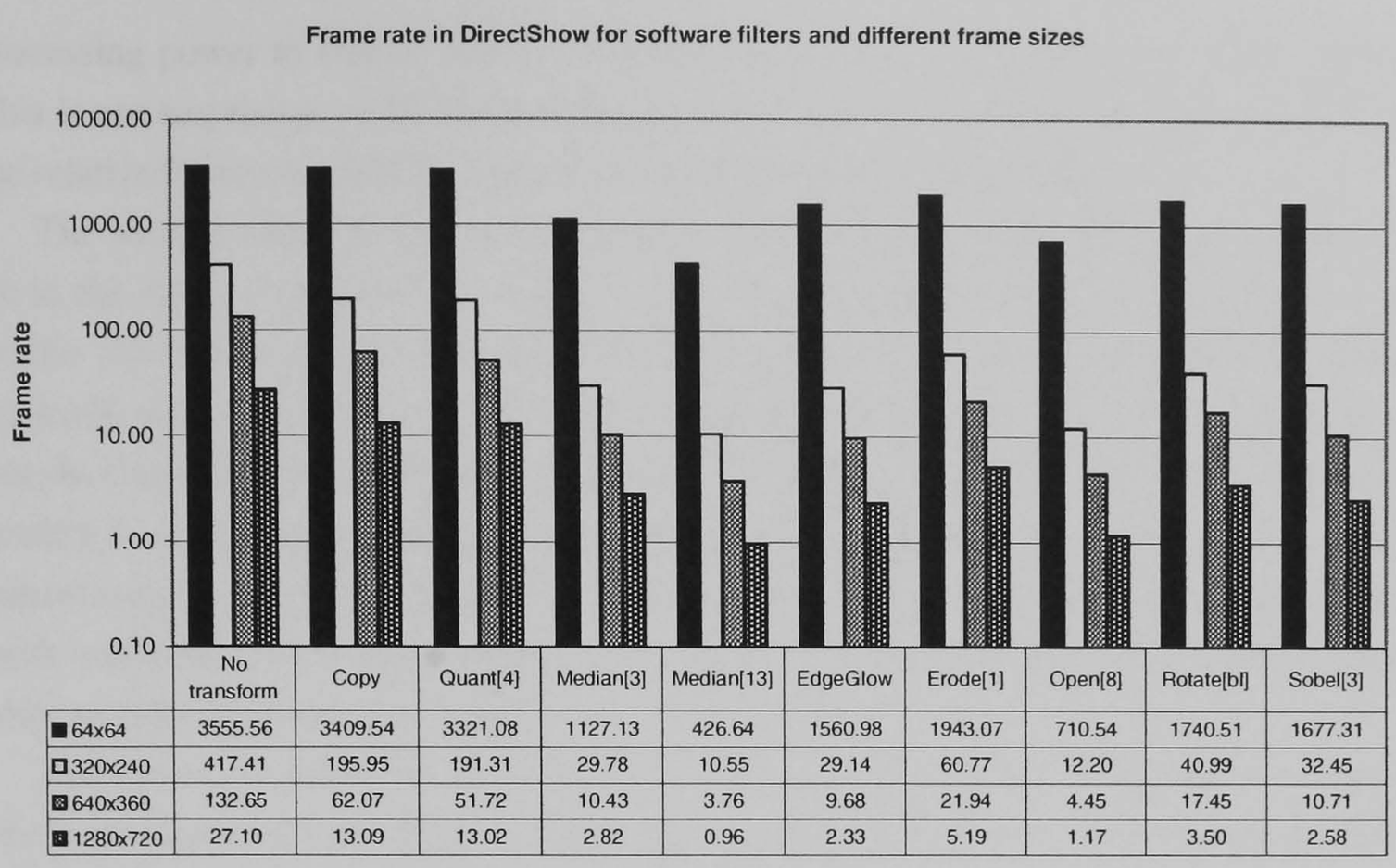
320x240 : Legacy This represents the minimum useful video frame size, and was common during the infancy of computer based video due to its relatively low pixel-count and processing requirements. It is still used for broadcasts over the Internet, and for real-time video capture and processing applications where larger frames cannot be accommodated. Compressed using MPEG-1.

640x360 : DVD This is the pixel based frame size of an NTSC DVD with a 16:9 aspect ratio. Compressed using DIVX (an MPEG-4 derivative).

1280x720 : 720p This is a standard resolution for High Definition (Hi-Def) progressive video, and is used in some DVDs enhanced for computer playback. Compressed using WMV9, a proprietary Microsoft codec (related to MPEG-4).

A real-time frame rate is defined as anything above 30 frames per second. Although the minimum frame rate for smooth video is about 24 frames per second, some extra headroom is needed so that during complex portions of the video the frame-rate does not drop. The video is all processed using RGB32 unless otherwise stated.

All tests were performed using a dual Athlon 1.4GHz machine with 1GB of RAM. Both IGOL and some non-IGOL components (such as video codecs) are multi-threaded, and so



	No transform	Copy	Quant[4]	Median[3]	Median[13]	EdgeGlow	Erode[1]	Open[8]	Rotate[bl]	Sobel[3]
64x64	3555.56	3409.54	3321.08	1127.13	426.64	1560.98	1943.07	710.54	1740.51	1677.31
320x240	417.41	195.95	191.31	29.78	10.55	29.14	60.77	12.20	40.99	32.45
640x360	132.65	62.07	51.72	10.43	3.76	9.68	21.94	4.45	17.45	10.71
1280x720	27.10	13.09	13.02	2.82	0.96	2.33	5.19	1.17	3.50	2.58

Figure 5.1: Frame rates measured in DirectShow for pure software processing. The different bars for each filter are different frame sizes, from small to large. "No Transform" is the frame rate when no filter is being applied, so video is being decoded then sent directly to the screen. "Copy" is a filter which does nothing, so represents the minimum overhead which must be introduced to actually intercept video frames.

will derive some benefit from the second processor, but this will apply equally in all tests whether hardware or software processing is used.

The UltraSonic is populated with three pipes, each containing an xcv1000e. Although the test machine supports PCI-66/64, the UltraSonic is only operating in PCI-33/32 mode.

The RC1000 uses an xcv2000e part, and operates in PCI-33/32 mode.

During tests the machine was not performing any significant network activity, but was maintaining idle connections such as network drives and remote shells.

5.3 Basic Filter Performance

First the performance of pure software is measured to provide a baseline for performance. Figure 5.1 shows the performance, measured as displayed frame rate, for different video filters. The act of copying the video frame to video memory is considered as the point at which that frame is displayed, even if that frame never becomes visible on-screen, so some of the frame rates far exceed the capability of any display.

The first filter, "No Transform" represents a DirectShow graph with no IGOL filter inserted, so video frames are passed straight from the video decompresser to the screen. This provides the minimal frame rate without any filters being applied, and can be used to calculate how much adding filters to the process affects performance. All the videos can be played back at real-time except for the 720p video. The test computer does not have enough

processing power to display this in real-time, even when no processing is being applied. This is not surprising, as Hi-Def resolution on PCs has only recently become possible, but the relative frame-rate will still provide a useful measure of performance.

The second filter, "Copy" is the simplest IGOL filter possible, which does nothing at all to the frame it is presented with, so the video displayed will be exactly the same as for the "No Transform" case. There is always a fixed cost associated with just being able to intercept each video frame, even if it is not altered. Comparing the the two frame rates shows that the Copy transform is less than half the speed of the No Transform case. The reason the penalty is so high is that the video processing is performed in RGB, but the video codec's natural colour space is sub-sampled YUV. Forcing the codec to output RGB32 requires more work on the part of the codec, makes each frame twice the size, and so also makes the final write to video memory take longer.

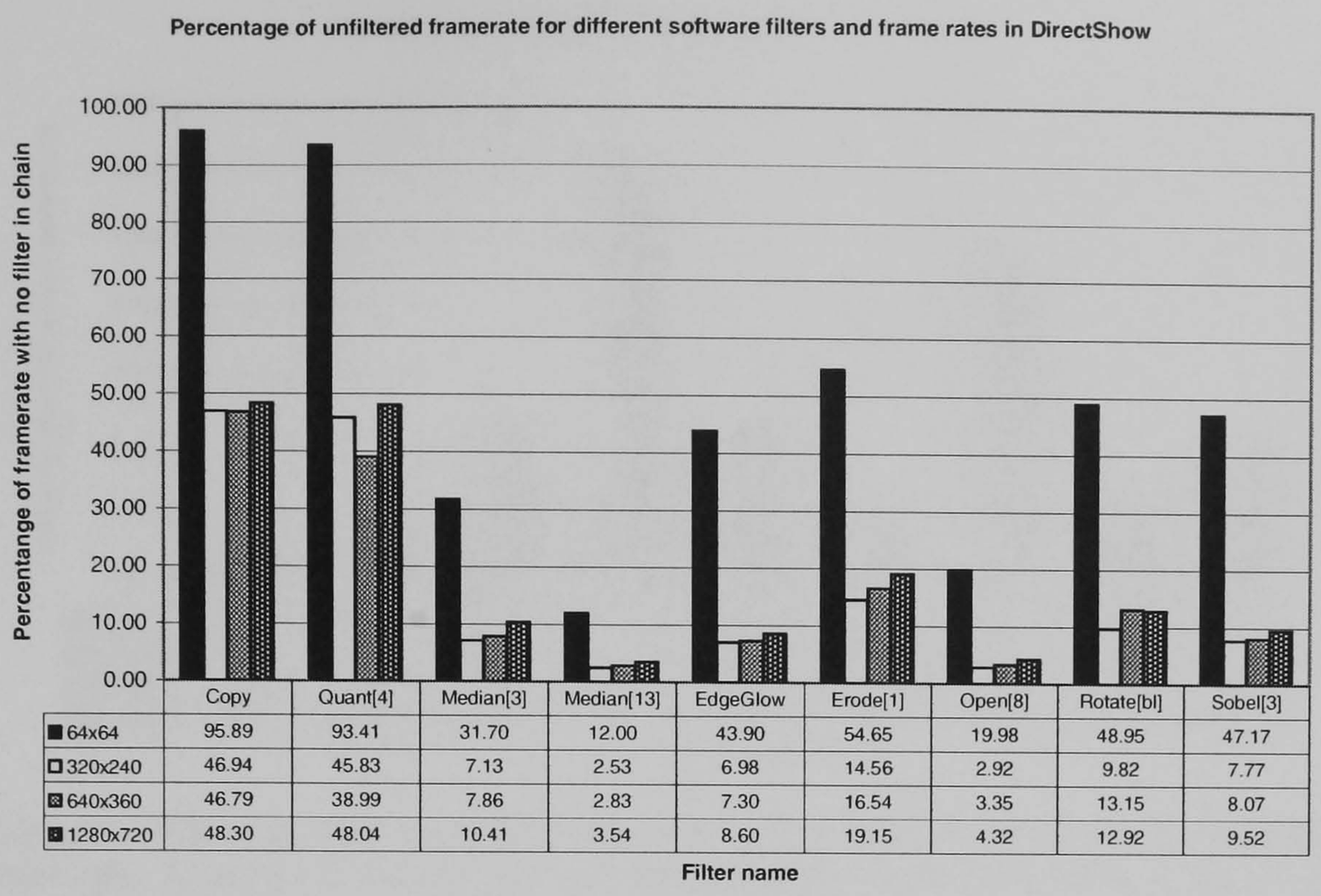
The rest of the filters are general purpose graphics operations or video effects, some of which are parametrised. Where a parameter is used it is included in square brackets after the filter name. Here is a brief summary of the filters:

Quant[n] Quantises each pixel component from 256 input levels to n discrete output levels. Creates large areas of constant colour in the output image. Commonly called "posterise" when used as a video effect, but also used as a component in video compression.

Median[n] For each output pixel the corresponding input pixel and the nearest neighbours of that pixel are collected in an array and then sorted by value. The middle value in the list is then used as the output value. The size of the neighbourhood is controlled by the parameter n. For example, at n=3 a 3x3 neighbourhood is used, at n=13 a 13x13 neighbourhood is used. This filter is used with small values of n as a noise reduction filter, as it removes speckle noise without smoothing edges. At larger values of n it is used as a video effect which smooths out detail and rounds off edges.

EdgeGlow A video effect that selectively blacks out regions depending on how "edgy" that region is. Regions containing no edges are blacked out, while regions which are near an edge are left in. This creates an effect where large areas such as sky and ground are blacked out, but faces and small detail remain.

Erode[n] Similar to the median, expect instead of the middle value the minimum value in the neighbourhood is taken. This is used in image processing as a morphological operation, which means it is used to process information about the shape and structure of visible objects. The visual effect is to make light objects on dark backgrounds shrink and thin out, or to make dark objects bigger. The parameter n controls the number of times that an erode with a 3x3 neighbourhood is applied, although it can also be taken to mean that a single erode with an (n+2)x(n+2) neighbourhood is performed, as the two are equivalent.



	Copy	Quant[4]	Median[3]	Median[13]	EdgeGlow	Erode[1]	Open[8]	Rotate[bl]	Sobel[3]
64x64	95.89	93.41	31.70	12.00	43.90	54.65	19.98	48.95	47.17
320x240	46.94	45.83	7.13	2.53	6.98	14.56	2.92	9.82	7.77
640x360	46.79	38.99	7.86	2.83	7.30	16.54	3.35	13.15	8.07
1280x720	48.30	48.04	10.41	3.54	8.60	19.15	4.32	12.92	9.52

Figure 5.2: Frame rates for software processing expressed as percentage of the frame rate achieved when no filter is being used (the No Transform case). The amount of processing required for each filter is indicated by the relative frame rates.

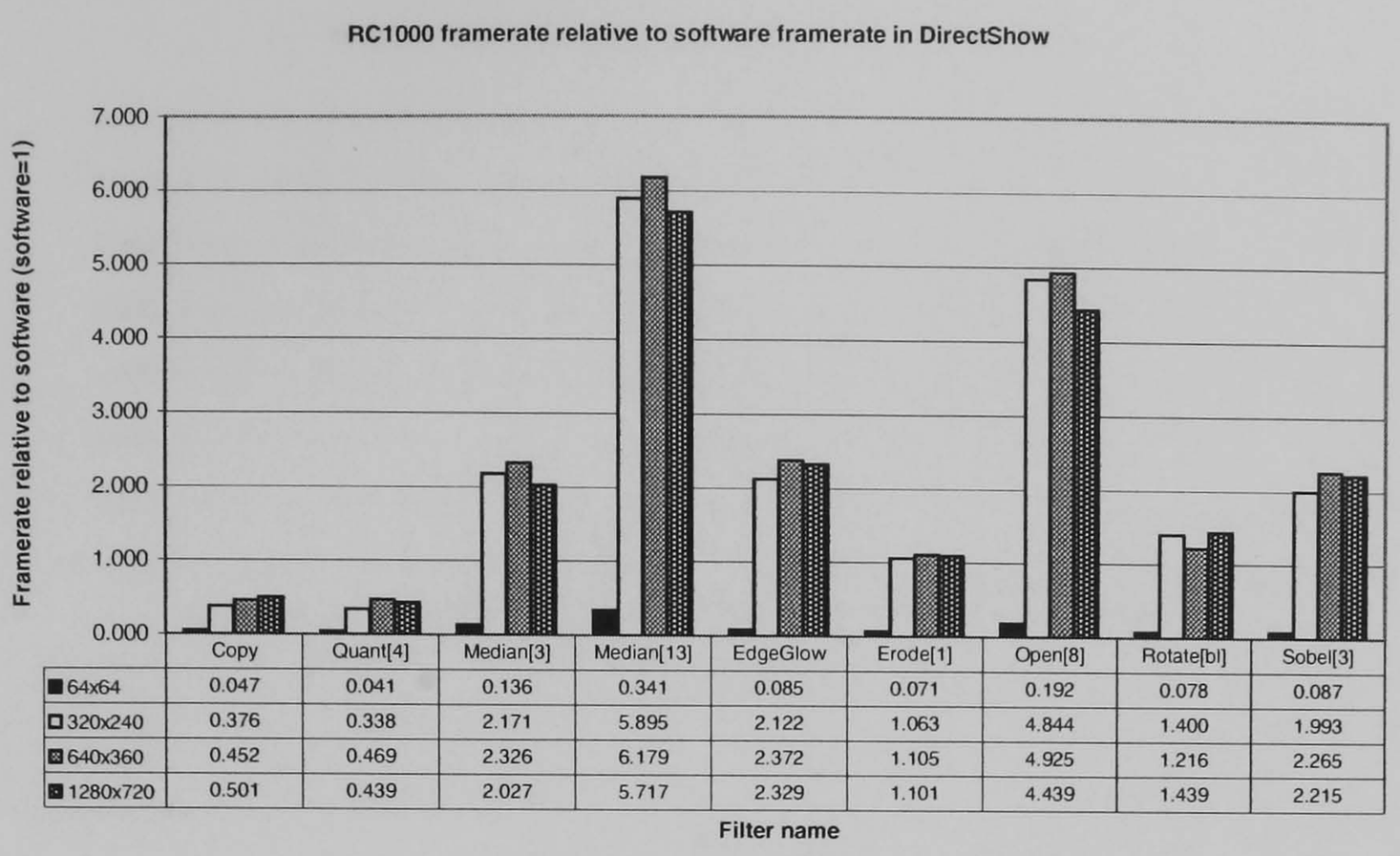
Open[n] Equivalent to Erode[n] followed by Dilate[n]. Removes small light objects or breaks thin connections between light objects.

Rotate[q] General purpose rotation. Along with the obvious parameter controlling how much to turn, there is also the parameter q which controls the quality. If q=nn then for each output pixel the nearest input pixel at the inverse rotation is sampled, while if q=bl the the four nearest input pixels are bilinearly interpolated. Bilinear interpolation provides much better quality, but takes more time.

Sobel[n] The Sobel operator looks at the nxn neighbourhood surrounding the input pixel to determine whether that pixel lies on an edge. The output pixel is then coloured according to how much of an edge occurs in each channel.

In the following results, unless explicitly stated, the frame rates and times include the time taken for both video decompression and display. As this is a test of video processing at interactive rates it is important not to claim large hardware speedups when some other part of the process is the bottleneck in real applications: the performance is measured as experienced by the user.

The amount of processing time used for each filter can be seen in figure 5.1 by comparing the frame rates. The log scale needed to show the y axis hinders interpretation, so in figure 5.2 the frame rate is shown as a percentage of the frame rate achieved when no processing at



	Copy	Quant[4]	Median[3]	Median[13]	EdgeGlow	Erode[1]	Open[8]	Rotate[bl]	Sobel[3]
64x64	0.047	0.041	0.136	0.341	0.085	0.071	0.192	0.078	0.087
320x240	0.376	0.338	2.171	5.895	2.122	1.063	4.844	1.400	1.993
640x360	0.452	0.469	2.326	6.179	2.372	1.105	4.925	1.216	2.265
1280x720	0.501	0.439	2.027	5.717	2.329	1.101	4.439	1.439	2.215

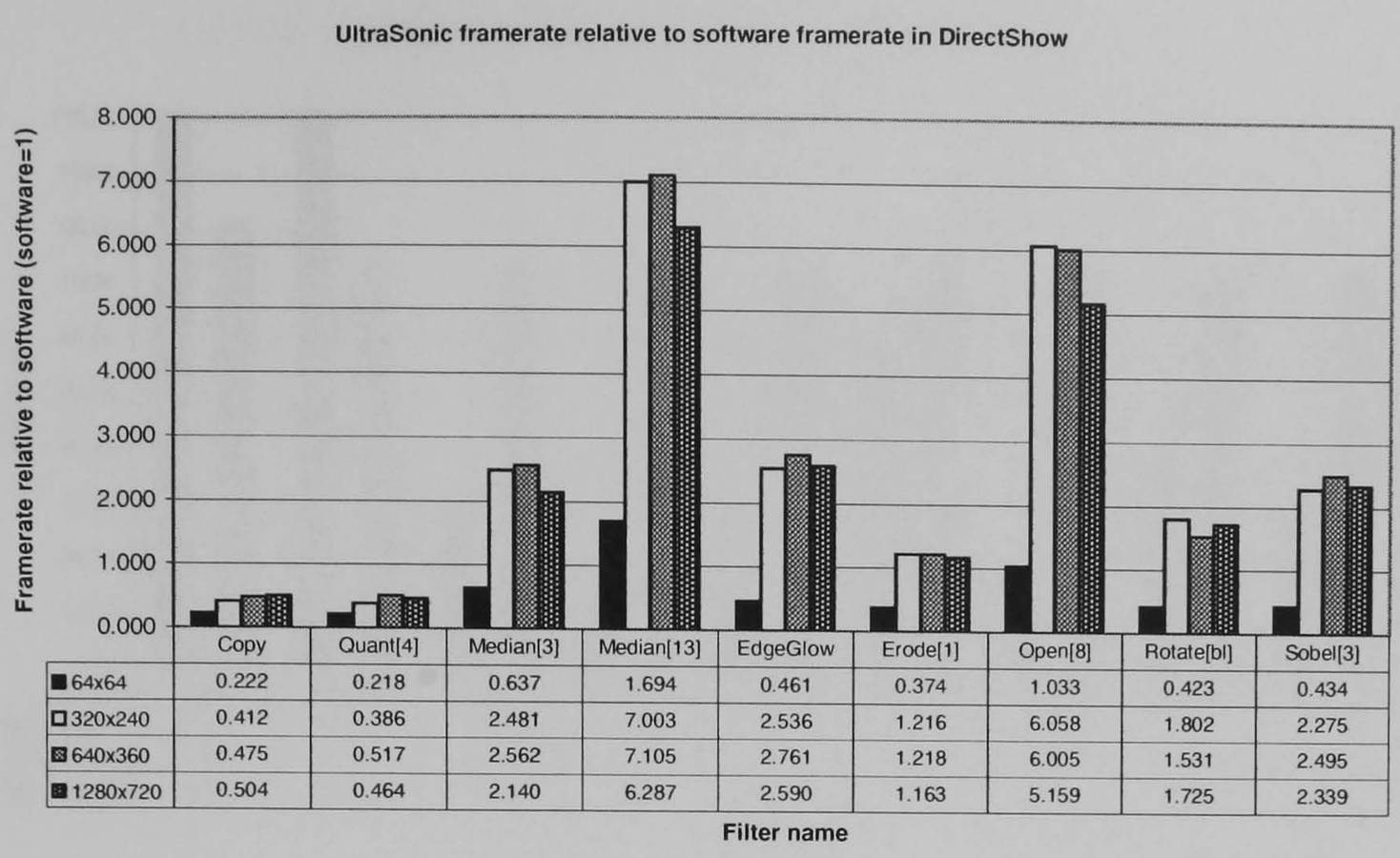
Figure 5.3: Speedup when using RC1000 filters against software filters, shown as relative frame rate. A relative frame rate less than one means the hardware is slower, while greater than one means it is faster.

all is performed. Here Quant is almost as fast as the basic Copy transform, while Median[13] is far slower due to the large amount of memory accesses and operations needed per pixel.

The hardware speed relative to software is now shown, starting with the RC1000. Figure 5.3 shows the processing speed using the RC1000 relative to the software frame rate. For filters that do very little processing, such as Copy and Quant, the hardware is much slower than software, with about half the frame rate. This is because for very simple operations the limit is memory bandwidth rather than processing power, so not only is the hardware slower at performing the processing, but it also has to move the data to and from the card, an overhead the software does not have. This is demonstrated most by the 64x64 Control video, where the overhead for moving data back and forth between the host and card, and the relatively high amount of time spent synchronising the two, mean that the hardware is much slower than software, no matter how complex the processing is.

For the rest of the filters the hardware is faster, but the speedup varies significantly. The Erode filter is implemented in software using MMX vector instructions, so once the communication overheads are included, the hardware is only a little faster. By contrast the Median[13] is much faster in hardware, because even though the software is MMX optimised, each pixel requires many instructions to be executed, and so takes a long time. In hardware most of the processing is performed in parallel, allowing a much larger speedup.

In Figure 5.4 the equivalent results when processing using the UltraSonic (using just one pipe) are shown. The overall shape is the same, but with a slight overall speedup due to the faster data transfer characteristics, and the higher data processing rate. Interestingly a speedup of the Median filter on the 64x64 Control video is achieved, made possible by the



	Copy	Quant[4]	Median[3]	Median[13]	EdgeGlow	Erode[1]	Open[8]	Rotate[bl]	Sobel[3]
64x64	0.222	0.218	0.637	1.694	0.461	0.374	1.033	0.423	0.434
320x240	0.412	0.386	2.481	7.003	2.536	1.216	6.058	1.802	2.275
640x360	0.475	0.517	2.562	7.105	2.761	1.218	6.005	1.531	2.495
1280x720	0.504	0.464	2.140	6.287	2.590	1.163	5.159	1.725	2.339

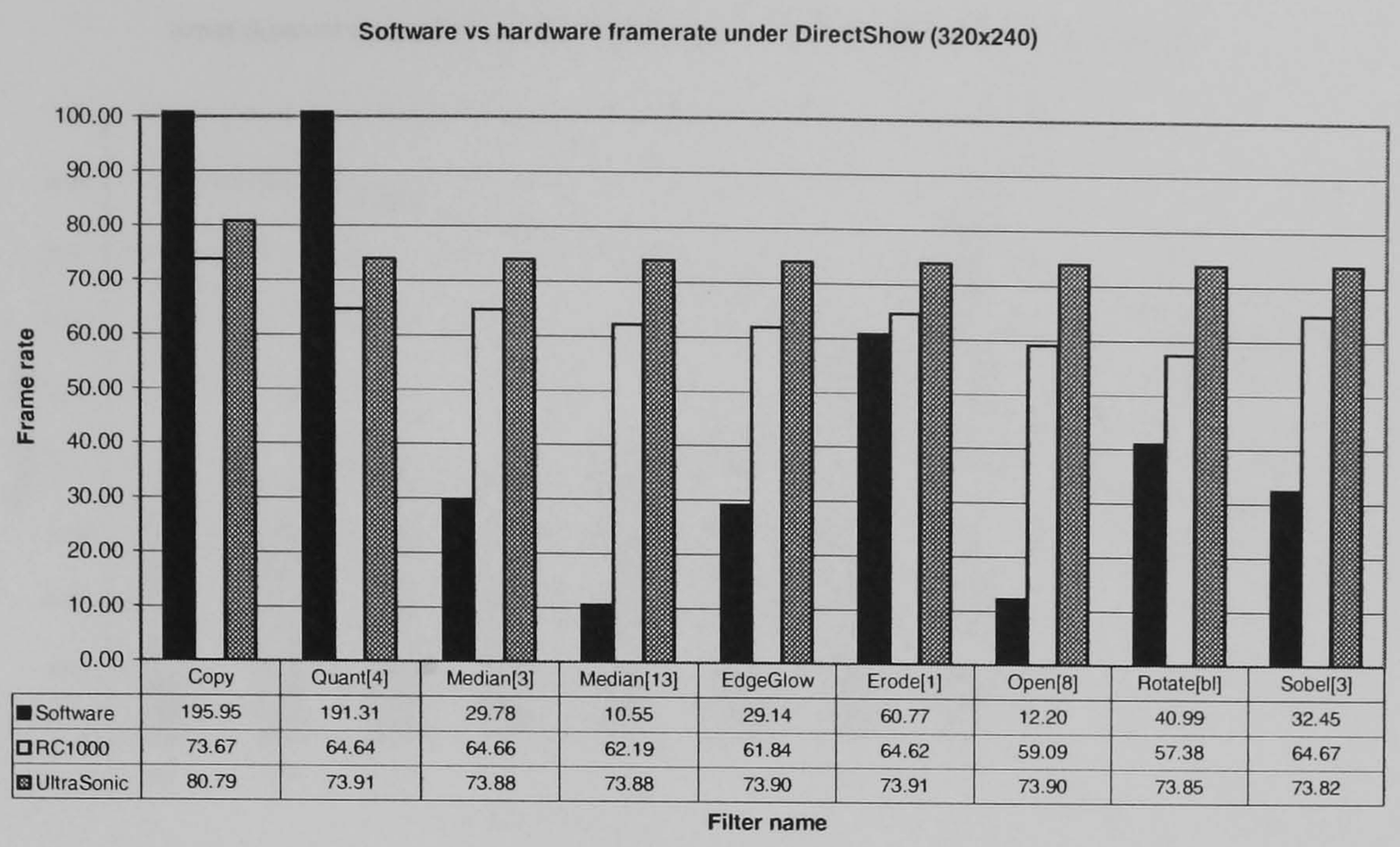
Figure 5.4: Speedup when comparing UltraSonic filters against software filters, shown as relative frame rate. A relative frame rate less than one means the hardware is slower, while greater than one means it is faster. Only one pipe on the UltraSonic is used.

lower per-frame overhead on the UltraSonic compared to the RC1000.

Figure 5.5 shows the raw frame rates for the different software and hardware implementations with 320x240 video. The fixed processing clock of the UltraSonic is obvious, as every filter takes exactly the same amount of time: the pixel-stream nature of the architecture means that any filters which cannot conform to the architecture suffer a large performance penalty.

The RC1000 can apply a varying clock, with a maximum usable clock rate of 41MHz, determined by the maximum speed of the RAM banks. Each RAM access takes three cycles, so the clock rate is actually a divided down 120MHz clock, but it is convenient to use the divided clock so that one ram access can be performed in each clock cycle. Many of the simpler filters such as Quant[4] and Median[3] can run at this maximum clock rate, which provides the optimum performance. Other more complex filters such as Open[8] and EdgeGlow cannot run quite so fast as the circuits use a lot of the chip's resources, increasing the critical path, so the processing takes slightly longer. Even then the large difference between processing speed and data transfer speed means that the frame rate hardly changes: Rotate[bl] takes two clock cycles to process each pixel, so the processing time is doubled compared to Quant[4], but the resulting frame rate only drops slightly.

The Copy transform demonstrates how little the time spend processing on the FPGA contributes towards the total time taken. To implement the hardware Copy operation, data is transferred to the device memory, then immediately transferred back again without any processing being done on the FPGA. For both the RC1000 and the UltraSonic this only increases the frame rate by 15%. The bottleneck here is clearly data transfer and not process-



	Copy	Quant[4]	Median[3]	Median[13]	EdgeGlow	Erode[1]	Open[8]	Rotate[bl]	Sobel[3]
Software	195.95	191.31	29.78	10.55	29.14	60.77	12.20	40.99	32.45
RC1000	73.67	64.64	64.66	62.19	61.84	64.62	59.09	57.38	64.67
UltraSonic	80.79	73.91	73.88	73.88	73.90	73.91	73.90	73.85	73.82

Figure 5.5: Software and hardware frame rates under DirectShow for 320x240 video. The variations in frame rate for the RC1000 are due changes in the maximum clock rate that the filter can run at. The UltraSonic uses a fixed frequency, so all the filters execute at the same rate.

ing time.

In Figure 5.6 this has been explored further, by providing a breakdown of how the time is spent processing each frame, and how the proportions change as the processing clock speed is decreased. All the distinct operations in the filter are timed to provide the breakdown. Although the act of timing slightly interferes with the measured results, the relative timing should still be accurate enough for comparison. The time marked as "external" represents the time taken by the video codec and other components outside the filter, and was calculated by taking the leftover time after processing had finished. The filter used happens to be Median[3], but it could have been any filter which produces one pixel per cycle and can run at up to 40MHz.

It is now clear that processing takes up just one quarter of the time spent in the filter, and only a sixth of the overall time per frame. The processing clock rate has to be reduced to 10MHz, a quarter of its maximum, until processing takes more time than data transfer. This imbalance between processing speed and frame rate is clearly a problem, but one that is will only be solved with higher bus speeds. It is worth considering how performance would change if transfer speed were increased, rather decreasing the processing speed.

This could be calculated by substituting lower transfer times into the graph, but it can also be explored by using the IGOL support for emulated hardware devices. There is an emulated RC1000 device in the HAPI abstract device library, so all that must be provided is a configuration file in the form of a DLL which implements the pseudo-FPGA's functionality. In this case the processing itself is not important, just how long it would take, which can be

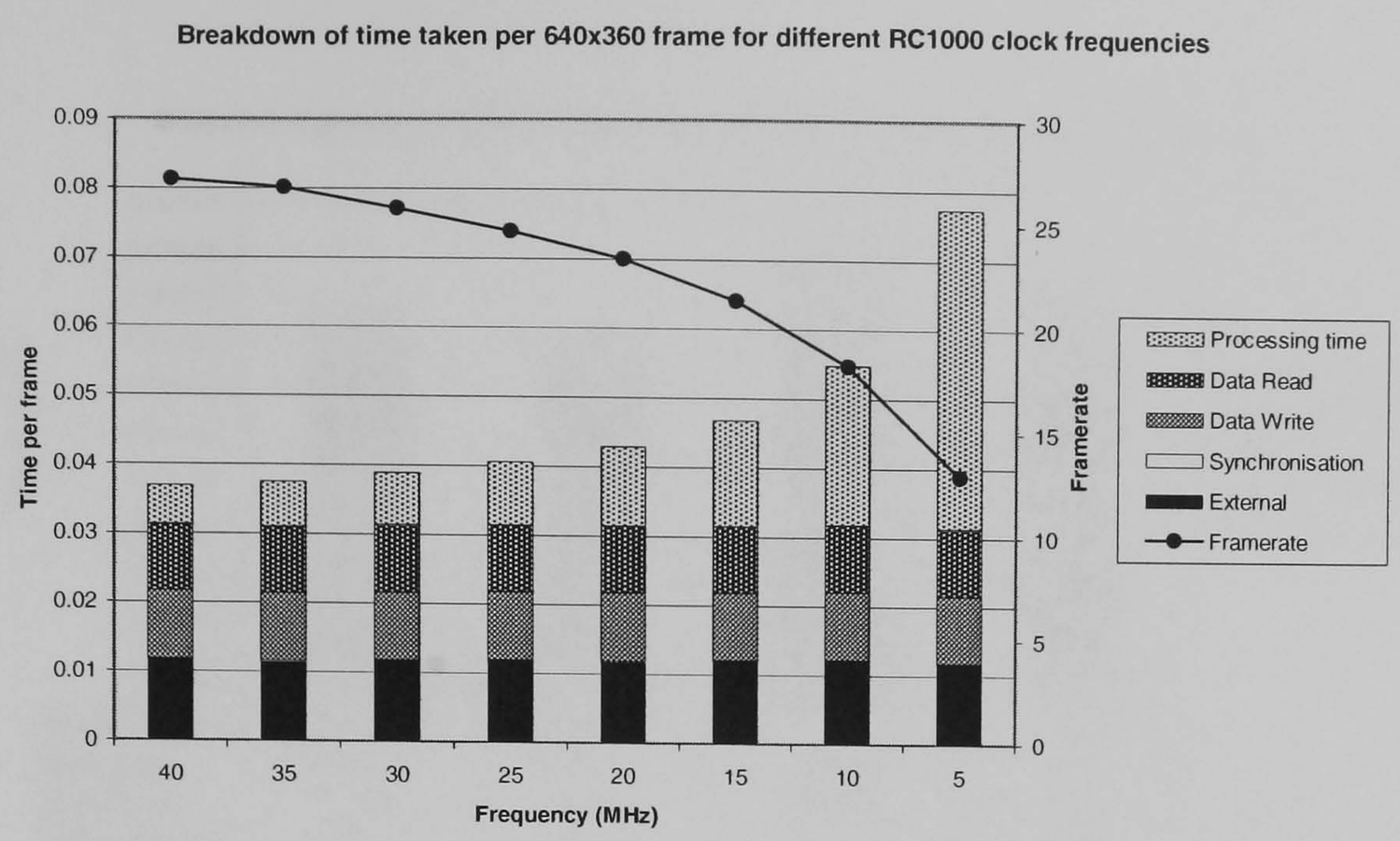


Figure 5.6: Breakdown of per-frame processing time for 640x360 frames on the RC1000, where the filter is processing 1 pixel per cycle. As clock rate decreases the proportion of time spent on processing gradually increases until at 10MHz it is greater than the time spent performing data transfers. The time spent performing synchronisation is less than 1ms, and so is not visible in the graph.

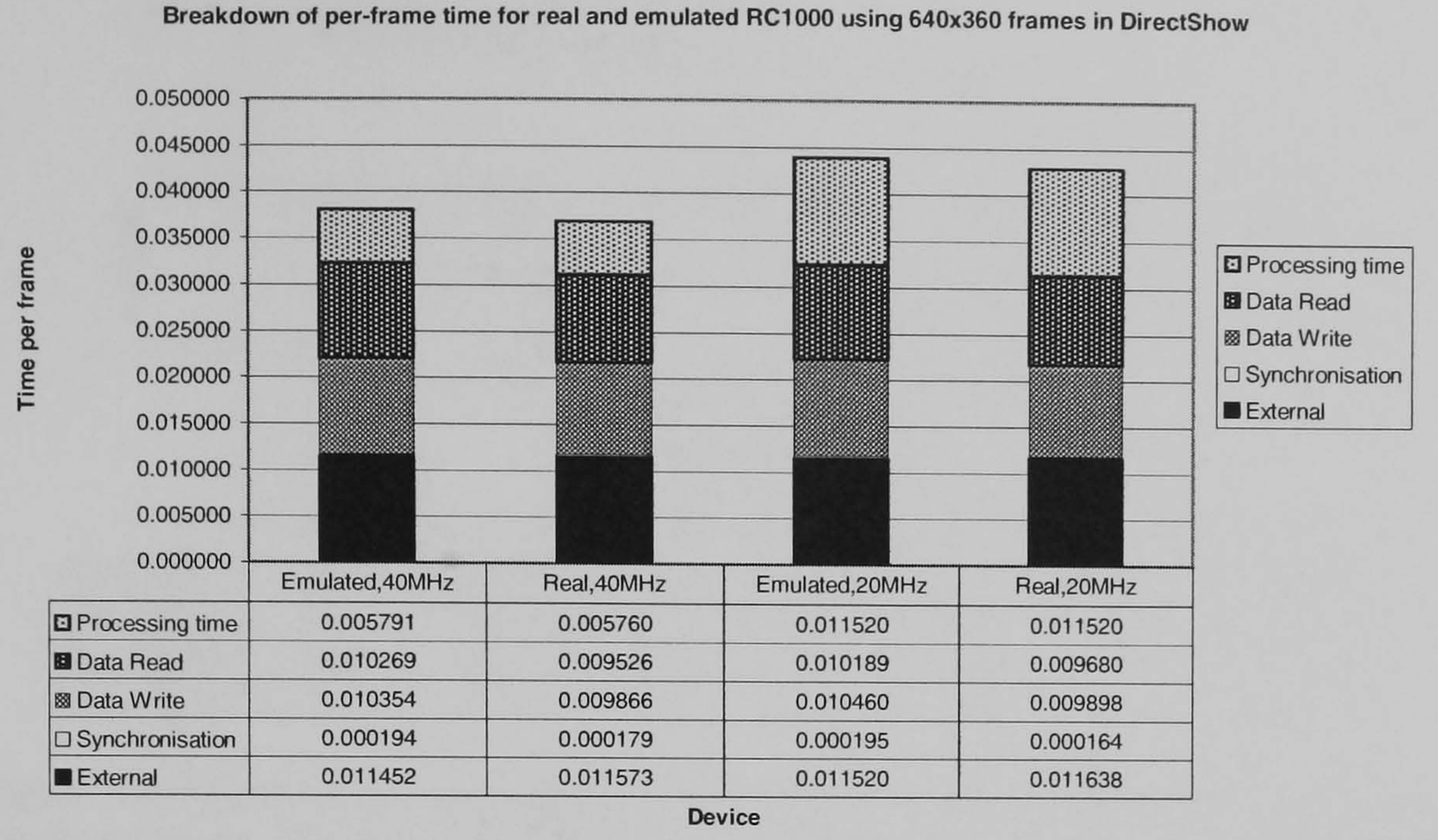
easily implemented as code that simply pauses for the correct amount of time rather than doing any processing.

The logic is compiled into a DLL, and then loaded into an IGOL filter which uses the configuration via an emulated device, and then inserted into a DirectShow graph like any other IGOL Operation. Here the emulation is used to emulate wall-clock time, so the frame-rate seen in DirectShow should be the same as the emulated device.

Figure 5.7 shows the timing breakdown for the emulated and real device at two different clock rates. The emulated device is very slightly overestimating the time taken for each stage, so the emulation frame rate is slightly slower than the real device, but the overall approximation is quite close. The emulated device configuration is now adjusted to scale up the data transfer rate, showing the kind of performance gains that might be expect if the RC1000 data transfer rate were increased to match the processing rate.

The emulated device contains a model of the data-transfer rate of the RC1000 against transfer size, so this will simply be scaled in height. This makes the implicit assumption that on a higher bandwidth bus the card would have lower setup times, which may or may not be true, but as long as small transfers are not used the error should be tolerable.

Figure 5.8 shows the estimated performance for faster bus standards. As expected, it appears that faster data transfers would increase overall processing rates significantly, but this experiment has also demonstrated how emulated devices can be used within applications such as DirectShow to explore hardware alternatives.



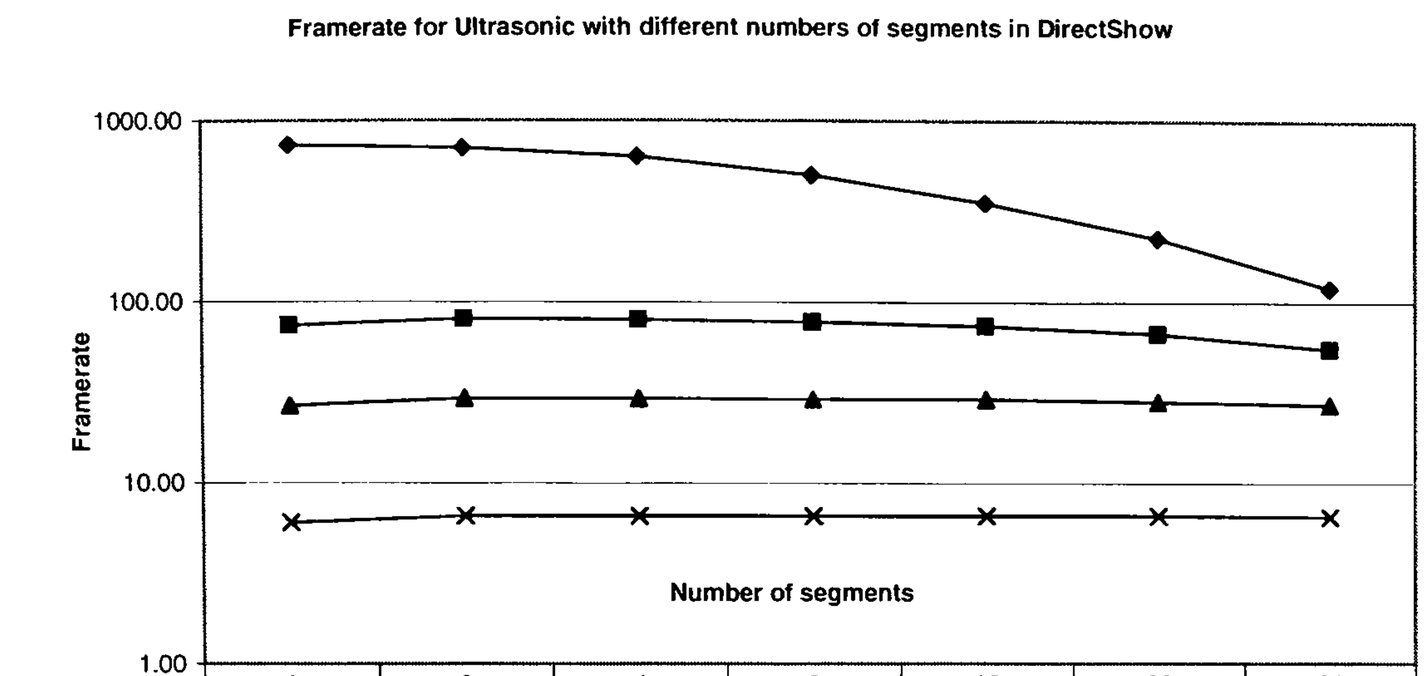
	Emulated,40MHz	Real,40MHz	Emulated,20MHz	Real,20MHz
Processing time	0.005791	0.005760	0.011520	0.011520
Data Read	0.010269	0.009526	0.010189	0.009680
Data Write	0.010354	0.009866	0.010460	0.009898
Synchronisation	0.000194	0.000179	0.000195	0.000164
External	0.011452	0.011573	0.011520	0.011638

Figure 5.7: Comparison of the time per-frame from a real RC1000 and an emulated RC1000 in DirectShow. The emulated RC1000 is merely implementing the communications protocol and timing behaviour of the real RC1000, and does not actually do any processing.



	PCI-32/33	PCI-64/33	PCI-64/66
Processing time	0.005796	0.005771	0.005776
Data Read	0.010196	0.005078	0.002537
Data Write	0.010471	0.005189	0.002609
Synchronisation	0.000194	0.000186	0.000178
External	0.012534	0.012249	0.012369
Framerate	25.52	35.12	42.61

Figure 5.8: Comparison of the time per-frame when using an RC1000 with faster bus speeds. The emulated device estimates the time taken to transfer data by scaling the data-transfer curve for the existing RC1000.



	1	2	4	8	16	32	64
64x64	732.29	705.00	633.18	503.27	354.10	226.72	120.83
320x240	73.82	80.80	79.64	77.45	73.96	67.29	55.34
640x360	26.72	29.46	29.37	29.24	28.99	28.31	27.17
1280x720	6.04	6.59	6.59	6.58	6.57	6.56	6.52

Figure 5.9: Frame rate for an UltraSonic filter using different numbers of processing segments per frame. The filter is Median[3], but the frame rates will be the same for all pixel-per-cycle filters (i.e. almost all UltraSonic video filters).

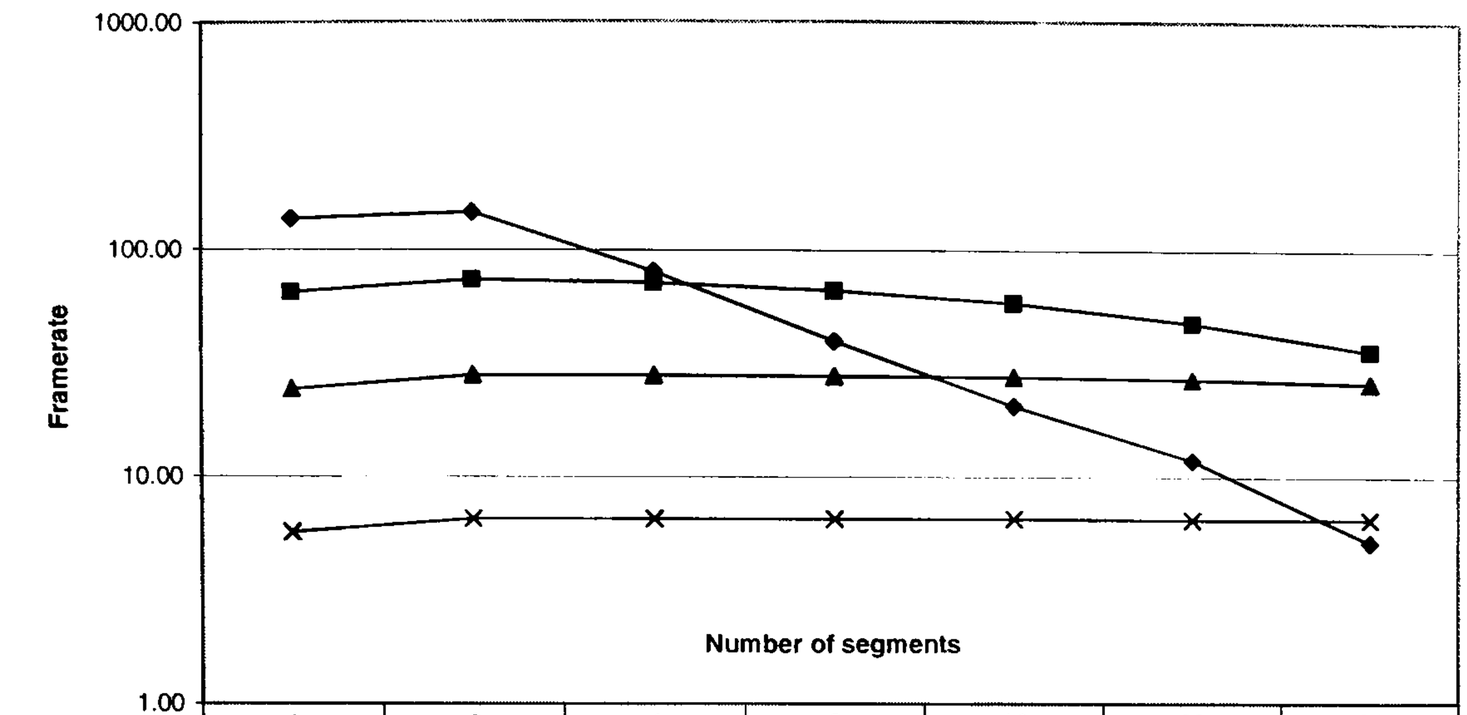
5.4 Segmented Processing

Segmented processing is a way to try and hide some of the latency introduced into data processing by the need to transfer data to and from the device. In video this is handled by processing data in strips, so that except for the first and last strip the data transfer and processing can be performed in parallel.

Segmented processing can be automatically used by IGOL, but only for Operation Implementations that explicitly mark themselves as tileable in their meta-data, and only if the hardware-software communication protocol is being handled by the framework. The hardware components do not know exactly what IGOL will do if they mark themselves as tileable, but it is guaranteed that the semantics and protocol of the operation will remain correct for both application and hardware component.

Figure 5.9 shows how frame rates change in DirectShow when using different numbers of segments per frame. The filter used is a 3x3 Median filter, which means that for each segment two extra scanlines must be transferred at the edge of each segment to allow the calculation of the 3x3 neighbourhood around each output pixel in the segment.

Moving from one segment to two segments, a slight increase in frame rate is, but with any more segments the per-segment overhead for synchronisation and data transfer setup results in a steady decrease in performance.For bigger frame sizes the falloff is slower as the time taken for per-segment synchronisation is independent of the frame size, so it takes more segments to become comparable to the time taken for data transfer and processing. For the 64x64 frame size the two segment case is actually slower, as the per-frame data and



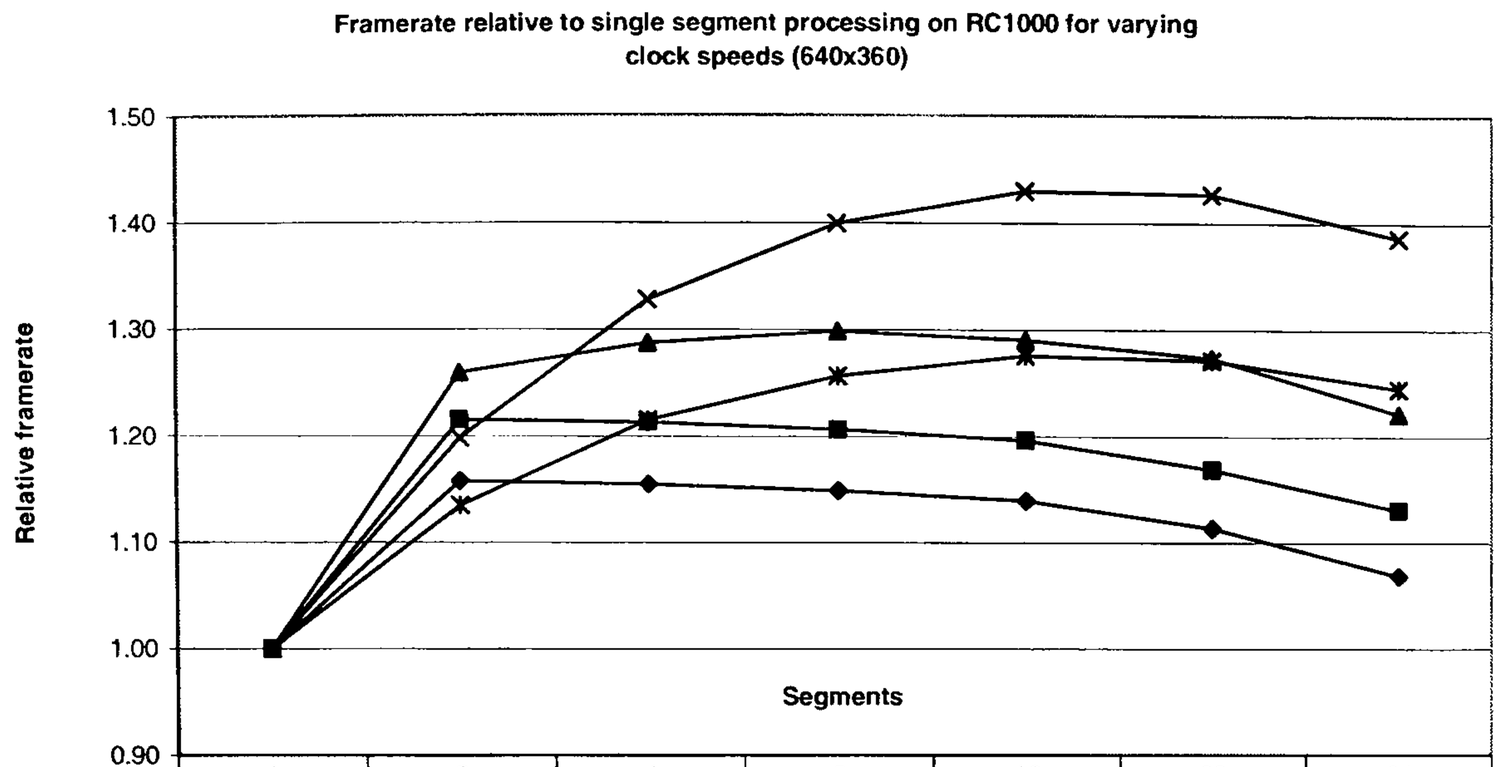
	1	2	4	8	16	32	64
64x64	136.37	145.27	80.08	39.74	20.58	11.85	5.15
320x240	64.60	73.80	71.70	66.38	58.68	47.65	35.98
640x360	24.26	28.09	28.01	27.87	27.63	27.01	25.92
1280x720	5.72	6.56	6.56	6.55	6.54	6.52	6.49

Figure 5.10: Frame rate for an RC1000 filter using different numbers of processing segments per frame. The filter is Median[3], but the frame rates will be approximately the same for all pixel-per-cycle filters which can run at 41MHz.

processing times are already comparable to the synchronisation times.

Figure 5.10 shows the equivalent chart for the RC1000, with a very similar curve. Processing takes slightly longer relative to data transfer compared to the UltraSonic, so the speedup when moving from one segment to two is slightly better, but still only provides an improvement of 5% to 15%. Figure 5.6 showed that data processing only takes a quarter of the time taken up by data transfer, so this modest improvement is still useful. It is important to remember that the external video codec overhead is not changing, so the speedup for just the filter time will be a bit higher.

The same method employed earlier of varying the clock rate to change the amount of time spent in processing versus data transfer is now also used. The situation where processing and data transfer take roughly the same amount of time is the most interesting point, as this should result in the greatest speedup. Figure 5.11 shows the speed using a 640x360 frame size. As hoped, the speedup increases as the processing/data transfer ratio comes closer to one. At 20MHz, where processing takes about half as long as data transfer, the point has been reached where using more than two segments is slightly faster than just two segments. At 10MHz, processing and data take about the same amount of time, so should provide the maximum speedup. The graph shows a gentle curve for the 10MHz, slowly increasing up to a maximum speedup of about 1.4 times at 16 segments, then decreasing again as the synchronisation and setup costs increase. At 5MHz the reverse of our initial



	1	2	4	8	16	32	64
40MHz	1.00	1.16	1.15	1.15	1.14	1.11	1.07
30MHz	1.00	1.22	1.21	1.21	1.20	1.17	1.13
20MHz	1.00	1.26	1.29	1.30	1.29	1.27	1.22
10MHz	1.00	1.20	1.33	1.40	1.43	1.43	1.39
5MHz	1.00	1.13	1.22	1.26	1.28	1.27	1.24

Figure 5.11: Speedup for an RC1000 Median[3] filter using different numbers of processing segments per frame at 640x360. Speedup is relative to the frame rate when using one segment, i.e. standard frame based processing. Different processing clock rates are used to vary the ratio between time take to transfer and process frames.

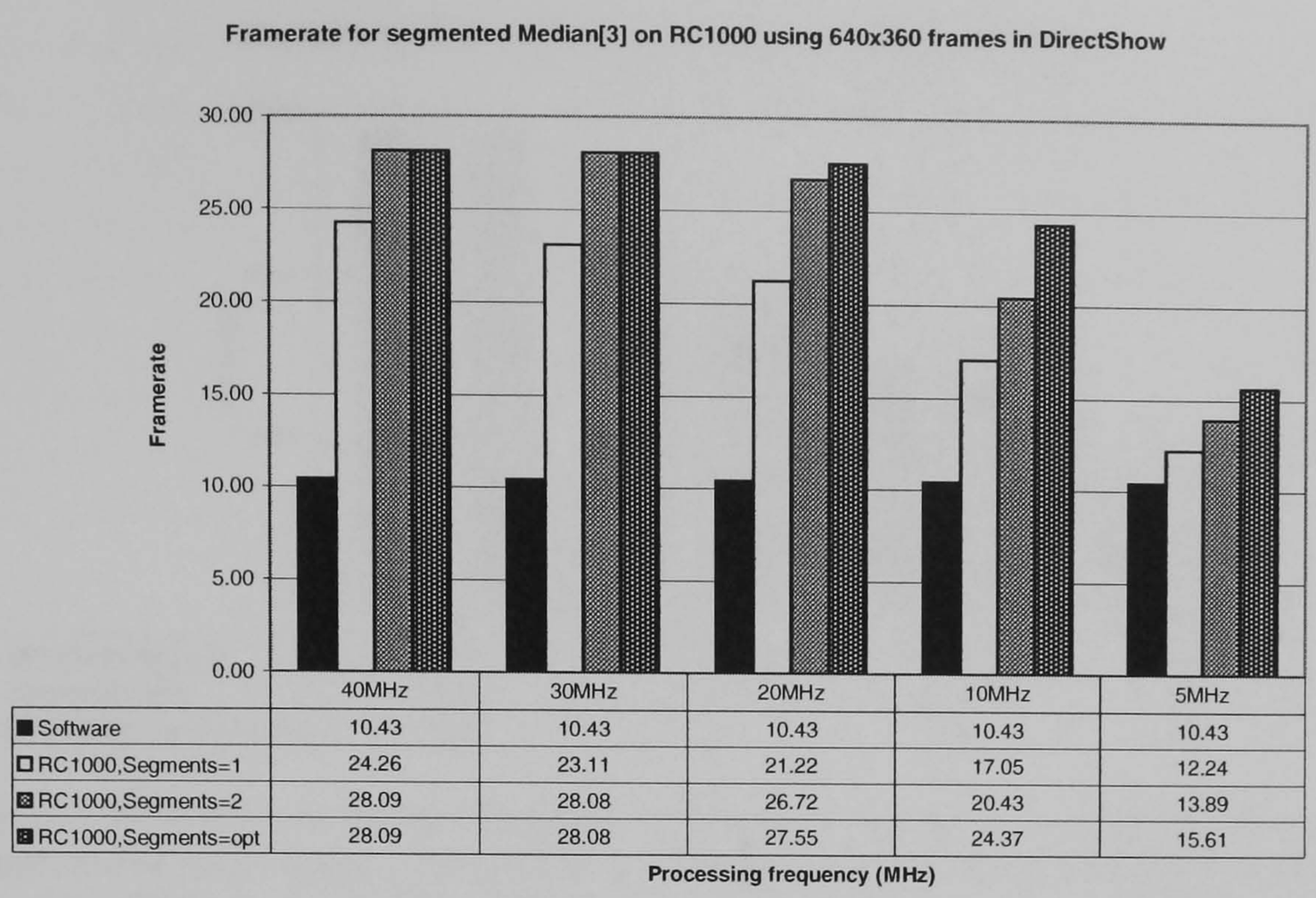
problem occurs: processing now takes much longer than the data transfer, resulting in a smaller speedup.

One point that must be remembered is that this is actually decreasing the overall frame rate when the clock rate is reduced. The speedup figures shown in Figure 5.11 are relative to the single segment case rather than absolute. Figure 5.12 shows the absolute frame rates from which the relative rates are derived. For each frequency it shows the rate for a single segment, two segments, and the optimal number of segments, i.e. the number of segments which was found to maximise frame rate.

By automatically segmenting the processing of video frames, the IGOL framework is able to increase speed by up to 1.15 times, providing a small but potentially important increase in frame rate. This demonstrates how the framework can optimise the connection between application and hardware without either parties involvement.

5.5 Device Sharing

DirectShow is almost unique amongst plug-in architectures, in that it allows a filter discover what other filters are being applied to a video stream, and also where those other filters occur in the chain. This information allows the safe application of some optimisations that are impossible in other applications, due to a lack of knowledge about the big picture. The most



	40MHz	30MHz	20MHz	10MHz	5MHz
Software	10.43	10.43	10.43	10.43	10.43
RC1000,Segments=1	24.26	23.11	21.22	17.05	12.24
RC1000,Segments=2	28.09	28.08	26.72	20.43	13.89
RC1000,Segments=opt	28.09	28.08	27.55	24.37	15.61

Figure 5.12: Frame rate for segmented Median[3] on the RC1000 as the processing clock rate is changed to adjust the ratio between processing time and data transfer time. Although the processing rate is slowing down, it demonstrates that when the ratio is roughly 1:1 (at 10MHz) the best speedup from segmented processing relative to single segment processing is achieved.

important optimisation is that if one hardware filter feeds directly into another hardware filter then the redundant data transfers to and from the device can be eliminated. This is particularly advantageous on the UltraSonic, as the built-in data-paths allow filters placed on pipes to stream data directly to each other.

Three different test scenarios are tested where more than one filter is used to process the same stream of video in DirectShow. The combinations are slightly contrived for benchmarking purposes, but each is based on a real task or performs a desirable effect:

AddNoise;Median[3] The AddNoise filter adds speckles of noise to the images, while the median filter attempts to remove speckles. This could be used to interactively determine how much noise can be tolerated in a video source that will be cleaned up using a median.

Median[9];Sobel The Sobel filter is good at finding edges, but also picks up low level noise as edges, so a smoothing operator is often applied first. A Gaussian filter is often used, as it is fairly quick, but it has the effect of smearing edges at the same time as reducing noise, particularly if the Gaussian kernel is quite large. A Median filter requires more computation, but has the advantage that edges remain crisp, and are also smoothed out. The visual effect is also more interesting than a standalone Sobel.

Median[5];EdgeGlow;Open[4] This is a composition of filters that produces a pleasing

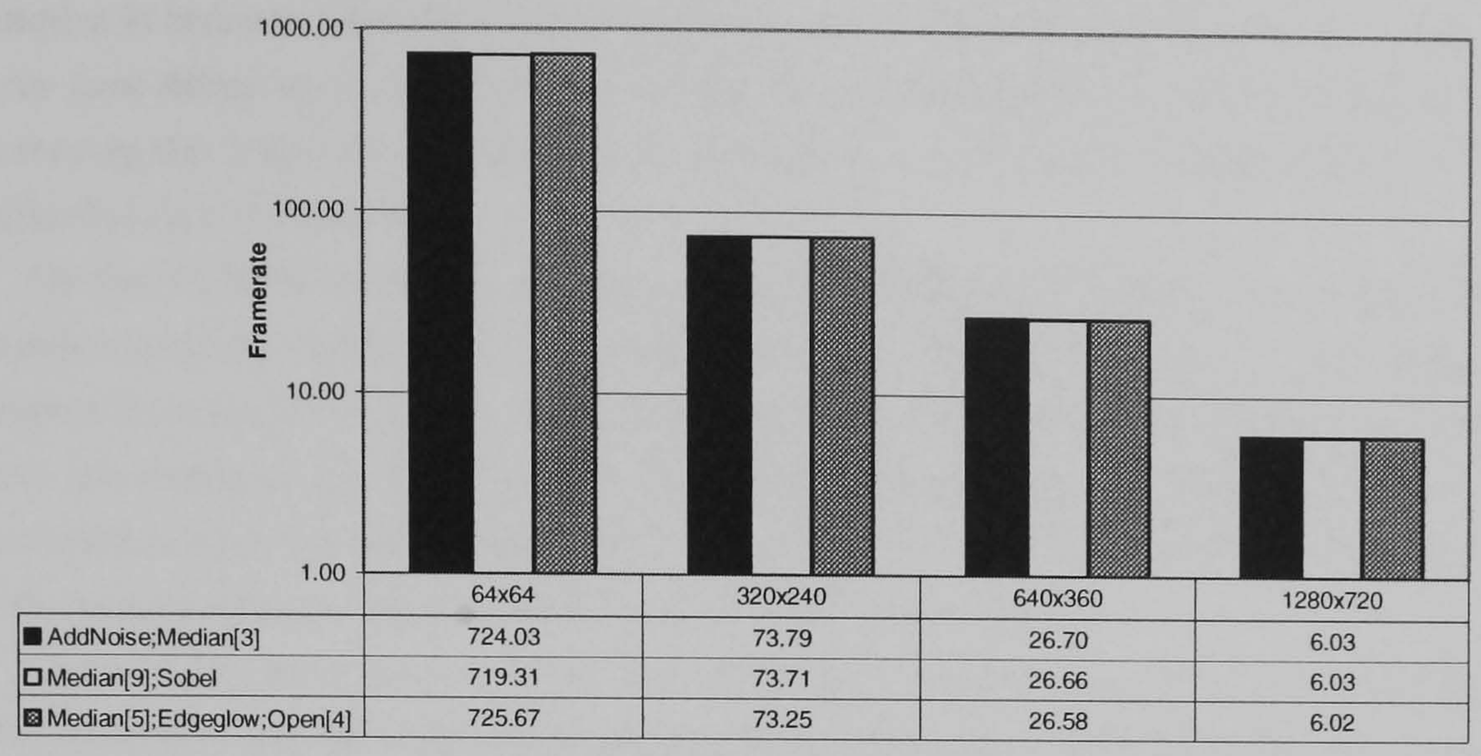


Figure 5.13: Frame rate for the UltraSonic when multiple connected DirectShow filters are implemented using separate pipes in a shared UltraSonic device. Data is streamed between the pipes directly using hardware data-paths, so there is almost no difference between applying two filters or three.

visual effect. The Median provides initial noise reduction and edge smoothing, then the EdgeGlow removes areas with no detail. The final Open spreads the remaining areas out into the black. This kind of composition is typical for providing special effects in video, where multiple filters are chained to achieve something aesthetic, rather than any specific task.

The combinations are ordered roughly by complexity.

Three techniques are supported in IGOL that allow multiple filters to share hardware. The first is to use any device-specific support, such as using the inter-pipe streaming on the UltraSonic. The second is to find or generate configurations that contain two or more of the filters, which can then be selectively executed. The third is to use separate configurations for each filter, and to reconfigure the FPGA between executing each filter. The objective of all these is not just to reduce processing time, but to minimise data transfers between the host and device by re-using data that is already on the device.

The UltraSonic makes this task quite simple due to its internal data paths: once IGOL has determined that the filters are adjacent, and that pipes are available, it configures each pipe with its filter configuration and establishes data channels in hardware between the pipes. At run-time the two or three distinct filters seen by DirectShow behave as a single filter behind the scenes, with one data transfer to move the frame to the hardware, a single call to start processing, then one data transfer to move the finished frame back.

Figure 5.13 shows the frame rate for the three test cases when executed on the UltraSonic. The frame rate is almost exactly the same for all three filters, but if examined carefully a slight decrease in frame rate can be seen for all frame sizes (except for the 64x64

video with three filters). This very slight decline in performance as the filters become more complex is because when filters need to build an nxn neighbourhood the filter has to spend some time filling up its buffers before it can start outputting pixels. This is equivalent to increasing the height of the image to be processed by n scanlines, and produces the very slight decrease in frame rate.

On the RC1000 there is no support for chaining operations together, so the filters must be either spatially multiplexed by combining all the filter designs into one configuration, or temporally multiplexed by reconfiguring the device between applying each filter. In both cases the output of one filter is left in the device memory banks, then used as input to the next filter, so only the initial data transfer of the original frame to the device and the transfer of the processed frame back to the host needs to be performed.

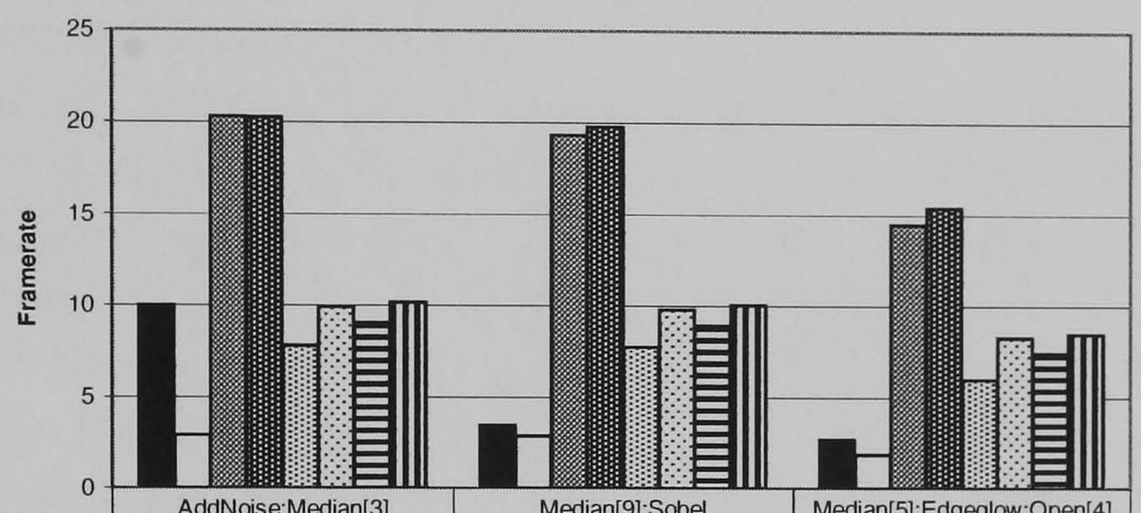
Figure 5.14 shows the results for some of these device sharing strategies in DirectShow. The first two results show the frame rate for software and for completely naive device sharing. This means that each filter in the DirectShow filter chain shares the same device, but does not co-operate with any other filter. As the first filter gets given a frame to process, it configures the FPGA, transfers the data to the device, processes the data, then copies the data back to host memory. This memory buffer is then passed on the the next hardware filter which performs the same set of steps, and so-on. The inefficiency of this strategy is very clear when compared to software, as we the software has been significantly decelerated.

The next two variants, named [shared-config], use a single configuration file with all the filters compiled in. This configuration file must already be compiled for the IGOL DirectShow component to make use of it, as it will not try to generate it at run-time. While IGOL can create these combined configurations automatically (under certain conditions such as available source code and tool-chain availability), DirectShow is primarily an interactive system, and processing chains are expected to last for a relatively short period of time. In this case starting a compilation process in the background might reduce the frame rate for the user, even if the frame rate would eventually be higher. In this test the configurations were explicitly generated before the test started.

The first of the [shared-config] results, [shared-config,fixed-clock] is the standard case, where the design is clocked at the maximum speed indicated by the place-and-route tools for the whole combined circuit. This shows an acceptable speedup over software, even for the first task where the AddNoise component takes very little time in software. Moving from two filters to three filters produces a drop in frame rate because the framework must now wait for three processing stages between transfers instead of two.

The [shared-config,opt-clock] version uses a separate clock rate for each processing stage rather than using the same clock all the time. This is based around the observation that because the filter circuits in the configuration are relatively independent (except where they are connected to RAMs and high level control), they can often be run at different clock rates. When using one filter within the design it usually does not matter if the other filters in the chain are functioning incorrectly, as long as they are not currently processing data. There

Framerate for different RC1000 sharing strategies in DirectShow at 640x360



	AddNoise;Median[3]	Median[9];Sobel	Median[5];Edgeglow;Open[4]
Software	10.01	3.46	2.69
xcv2000e, naive	2.88	2.84	1.85
xcv2000e, shared-config, fixed-clock	20.28	19.32	14.47
xcv2000e, shared-config, opt-clock	20.27	19.78	15.38
xcv2000e, reconfig	7.81	7.74	6.00
xcv1000, reconfig	9.92	9.80	8.27
xcv2000e, reconfig partial	9.03	8.94	7.42
xcv1000, reconfig partial	10.16	10.04	8.50

Figure 5.14: Frame rate for different RC1000 sharing strategies in DirectShow at 640x360. In the shared-config versions all the filters have been combined into one configuration file which is used by all the filter instances. For the fixed-clock variant the clock rate is the same for all the processing stages, while in the opt-clock variant a different clock rate is used for each processing stage, depending on how fast the logic in that stage can operate. The reconfig versions all change the configuration of the FPGA before each processing stage. The v1000 has a smaller configuration file, so it can be swapped out a little faster. The partial variant uses configuration compression to reduce the bit-file size, and so speed up the reconfiguration.

is a possibility that the control logic for a part of the circuit may get into an unrecoverable state, but this can be avoided by resetting the FPGA before starting each stage. This is a fast operation, and does not affect the frame stored in the ram banks. The different clock rates were determined empirically by simply increasing the clock rate until the filtered image started to degrade (in some cases the clock rate was limited by the ram minimum period rather than logic).

Varying the clock rate shows a small, but still noticeable, effect over just using the clock rate indicated by the tools. This technique is not actually part of IGOL, as it is so unsafe: it was implemented as a custom hardware component rather than as a generic optimisation. If this was found to be an important feature then the clock rates would need to be set or determined using proper constraints or timing analysis.

The final four results, tagged [reconfig], are for the case where the device is reconfigured between each filter, but the frame data is left on the device banks. The v2000e provides a speedup of two over both software and the naive shared hardware implementation, except for the simplest combination of filters. The poor performance for the frame size of 640x360 is because the size of the frame, at 921600 bytes, is smaller than the size of the configuration, which is 1270045 bytes. Coupled with the fact that DMAing configuration is slower than data, the bottleneck is now reconfiguration rather than data transfers, so reducing the size of the configuration stream is important in order to increase overall processing speed.

The simplest way to reduce configuration size is to use a smaller FPGA, as configuration size is proportional to the number of logic cells that need to be configured. The RC1000 is also available with a v1000 rather than a v2000e, and the configuration stream is 766055 bytes long, a little over half the size of the v2000e stream. Because the v1000 is a smaller and older device the processing clock rate is a little lower, but the reduction in configuration data results in a 20% increase in performance.

A second way of reducing configuration size is to compress the bitstream, reducing the size of the stream by removing the configuration data for unused parts of the device, effectively creating a partial configuration (and shown as [reconfig,partial] in Figure 5.14). This only helps for designs which do not completely fill the FPGA, so this benefits v2000e designs more than v1000 designs, as something that fits in the v1000 will not completely fill the v2000e, leaving more unused logic that can be compressed out.

Figure 5.15 summarises the frame rates achieved for different implementations at a frame size of 640x360. The UltraSonic is the clear winner due to its built in support for chaining filters. Because the configuration stream size is independent of frame size, the reconfiguring version should provide better performance for a larger frame size. Figure 5.16 shows the same results for 1280x720 frames, and demonstrates that device sharing through reconfiguration does become much more efficient with larger frame sizes. In fact, when three filters are performed the shared configuration version is only slightly faster. This is because the combined configuration runs at a lower per-pixel clock speed, and with such a large frame size the processing time dominates the configuration time.

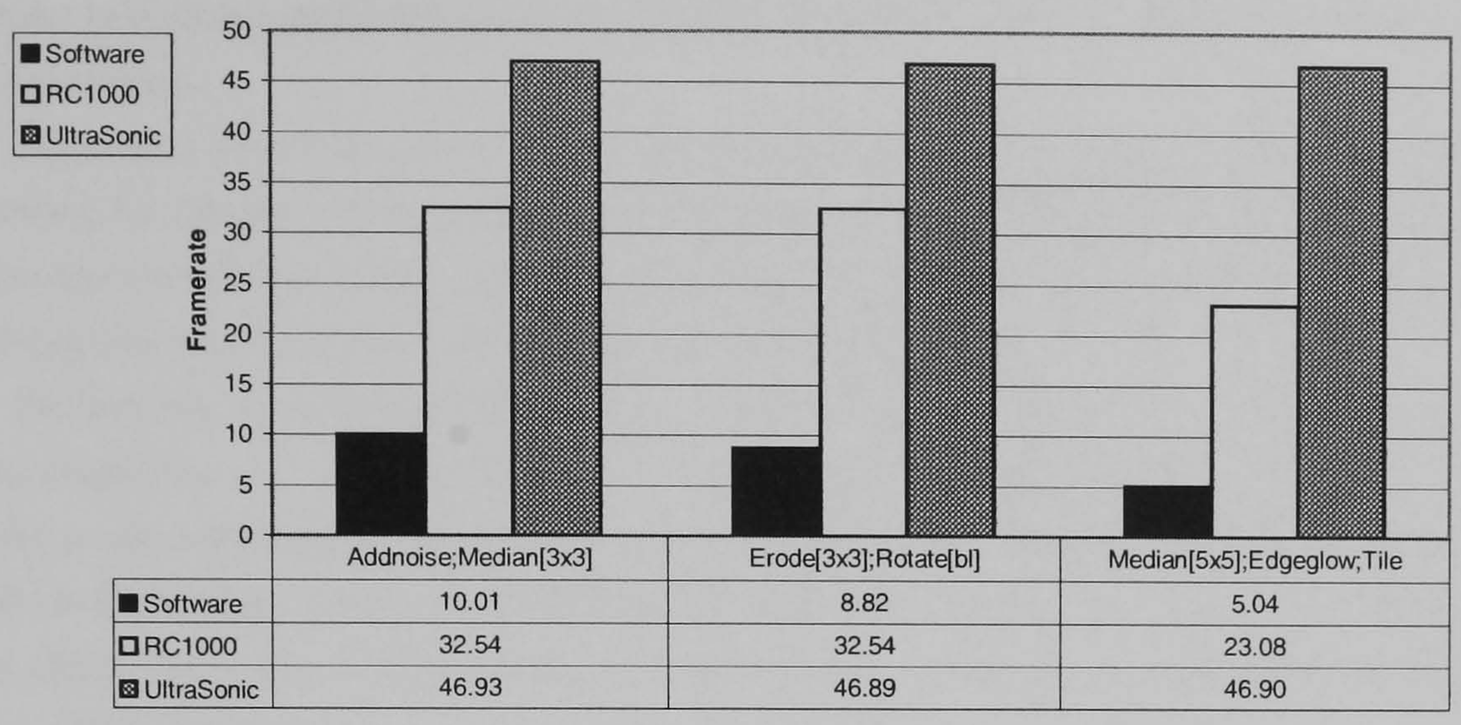


Figure 5.15: Frame rate for chained filters using software and hardware in DirectShow with 640x360 frames.

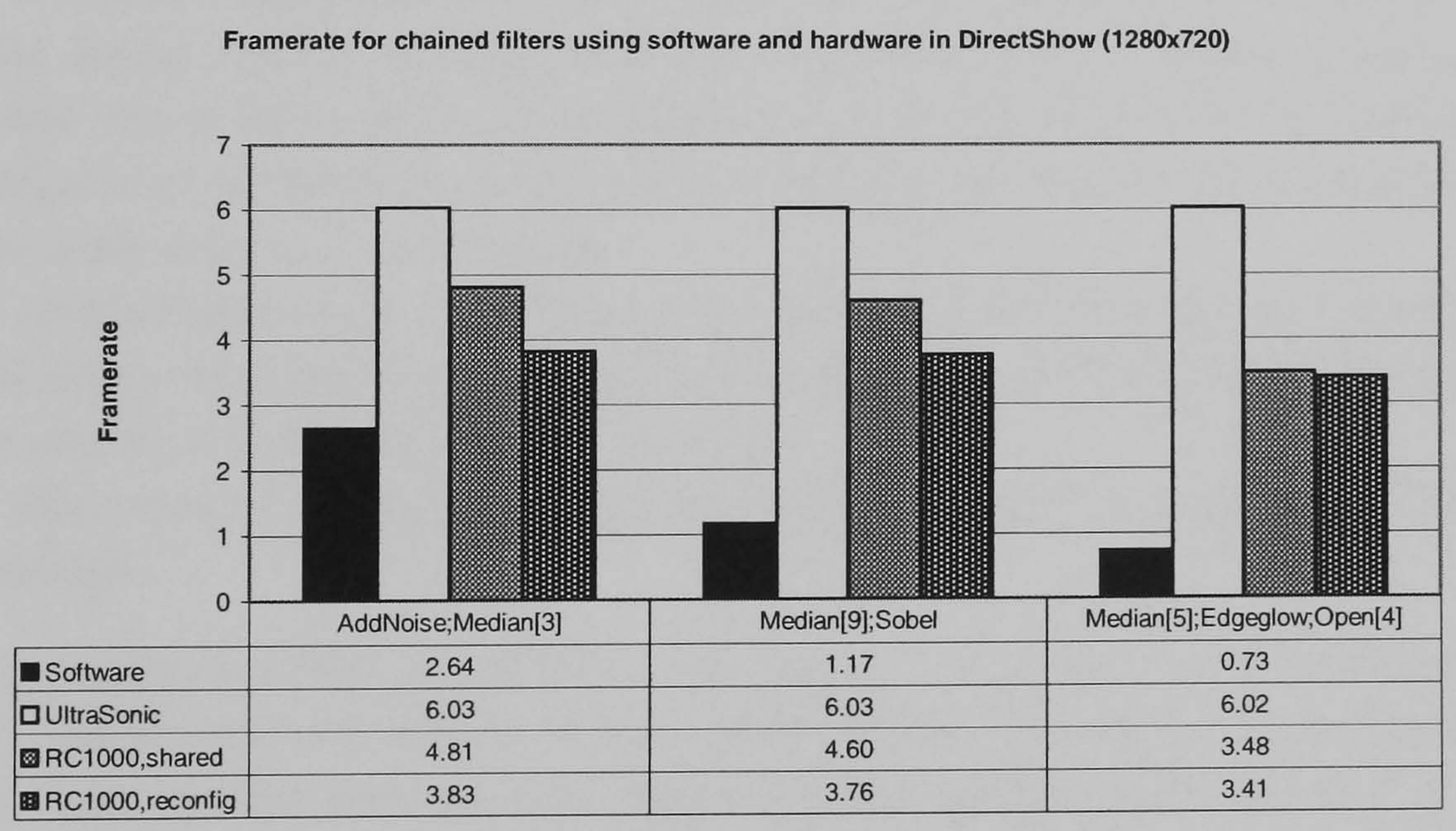


Figure 5.16: Frame rate for chained filters using software and hardware in DirectShow with 1280x720 frames.

5.6 Summary

IGOL can be used to provide generic optimisations to applications and Operations Implementations without needing to change the underlying operation implementation. In this chapter two such generic optimisations have been demonstrated: segmented processing and device sharing.

Segmented processing splits the processing of each video frame into multiple segments, allowing for the processing and transfer of some portions of the frame in parallel. This reduces the overall time taken to process each frame by the filter, and provides a 15% increase in framerate with a fast RC1000 design, and up to 43% for a slower RC1000 design.

Device sharing allows multiple plug-ins within an application to share hardware rather than restricting access to just just one of the plug-ins. Each plug-in appears as a distinct entity to the application, but behind the scenes the IGOL Operation Instances work co-operatively to share access to the underlying hardware. For devices such as the UltraSonic this allows speedups of over 8 times to be achieved, by using internal data-paths to stream data within the device.

For devices which do not support dedicated data-paths for streaming data such as the RC1000 device sharing through spatial multiplexing and temporal multiplexing is also possible. When the filter designs can all fit into one configuration the filters can be applied consecutively without performing redundant data-transfers between host and device. While less efficient than the UltraSonic this still allows a speedup of 2 to 5 times that of the software processing rate.

If combined configurations are not available then IGOL can also apply reconfiguration based sharing whereby each filter reconfigures the FPGA before processing, while intermediate data is left on the device memory banks. Even though the FPGA is configured multiple times per frame this can still provide a processing rate 3 times faster than software, particularly when frame sizes are large.

All these optimisation are automatically provided for hardware operation implementations that use the standard hardware communications protocols in IGOL, and do not require any software to be written by the filter developer.

This results of this chapter demonstrate some of the key benefits of using the IGOL framework:

1. Applications which use the framework to perform processing are able to access many different types of hardware through a single interface. Not only does this allow multiple hardware platforms to be used, it provides handling for the variations of the platforms, such as the number of pipes in the UltraSonic and the specific FPGA part installed on platforms. It also provides a way for applications to access new hardware devices; the DirectShow plug-in was written before the UltraSonic was even created, yet required no changes in order to use the UltraSonic based hardware implementations when they became available.

2. Hardware implementations that are made available through IGOL can delegate the management of configuration and data transfer to the framework, reducing the complexity for hardware designers. None of the hardware components benchmarked in this system had any associated software to handle communications: only the hardware configuration and meta-data needs to be supplied by hardware designers.

3. The framework is able to apply optimisations using information that is only available at run-time, providing performance increases that would be difficult or impossible to provide otherwise. For example, hardware designers cannot anticipate what other component implementations may need to share the hardware at run-time, but by using IGOL this can be automatically managed without requiring any explicit support.

In summary the IGOL framework reduces the work required by hardware and software developers to develop a system, offers increased performance over a simple hardware-software interface due to optimisations applied at run-time, and increases the potential scope for the hardware acceleration of applications due to forwards and backwards compatibility.

Chapter 6

High Quality Uniform Random Number Generation

As FPGAs become larger and capable of running more complex tasks, many applications that used to be executed on high-end PCs and super-computers are migrating into reconfigurable hardware, such as matrix operations [226, 252] and simulations [54, 113, 130, 166, 203, 249]. Simulations and other stochastic applications are a particularly good match for FPGA architectures, as they can often take advantage of the fine and coarse grain parallelism available, and typically require little or no communication between parallel units. The ability to include many floating point units in modern devices [80, 218] also offers the potential to support more complex and sophisticated simulations such as numerical integration and computational physics. High quality uniform random number generators are critical for the correct operation of these simulations, but current hardware based generators focus mainly on efficient resource usage rather than quality [195, 3].

This chapter explores the requirements of such long running simulations, which require very large numbers of high quality random numbers to be supplied at high sample rates, and examine how FPGA based generators can provide such random number streams while also meeting resource constraints. Existing bit-level techniques are not appropriate for long-running parallel Monte-Carlo simulations, instead hardware implementations of software based generators are considered. Due to the differences between general purpose processors and FPGAs, modifications to the software techniques must be made, particularly to the amount of memory required.

The principle contributions of this chapter are:

- design of three high-quality random number generators tailored for efficient hardware implementations;
- demonstration of the hardware performance of these generators, showing how the three generators allow the selection of either logic, RAM blocks or DSP blocks as the

principle resource used;

- evaluation of the statistical quality of all three generators through batteries of statistical tests, showing that all three pass the most stringent empirical tests available.

6.1 Uniform Variate Sources

The traditional technique for generating uniform random numbers in hardware is to use a bitwise generator, such as a collection of Linear Feedback Shift Registers (LFSRs) [10, 3, 75], or Cellular Automata [93, 195]. The motivation for using these techniques in hardware is that they fit well into the bit oriented logic cells of FPGAs, have good performance, and provide a sequence quality sufficient for the tasks to which they had have previously been applied, such as BIST (Built-In Self Test) [94, 180].

There are two problems with this approach. First, the focus on bit-level generators means that any software generators that operate on words are ignored because they are seen as inefficient: there appears to be no examples of word-level generators being used in the FPGA literature. Word level generators may provide a better area/quality/speed trade-off, but there appears to be an assumption that they will be inefficient.

Second, FPGAs are now becoming powerful simulation platforms, taking on some of the tasks that used to be performed on massively parallel software platforms. This means that the problems that the supercomputer community has experienced over the past twenty years with poor-quality random numbers [90, 170] are about to hit the FPGA community. The two main requirements are that any random number generator used must have both a very long period and good statistical randomness over very large numbers of samples.

A long period is needed in long running parallel simulations both because a large number of samples will be consumed, but also to reduce the chance of two processes in the simulation from using same sub-sequence of numbers. Good statistical randomness is easy to provide for small numbers of samples, but the appearance of randomness must be maintained over trillions of samples to avoid bias in long running simulations. These two problems are solved in software by using large amounts of memory, but in hardware these two requirements must be satisfied without consuming large amounts of resources. In this section the requirements for high quality random number generators used in FPGA simulations is investigated, with a particular view towards long running and massively parallel simulations.

6.1.1 Testing Statistical Randomness

A method for measuring the quality of random number generators is needed, so that the difference between generators with good and bad statistical randomness can be distinguished. There are some basic tests that can be applied, for example, taking a histogram of the samples in a sequence and performing a Chi-Squared test [173]. However, almost all random

number generators can pass these simple tests, so it is necessary to utilise many tests looking at lots of different properties that a statistically random sequence should have. This is provided by using test suites or "batteries", which test for many different aspects of randomness, all of which a true random sequence would satisfy.

A minimum test for statistical quality is provided by the Diehard [136] suite. This runs a batch of tests over a sample (approx. 10MB) of random data from the generator being examined, and for each test produces one or more "p-values". If the input sequence is statistically random then the p-values should all be uniformly distributed between 0 and 1, so every value between 0 and 1 should be equally likely. Any values very close to 0 or 1 make it very unlikely that the sequence would be produced by a "perfect" random number generator. However, such a generator could produce any sequence, such as one million zeros in a row, so it is impossible for Diehard to produce any absolute yes/no judgement on whether the generator produces random samples. It is up the user to examine the p-values and decide for themselves, or to establish a testing methodology that will decide in each case.

There are 17 different tests in the Diehard battery, which each look for a specific property that a truly random sequence should have. An example of the type of test performed is the Birthday Spacings test. Here is the description provided for the test in the Diehard documentation.

> Choose m birthdays in a year of n days. List the spacings between the birthdays. If j is the number of values that occur more than once in that list, then j is asymptotically Poisson distributed with mean $m^3/(4n)$. Experience shows n must be quite large, say $n >= 2^{18}$, for comparing the results to the Poisson distribution with that mean. This test uses $n = 2^{24}$ and $m = 2^9$, so that the underlying distribution for j is taken to be Poisson with $lambda = 2^{27}/(2^{26}) = 2$. A sample of 500 j's is taken, and a chi-square goodness of fit test provides a p value. The first test uses bits 1-24 (counting from the left) from integers in the specified file. Then the file is closed and reopened. Next, bits 2-25 are used to provide birthdays, then 3-26 and so on to bits 9-32. Each set of bits provides a p-value, and the nine p-values provide a sample for a KSTEST.

A single number metric for quality is useful when attempting to compare the relative performance of generators, so the approach outlined in [157] is adopted here. Given the set of p-values $p_1 ... p_n$ from Diehard, an overall score s is calculated using the formula:

$$s = \sum_{i=1}^{n} \begin{cases} 4 & \text{if } 0.998 \leq p_i \\ 2 & \text{if } 0.95 \leq p_i \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

This formula simply performs a weighted sum, with the most unlikely p-values receiving a higher weight. Each run of Diehard produces over 215 p-values, so if the p-values were

uniformly distributed the expected sum would be:

$$215 \times (0.002 \times 4 + 0.05 \times 2) = 23.22 \tag{6.2}$$

A value larger or smaller than this would indicate that the p-values are probably not uniformly distributed, though as this score is only for comparison purposes the exact probabilities of the scores are not needed.

As this model was applied it became apparent that it is too simple. The first problem is that while the score is scaled up for unlikely values, no special weighting is given to "impossible" values. Strictly speaking, values of 0 and 1 may occur in normal operation, but the probability of this occurring is extremely low. If any such values are discovered, then the generator under test should be treated with extreme suspicion.

The second problem is that the formula only considers p-values near one, but a p-value near zero is also suspicious. During initial testing some generators were found which returned long sequences where $p = 0$. Using the above formula the generator would be considered to have done extremely well in the test, when in fact it had failed spectacularly.

The modified approach used here is to rescale the weighting bounds, add a new band for "impossible" values, and to make the system symmetric:

$$s = \sum_{i=1}^{n} \begin{cases} 4 & \text{if } p_i = 0 \vee p_i = 1 \\ 2 & \text{if } 0.998 \leq p_i \vee p_i \geq 0.002 \\ 1 & \text{if } 0.95 \leq p_i \vee p_i \geq 0.05 \\ 0 & \text{otherwise} \end{cases} \tag{6.3}$$

This weighting system is just as artificial as the original, but it does a better job of differentiating generators that are not performing very well against those that are failing completely.

One problem with Diehard is that the input sample size is too small: assuming an application consumes random numbers at the rate of 66MHz the 10MB input file only represents 50ms worth of processing time. The sample size used by Diehard cannot be changed, so the results presented here are based upon the aggregate scores of four separate Diehard tests, with the four input datasets created by splitting a contiguous 40MB sequence from the source generator into four quarters. The total number of p values produced by each Diehard run is 215, so over four runs 860 separate ps are accumulated. If the sequence being tested is uniformly distributed then the expected score according to Equation 6.3 should be 93, although due to the probabilistic nature of the tests even perfect generators will produce results in a small region around this point.

Diehard is a well established and respected test, but the limited number of samples tested is a problem when testing random number generators that will be used in long-running simulations. Even the most shortest modern simulation is likely to consume more samples during its one second running time than Diehard uses during its tests, and it is impossible for Diehard to detect flaws over sequence lengths longer than the test sample it is given. New

types of statistical flaws in random sequences have also been found, often driven by specific problems caused be defects in random numbers discovered during simulations. As Diehard has not been updated since its initial release it simply does not know how to look for these defects.

Recently a new battery of tests called TestU01 [121] has been released, which includes more tests and uses larger samples sizes. As well as allowing the tester to pick and choose amongst the tests, it also includes three predefined batteries of tests: SmallCrush, Crush and BigCrush. As the names suggest, these tests each consider different sample sizes, and so have different execution times. SmallCrush uses a number of samples of the same order as used by Diehard, but is able to spot more problems due to more sophisticated and newer testing methods. Crush is a much more strenuous test, using almost 2^{35} samples, and takes a few hours to run to completion. BigCrush offers very extensive testing, using 2^{38} numbers, but can take almost two days to execute. The Crush batteries do not have quite the authority and brand-recognition that Diehard has acquired over the years, but is beginning to gain momentum, particularly in an era where almost every new generator can pass Diehard. Here it is used as an additional suite of tests applied after Diehard, in order to differentiate amongst those generators that can pass Diehard.

The output of the TestU01 batteries is a set of p-values, just as in Diehard, although there are far fewer, for example SmallCrush produces 14 values. It is possible to use the same scoring system as used with Diehard, but there is no longer the averaging effect that using many p-values provides, making the approach less robust. Here TestU01 is used as a tool for rejecting bad generators, rather than as a way of providing some kind of relative comparison (it takes too long to run for these purposes). Instead a system of classifying results as good, suspect or bad is used. If all the p-values are in the range $[0.01, 0.99]$ then that test is consider to have been passed; if there are values are "obviously" headed to 0 or 1 then the test is considered failed; otherwise the results are suspect. This might be clarified by running the test again, or moving onto a bigger test. It is necessary to perform more interpretation with the TestU01 results, and to a certain degree we must rely on the judgement of the tester as to what is failing or passing.

6.1.2 Linear Feedback Shift Registers

LFSRs are commonly used in FPGA applications that need random numbers, both when uniformly distributed numbers are needed as inputs to an application [10] and when used as a base for producing random numbers with different distributions [20, 123, 151]. Each new output from an LFSR sequence is formed from the weighted sum (modulo 2) of the n previously generated values in the sequence:

$$x_i = \sum_{t=1}^{n} w_t \times x_{i-t} \mod 2 \tag{6.4}$$

where the weights $w_1..w_n$ are either 0 or 1. The period of the bit sequence depends on the set of weights chosen and the value of n. The algorithm can be restated in a form more appropriate for hardware by considering the sequence of bits $x_{i-1}..x_{i-n}$ to form a single n bit state vector s:

$$s_i = (\sum_{t=1}^{n}(w_t \times s_{i-1}[t]), s_{i-1}[n], s_{i-1}[n-1], ..., s_{i-1}[2]) \tag{6.5}$$

Stated this way the LFSR has a single n bit state which depends on only the previous state. As the sequence progresses then s will form a cyclic sequence, which if the weights are chosen correctly will cycle through every n bit combination except for all zeros before repeating.

Creating a maximal length sequence requires that the weights are chosen to form a primitive generator polynomial. If Equation 6.4 is expanded for $n = 4$:

$$x_i = w_1 \times x_{i-1} + w_2 \times x_{i-2} + w_3 \times x_{i-3} + w_4 \times x_{i-4} \mod 2 \tag{6.6}$$

and the different terms are viewed as different powers of x, then the polynomial nature is clear. To get a maximum length sequence, weights must be chosen such that the generator polynomial formed by the set of weights is primitive. A primitive polynomial is a polynomial that cannot be factored, and that is also a factor of the polynomial $x^N + 1$ where $N = 2^m - 1$ (the length of the maximum sequence). Tables of primitive polynomials are widely available [110, 75] and more exhaustive tables can be generated on demand where the existing tables do not provide coverage [63].

The number of non-zero weights in the generator polynomial is referred to as the denseness of the polynomial. High density means that many of the previous bits affect the next bit, while low density means that only a few of the previous bits will affect the next state. For a given value of n there exist many primitive polynomials of various densities, though there is no guarantee a polynomial of a specific density can always be found. The lowest density polynomials with only two feedback taps are called trinomials.

There are two ways to run LFSRs, one of which is the forward direction shown above, and referred to as the Fibonacci configuration (due to the resemblance to the Fibonacci sequence). The other method is called the Galois configuration, and is similar to running the Fibonacci generator backwards. It is easiest to express using the state based method:

$$s_i = (s_{i-1}[n-1] \oplus w_1 \times s_{i-1}[1], s_{i-1}[n-2] \oplus w_2 \times s_{i-1}[1], ..., s_{i-1}[n]) \tag{6.7}$$

Either mode can be used in FPGAs, although the Fibonacci mode is more common as the feedback circuit requires fewer LUTs: the Galois configuration requires one exclusive or for every feedback tap, but the Fibonacci configuration requires only one.

The LFSR is a good fit for implementation in contemporary FPGAs, due to the area

and speed advantages in LUT based architectures [5, 3]. The period of an LFSR can easily be made larger than the total number of samples consumed by an application with features such as the Xilinx SRL16. The SRL16 is a very space efficient way of implementing shift registers, which combines the 4-LUT and Flip-Flop in a Virtex family logic element to create a shift register that can be configured to have a length from 1 to 16 bits [237]. Each SRL16 replaces 16 standard Flip-Flops, and 4 SRL16s and a LUT can can be used to create a 64 bit LFSR [75].

One issue with LFSRs is that if the n-bit state of the LFSR is treated as an n-bit variable then there are significant correlations between successive values (particularly in the Fibonacci arrangement). For this reason an n-bit random number generator is usually constructed by taking a single bit output from each of n LFSRs and treating the group as an n-bit number, which is the approach used in the following results.

6.1.3 LFSR Performance

There are two main choices when parametrising an LFSR: feedback length, and tap count. These will affect both the hardware needed to implement them, and the quality of the random numbers generated. There appears to have been no previous investigation into the variations in statistical quality caused by these parameters, so an evaluation is presented here.

Any feedback length of 2 or greater may be used, allowing the length of the sequence to be selected based on the demands of the application. The length needed should be selected based not just on the amount of numbers consumed in the longest possible single run, but rather based on the number of samples consumed in every single execution of the application in its lifetime. This is to try and make sure each different application run is executed using a different sub-sequence from the complete sequence produced by the random number generator. This means that periods of 2^{128} and upwards must be considered for some applications, even though far fewer samples are used in the longest single run. The downside of increasing the feedback length is that it requires more storage to hold all the values in the history.

For a given feedback length the number of taps that may be chosen is constrained, as for feedback lengths there are no primitive polynomials with every possible number of taps. For example, there is no primitive polynomial for a sequence length of 32 with 2 taps. Increasing the number of taps will increase hardware size, as more calculations are needed for each feedback bit.

Given a feedback length and tap count only a subset of possible tap positions are available to us, as only some of the possible polynomials will be primitive. For smaller feedback lengths the number of primitive polynomials is small, but when feedback lengths of 32 and above are used, or more than two taps, the number of polynomials available starts to become much larger. If a specific pattern of taps is desired then it is usually possible to find a primitive polynomial with a similar pattern, for example with all the taps concentrated at

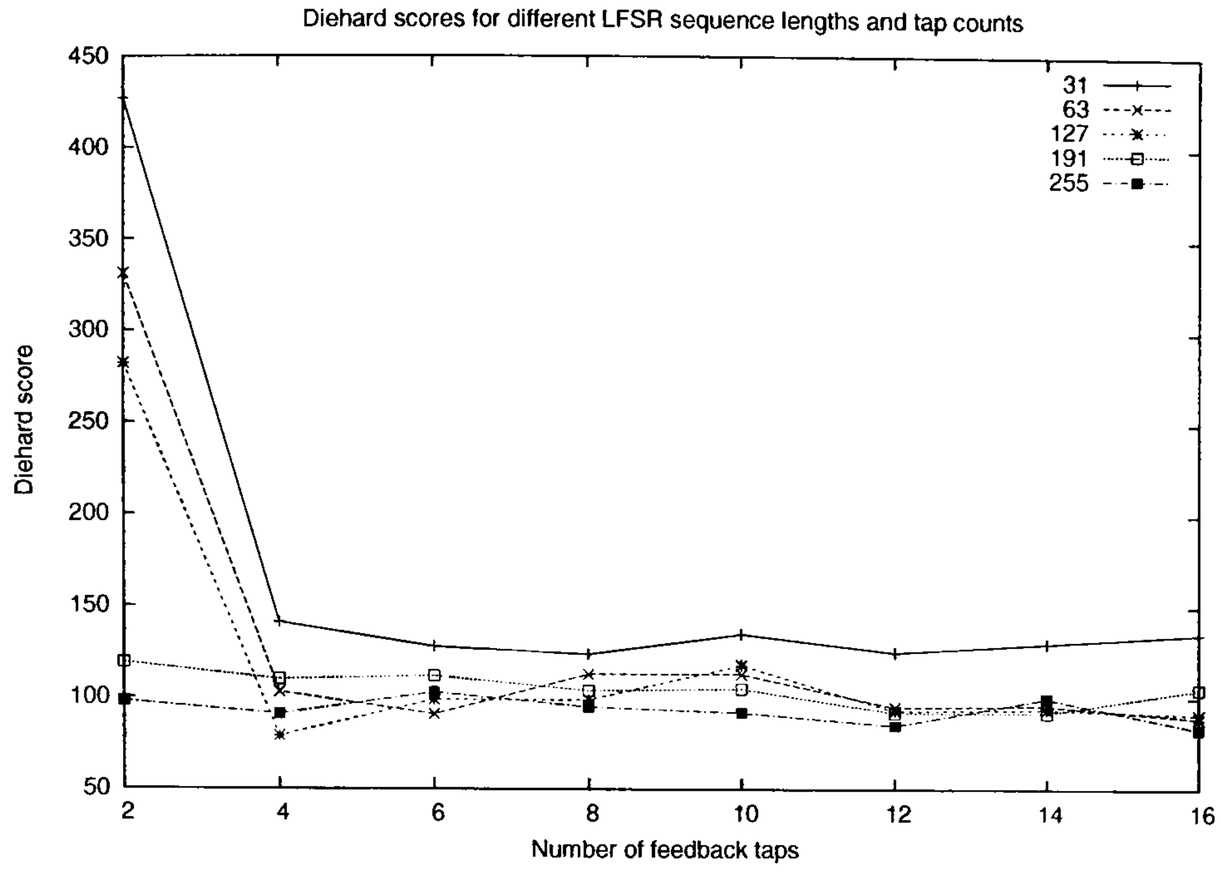


Figure 6.1: Diehard score for LFSRs having different sequence lengths and number of taps

one end, or with more than a certain distance between taps.

Figure 6.1 shows the Diehard score obtained for LFSRs with different lengths and taps counts. It is clear that no matter how many taps are used, the feedback length of 31 is insufficient, as the score is consistently too high, but for longer feedback lengths all generators with four or more taps provide scores of around 93 and so are probably passing the tests. A feedback length of 255 passes the test even for just two taps, showing that increasing the length of the feedback eventually improves quality.

6.2 Xorshift

The Xorshift random number generator is a recently developed generator, introduced by Marsaglia in 2003 [138]. The goal of the generator is to have good performance while maintaining a decent quality level, and could be considered a modern version of the suggested "minimum standard" generator [170]. As the name suggests, the essential operations are xor and shifting, and the required state is of the order of a few words, so it is ideal for FPGA based implementation.

The algorithm is constructed around shift-xor combinations. For example one of the most basic examples given is a three statement 32 bit generator with period $2^{32} - 1$:

$$y = y \oplus (y << 1); \;\; y = y \oplus (y >> 3); \;\; y = y \oplus (y << 10); \tag{6.8}$$

Where $<<$ is a C-style left shift, $>>$ is a right shift, and ; indicates C-style statements, producing a left to right execution of the statements. This is one of a family of generators

with different periods, all based around the $y \oplus (y << a)$ and $y \oplus (y >> a)$ operations. The basic template is:

$$y = y \oplus (y << a); y = y \oplus (y >> b); y = y \oplus (y << c); \tag{6.9}$$

(some re-orderings of the statements are also valid), using a triple (a,b,c) that gives a maximum length period. In the paper, tables of triples are given for both 32 bit and 64 bit versions, although other widths are possible (but not considered).

Higher periods generators with state composed of multiple words are also possible. These take the form:

$$t_n = x_{n-k} \oplus (x_{n-k} << a); \tag{6.10}$$

$$x_n = (x_{n-1} \oplus (x_{n-1} >> b)) \oplus (t_n \oplus (t_n >> c)) \tag{6.11}$$

$$\tag{6.12}$$

allowing k to be increased until a period of sufficient length is achieved.

As shown in Equation 6.12 the nature of the generator becomes clearer: behind the shifts and xors it is another form of linear feedback shift register. The only difference is that the linear feedback is in this case rather more complicated than a simple exclusive-xor. This is explored by Brent [30], where the similarities between Xorshift and a standard bitwise LFSR are compared. In particular he shows that any k bit wide basic generator can be emulated by an array of k LFSRs of length k. For example the simple generator given above with triple (1,3,10) corresponds to the LFSR polynomial:

$$x^{32}+x^{29}+x^{28}+x^{27}+x^{21}+x^{19}+x^{18}+x^{16}+x^{12}+x^{11}+x^{10}+x^{9}+x^{6}+x^{5}+1 \tag{6.13}$$

If the output bits of 32 such LFSRs were concatenated it would be possible to create the same sequence as the XorShift generator, although it would require each LFSR state to be initialised to a precise configuration. The polynomial is very dense, much denser than LFSRs used in hardware which have two, or at most, four taps.

The relation to LFSRs seen as a weak point by Brent, as the statistical properties of LFSRs are well known and have been rejected in the software world. But from a hardware engineers point of view the generator still has potential: XorShift provides a way to replace k shift registers, each with k bits of storage, with a single k bit XorShift generator. In addition the equivalent LFSR is denser than most hardware LFSRs and the taps do not need to be concentrated at one end, so the statistical properties should be better.

The multiple word version from Equation 6.12 was implemented using four 32 bit intermediate states with $(a, b, c) = (11, 19, 8)$, providing a random number generator with a

Family	Slices	LUTs	FFs	Speed
Virtex	113	44	172	170
Virtex2	87	45	146	271
Spartan3	87	45	146	173
Virtex4	113	45	172	366

Table 6.1: Area and speed for XorShift 32 bit generator.

Family	Slices	LUTs	FFs	Speed
Virtex	256	96	460	156
Virtex2	230	100	436	286
Spartan3	230	100	436	198
Virtex4	254	100	460	310

Table 6.2: Area and speed for pipelined XorShift 32 bit generator.

period of $2^{128} - 1$. The equivalent code is:

$$t = (x \oplus (x << 11)); \;\; x = y; \;\; y = z; \;\; z = w; \tag{6.14}$$

$$w = (w \oplus (w >> 19)) \oplus (t \oplus (t >> 8)); \tag{6.15}$$

where t, x, y, z and w are all 32 bit values, and w is sampled after each iteration to provide the sequence output value.

This offers a period comparable to the maximum generally implemented using LFSRs, and is long enough that any contemporary simulation will not manage to wrap the sequence. As it reduces to an LFSR there is no advantage in extending the sequence to try and improve quality, rather this generator is offered as a medium quality generator, suitable for general purpose sourcing of random bits where the more subtle correlations do not matter. One example application is as input to distribution shaping functions such as Gaussian noise generators, where the deficiencies of the sequence are likely to be partially masked by the distribution transform, or for use in Built-in Self Test (BIST) [221]. For applications where very long, very high quality uniform generators are needed then Xorshift is not an appropriate solution (just as a standard LFSR would not be).

Because there are multiple calculations performed during the feedback calculation two versions are implemented. The first does the single feedback cycle in a single cycle, while the second pipelines the feedback over three cycles. As the pipelined version cannot provide a sample every cycle it actually comprises three separate Xorshift generator states, all cycling at once, with the output stream consisting of the interleaved streams of the three separate states. One consideration in using multiple generators in this way is that if initialised incorrectly then two of the generators might end up in quite close parts of the sequence, so the initial states must be chosen with some care.

Table 6.1 and Table 6.2 shows the hardware characteristics of the non-pipelined generator and pipelined generators, respectively. The results show that the pipelining is not needed,

Generator	Lag	Taps	Period	Slices	Diehard	Small-Crush
Xorshift	4	2	$2^{128}-1$	87	Pass (92)	Suspicious
LFSR	32	4	$2^{63}-1$	322	Fail (141)	Fail
LFSR (Core)	32	4	$2^{63}-1$	164	Fail (141)	Fail
LFSR	63	4	$2^{63}-1$	524	Pass (103)	Fail
LFSR (Core)	63	4	$2^{63}-1$	198	Pass (103)	Fail
LFSR	127	4	$2^{127}-1$	960	Pass (79)	Fail
LFSR (Core)	127	4	$2^{127}-1$	228	Pass (79)	Fail

Table 6.3: Comparison of Xorshift and LFSRs in the Virtex 2 architecture

as there is no bottleneck when the feedback is calculated in a single cycle: the pipelined version is twice as large, while only offering marginal speed improvements for the Virtex 2 and Spartan 3 families, against a speed decrease for Virtex and Virtex 4. This appears to be because the feedback logic for each bit maps easily into one level of LUTs, so the main barrier to performance is the routing. In the pipelined case the increased area that the bits must be routed around seems to have a direct affect on the clock rate. Even for the two families where the pipelined version is faster, Virtex 2 and Spartan 3, the speed of the single cycle version is unlikely to create a bottleneck for applications that have need of multi-bit random samples.

Table 6.3 shows the period and area of Xorshift against a selection of LFSRs. All the LFSRs have four taps, as the two tap LFSR is rejected based on its statistical properties. As well as the Handel-C based LFSR implementation it also includes results from using the LFSR core from Xilinx CoreGen [240], which uses explicit placement constraints to achieve very tight packing. Xorshift is smaller than all the LFSRs, while having a longer period and better statistical properties. Even though it doesn't pass Small-Crush it is the only one that comes close, with a single suspicious p-value of 0.007 compared with multiple failed tests for all the LFSRs.

In summary the XorShift has been shown to have two advantages over traditional LFSRs when uniform random numbers are used in FPGAs: the statistical quality of the generated sequence is better, passing Diehard and coming close to passing Small-Crush; and the area required to achieve a given period is much lower, requiring less than half the number of slices used by an LFSR to provide a 32 bit generator with a period of $2^{128}-1$.

6.3 Lagged Fibonacci Generators

In this section the hardware generation of uniform random numbers using the Lagged Fibonacci method is introduced, and it is demonstrated that, by selecting parameters carefully, high quality random number generators can be efficiently produced using these word-level generators. Finally, three different hardware variants are presented, which allow the type of resource used by the random number generators to be customised.

6.3.1 Basic Algorithm

The Lagged Fibonacci generator is a well established random number generator, often recommended by text books [110], and with extensive use in real-world applications. The algorithm is a generalisation of the standard Fibonacci sequence, where previous values in the sequence are combined to provide the next value in the sequence. For example, a generator based on the standard Fibonacci sequence would be:

$$x_n = (x_{n-1} + x_{n-2}) \mod m \tag{6.16}$$

Unlike the classic Fibonacci sequence each addition must be performed modulo m, as each number in the sequence must be stay within the range of a single word. This very simple generator has very bad statistical quality: successive samples in the sequence are highly correlated, and the simple structure means that certain sequences of values are impossible. For example if two low values occur then it is impossible that the next value will be less than the two previous values, i.e.

$$x_{n-1} < m/2 \wedge x_{n-2} < m/2 \implies x_n > x_{n-1} \wedge x_n > x_{n-2}$$

so this means the situation $x_n < x_{n-1} > x_{n-2}$ cannot occur when the generator is in many states.

To reduce these correlations the Lagged Fibonacci is used, which is a generalisation of the Fibonacci sequence that forms the next sample from two samples located further back in the sequence, rather than the two last generated samples. The basic form of the Lagged Fibonacci generator is:

$$x_n = x_{n-r} \bowtie x_{n-s} \text{ where } n > r \wedge r > s \tag{6.17}$$

The operator $\bowtie$ can be potentially any operator, but the most common are addition, subtraction, multiplication and exclusive-or. Except for exclusive-or these operators need to operate mod m, which for integers would usually be 2^k (where k is the width of computer words in bits), and for floating point would be 1. In the case of integer multiplication the sequence must also be restricted to odd integers, as any even number would cause the entire sequence to converge to zeros. The exclusive-or operator can only be used for integer sequences. A specific generator is characterised as $F(r, s, \bowtie)$, so for example a simple but widely used combination is $F(55, 24, +)$.

Selecting good values for r and s is critical to achieving a good sequence length, but the maximum sequence length achieved also depends on the operator chosen. For integer sequences with $m = 2^k$, periods of $(2^r - 1)2^{k-1}$ are achievable when using addition or subtraction as the feedback operator. Larger values of r and s not only increase the overall period, they also improve the quality of the sequence: as the sequence taps move further into

the past the simple correlation between the current sample and the two previously generated samples that it is based on becomes spread across a longer sub-sequence. While still easy to detect by considering correlations at steps of r or s, the likelihood of them affecting applications is much reduced. The major problem with large values of r is the amount of storage needed to buffer all the preceding values in the sequence. Advice in the software world suggests that using r of around 1000 is acceptable [135], but much larger values are often used as memory is usually not a scarce resource in software.

The operator also has a marked affect on the generator quality, beyond the differences in period, as it determines the degree to which the two samples are "mixed up". With multiplication, any bit in the two feedback samples has the potential to affect all the bits in the output sample, as the presence or absence of even a low order bit can cause large scale changes in the new sample. With addition and subtraction there is less intermingling of the different bits: inverting the least significant bit of one of the source samples will on average only affect the few least significant bits of the new sample. This lack of mixing means that with addition and subtraction the bits in different positions will take longer to affect each other, producing a lower quality sequence.

The exclusive-or operator deserves special mention, as it would appear to be the best solution, particularly for hardware. Unfortunately it produces the worst intermingling between bits in different positions of all; in fact there is none. It is more appropriate to consider such a generator as k different random streams of a single bit each rather than a singe stream of k bit wide integers. It is actually identical to a classic LFSR based generator based on bitwise composition of k different LFSRs using the Fibonacci feedback mode. The software community often call this configuration a Tausworthe generator [120], with a maximum period of $2^r - 1$.

Figure 6.2 shows the effect of increasing r on generator quality for addition, multiplication and exclusive-or. All the generators are formed using primitive polynomials (mod 2), and where possible these polynomials are taken from the set of polynomials commonly listed in the literature [110]. Where needed, additional points were gained by choosing appropriate primitive polynomials. The general trend is for quality to increase, but it does not smoothly converge as the feedback length is increased. This is because each primitive polynomial only guarantees maximum sequence length, it does not guarantee any kind of randomness, and some polynomials may produce a less statistically random sequence than others.

The multiplication based generator clearly provides the best statistical quality, starting with a lower score than the addition and exclusive-or and then rapidly descending. With a feedback length of 20 it produces a borderline passing score, and with a feedback length of 36 it is definitely passing Diehard.

The exclusive-or operator maintains a very high score until a feedback length of 17 where it starts to decrease: this is because for these low feedback lengths the period of the generator is actually lower than the number of samples consumed by Diehard, so the tested

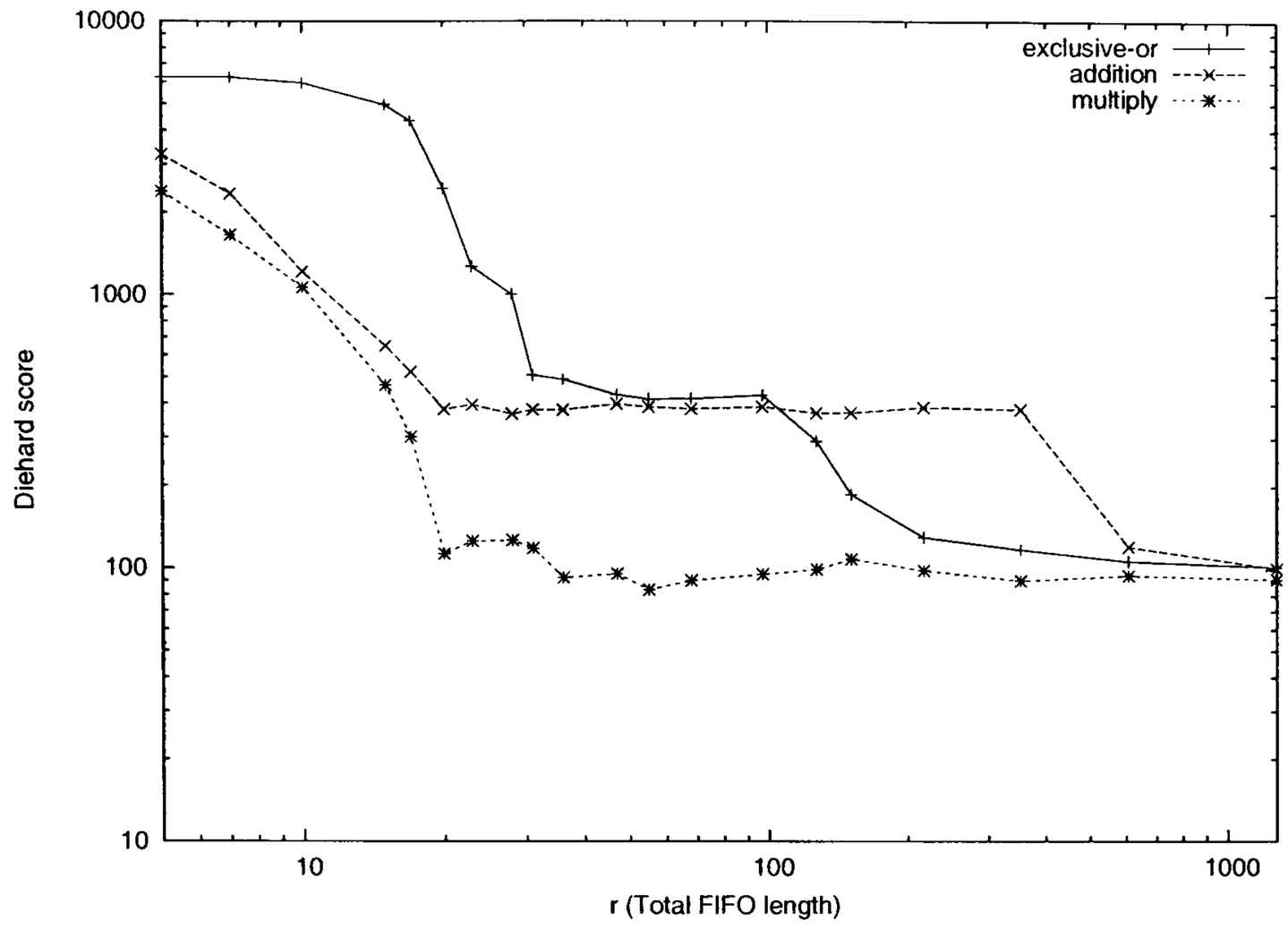


Figure 6.2: This chart shows the changes in quality (according to the Diehard tests) of lagged Fibonacci generators for different operators. As the feedback length increases the overall trend is for quality to increase, but the relationship is not linear.

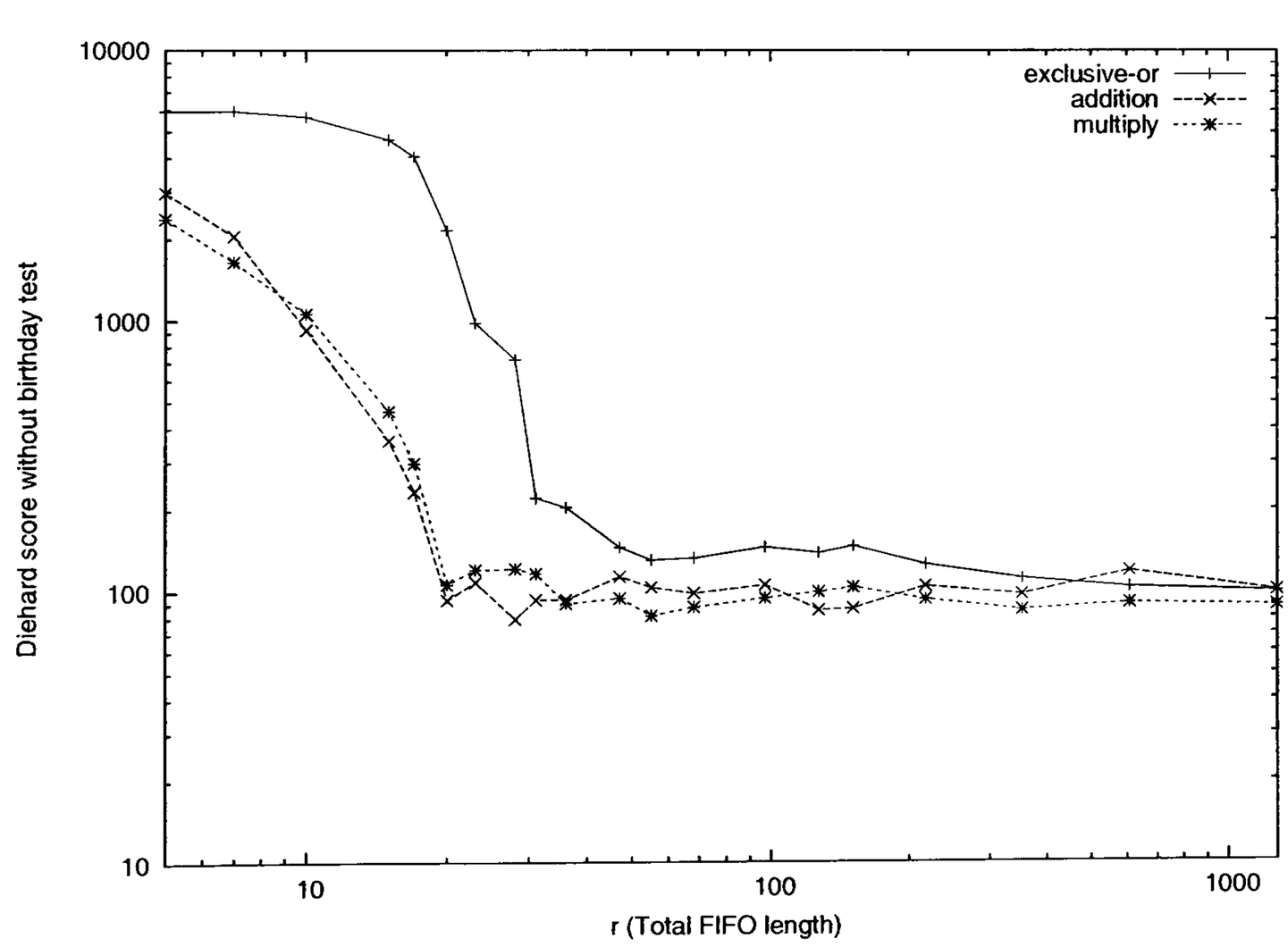


Figure 6.3: This chart shows the quality of the generators if the birthday test score is removed from the total.

sequence repeats. As the feedback length increases to 32 the period exceeds the number of samples tested, so this effect disappears. The plateau from 32 to 100 is mainly due to the contribution from just one test, the Birthday Test. This test contributes a high proportion of p-values to the overall test results, and is difficult for lagged Fibonacci's to pass, so it holds the score roughly level until a feedback length of 100, where the tests can be passed. Once the Birthday Test can be passed the exclusive-or does not pass Diehard convincingly: the curve gradually trends downwards, but it is not clear it has flattened out even using a feedback length of over 1000.

The addition operator is an interesting combination of the behaviour of the multiplication and exclusive-or. During the initial stages the score decreases steadily, although at a lower rate than the multiplicative generator. However after the initial descent the same plateau that the exclusive-or generator produced is found again, although it is hit much earlier, and the score it holds at is slightly lower. So the Birthday Test is again a big problem, and actually lasts for longer under addition than for exclusive-or. It is again unclear whether Diehard is passed with over 1000 feedback stages.

The Birthday Test is clearly difficult to pass for lagged Fibonacci generators using add and xor: in Figure 6.3 the Birthday score is removed from the total and the difference is clear. The addition based generator improves in quality at the same rate as the multiplicative generator, and it could similarly be concluded that it is passing all the tests by about the same point, 36. The exclusive-or score still encounters problems due to its low period, but by the time it reaches 1200 it would appear that it passes Diehard based on this reduced set of tests.

This behaviour is one of the reasons that random number generators that used to look almost perfect can suddenly become obsolete: the addition of a single test to a battery can drastically affect the perceived quality of any and all existing generators. Imagine that a random number generator was needed in an application, and that Diehard did not yet have the Birthday Test. If the minimum length additive generator that passed this early version of Diehard were selected, the 36 stage version would be used. Then when the Birthday Test is introduced it would suddenly be discovered that the generator, which previously passed Diehard, now fails it.

On the other hand, it could have been noted that 36 feedback stages was the minimum for the current battery of tests, but also that quality seemed to broadly increase with feedback length. Based on this information a much larger value than 36 could be used, reasoning that even though it is impossible to tell what future tests will look for in the random sequence, hopefully increasing the parameter that allowed previous tests to be passed will also allow future tests to be passed. Hence a 1200 stage feedback might have been chosen, which would allow Diehard to still pass once the Birthday Test was introduced. However, deciding which parameters to over-estimate in this way and by how much is very difficult, as it is not clear how each parameter effects the statistical randomness.

Based on the Diehard scores shown in this section it appears that, of these basic gen-

erators, only the multiplication based generator provides a high quality generator for reasonable amounts of storage. Based on the principle of over-estimating parameters feedback lengths of at least 500 would be advisable to try and future-proof the generator. If additive or exclusive-or based generators were used then feedback lengths of 4000 or higher would be needed to get a comfortable distance beyond the point where they pass Diehard. This would require storage of 128Kb per generator which. while acceptable in software, is not feasible in FPGAs due to resource limitations.

6.3.2 Hardware Considerations

The hardware considerations for implementing Lagged Fibonacci generators are very similar to those of normal LFSRs, with the major limitation being how many bits of FIFO storage can be dedicated to the generator, dictating the maximum value of r that can be supported. There are two choices for FIFO implementation: shift registers or circular ram queues. These two categories further split into four sub-categories: shift registers can be implemented using single flip-flops or from larger dedicated shift-register resources, while RAMs can be implemented using aggregated logic-based storage or using dedicated RAM blocks.

As with the bit-wise LFSRs, the dedicated shift-register resources such as SRL16s are a useful tool. They allow for dense packing of pure shift register parts of the sequence buffer, and with only two taps, at s and r (the end), almost all of the buffer is pure shift register. To support a k bit wide $F(r, s, \bowtie)$ generator around $r \times k/16$ SRL16s will be required, so the common $F(55, 24, \bowtie)$ configuration with 32 bit wide integers would need around 1760 bits of storage, which maps to 110 SRL16s.

This is appropriate for smaller values of r, but both the literature and the tests performed above suggest that r should be higher for any serious simulations [47], and the example usually given for an barely acceptable minimum generator using exclusive-or is $F(607, 273, \oplus)$. This would require almost 20Kb of storage, or 1214 SRL16s. At these lengths ram based queues are much more appropriate, in particular block ram based queues.

FPGA devices differ in the way that block memories are integrated on the chip. The Virtex family approach is to have one single type of medium-size, medium-capability memory [237]. The Stratix family instead incorporates three different types of memory with different sizes and capabilities [8]. For the purposes of buffering the generated sequence, the RAM organisation is unimportant as long as it can be used as a FIFO. Ideally this would be achieved in a single triple ported ram, with one write port and two read ports, but triple ported ram is not standard on current FPGAs. Instead the buffer has to be implemented as two separate FIFOs using two dual-ported rams, with the first stage of length s and the second of $r - s$. As the block ram size of the architecture is fixed, the generator should be designed around the size of available rams, rather than by choosing a generator then fitting it to the rams. The generator polynomial that maximises the number of utilised elements

Length	Ram type	Generator	Utilisation
32	Stratix/M512	$F(63, 32, \bowtie)$	98.4 (96.8,100)
64		$F(127, 64, \bowtie)$	99.2 (98.4,100)
128	Stratix/M4K	$F(225, 109, \bowtie)$	87.9 (90.6,85.1)
256		$F(503, 255, \bowtie)$	98.2 (96.9,99.6)
512	Virtex/BRAM	$F(977, 497, \bowtie)$	95.4 (97.1,93.8)

Table 6.4: Primitive polynomials for FIFO lengths using strict constraints

Ram type	Length	Generator	Utilisation	Excess	Gain
Stratix/M4K	128	$F(265, 127, \bowtie)$	99.6 (100,99.2)	9	40
Virtex/BRAM	512	$F(1033, 498, \bowtie)$	98.6 (100,97.3)	9	56

Table 6.5: Cases where allowing the polynomial to exceed the ram FIFO length slightly can increase overall sequence length.

in the chosen ram resources should be selected, and the only reason for specifying a specific generator before mapping it to ram resources would be to maintain compatibility with another existing generator, for example one already in use in software.

To select the best generator for a given RAM type requires selection of the primitive polynomial (mod 2) with the largest r, under the constraint that both $r-s$ and s are less than the length of each of the two FIFO segments. This is a strict constraint, as it assumes that if s or $r-s$ exceed the FIFO length by even one then the polynomial must be rejected. The length of the FIFO can always be extended a few stages using normal logic without drastically increasing the area required, so a more practical selection criteria is to try to maximise the overall utilisation of the two ram FIFOs. For a FIFO length of l the primitive polynomial that maximises r is chosen, with the constraint that $min(s, r-s) \leq l \wedge \max(s, r-s) \leq l+d$, where d is some constant amount of logic based buffering that can be accepted. The ability to tolerate a small amount of overflow allows for cases where there are no convenient primitive polynomials such that $s\ r-s \wedge \max(s, r-s) \leq l$, resulting in better utilisation of the block rams.

Table 6.4 shows the best polynomials for different FIFO sizes using the strict constraints. Where the size matches an existing block ram type in a 32 bit configuration it is also shown (The M512 does not support 32 bit widths, so it is assumed that 2 are used per FIFO). The utilisation field shows the overall utilisation of the two block rams, as well as the utilisation for each one. For most of the table sizes the polynomial found under the strict constraint is acceptable, but for sizes such as 128 the utilisation is low, as there are no suitable primitive polynomials at that size. Table 6.5 shows the gains if logic-based buffering is used. For a FIFO of size of 128 an additional 56 stages of buffering are gained, at the cost that 9 of those stages must be implemented in logic.

As well as the method used to implement the sequence buffer, the combination method must also be considered. As has been shown, the most common combination operator used in hardware is exclusive-or, in the shape of the standard LFSR. However, xor does not

allow bits in different positions to influence each other, with the double effect of reducing the period from $(2^r - 1)2^{k-1}$ to $2^r - 1$, and reducing the overall quality of the sequence (compared with addition or subtraction). The result is that to achieve the same quality as the additive generators the bitwise generators need a much longer FIFO to provide the feedback buffer. The improved performance and decreased area of using exclusive-or over a more complicated operator must be balanced against the increase in resources needed to implement the FIFO.

On modern FPGAs exclusive-or is not the only option with good performance, as addition is also very small and very fast. Because addition is a basic building block of many designs and applications, device manufacturers have made sure that addition is as fast as possible, and both methods need one LUT per output bit. Dedicated carry chains have reduced the area and increased the speed of adders [8, 237], and by using pipelining even 32 bit wide adders will not limit clock speed. This means there is actually no reason to use exclusive-or in light of its much poorer statistical quality.

Multiplication is also a common building block, and modern FPGAs now include specialised support in the form of block multipliers [237] and DSP blocks [8, 247]. By using these resources it is possible to implement multiplication based lagged Fibonacci generators without requiring large amounts of LUT based logic to build multipliers. The advantage is that due to the better mixing characteristics of multiplication the feedback buffer can be shorter, but the disadvantage is that DSP blocks are valuable resources. Each DSP block used by the random number generator decreases the number available to the rest of the design, and applications that require high quality random numbers are likely to be using them in numerical processing. This is compounded by the fact that if a 32 bit random number generator is used then the multiplier needs to be a full 33x33 multiplier (the extra bit is to ensure that the LSB can always be odd), which on the Virtex 4 will require 3 DSP blocks. On the Stratix this will require a single DSP block, but as the DSPs are more capable there are correspondingly fewer of them in the first place. One way to reduce the hardware needed is to to break the multiplication into two parts, so instead of a full 32x32 multiply, useg two 16x16 multiplies. This reduces the usage of DSP blocks, so only two Xilinx DSPs are needed, and more of the Stratix DSP is available to be used by other functions. This effectively replaces a single 32 bit sequence with two 16 bit sequences.

6.3.3 Increasing the Tap Count

It has now been determined that using basic lagged Fibonacci generators to implement high quality generators will either require large amounts of RAM and very little logic, in the case of additive and exclusive-xor based generators, or will require large amounts of logic or DSP resources and a moderate amount of RAM, for multiplicative generators. Neither of these combinations is ideal: a generator that used a small amount of RAM and a moderate amount of logic or DSPs would be the best compromise.

Varying the the feedback length of two-tap generators is the only parameter that is commonly varied in software. However, LFSRs can operate with more than two taps, which suggests that additive and multiplicative lagged Fibonaccis with more than two taps can also be used. The use of more than two feedback taps is usually not considered in the software world: it is much faster to use a large memory block, rather than doubling the number of feedback taps within a single memory block. Reading more values per samples requires more memory accesses, requiring more instructions to be executed per output sample, particularly as indexing into the memory buffer must be performed modulo the size of the buffer to act as a FIFO. Trading off high memory use for speed is a good compromise in software.

In hardware almost the opposite situation is found, as memory is relatively expensive, but doing double the number of operations in every cycle will only use more area, it will not reduce speed. For this reason using more than two taps in hardware is a good compromise, particularly if it allows a single expensive multiply to be replaced with two or more comparatively cheap adders.

The existing random number literature does not make it clear that any operator other than exclusive-or can be extended to more than two taps in this way. Based on the information in [27] it appears that a maximal length period can be achieved for any linear recurrence of $2^{w-1}(2^r - 1)$, which supports this conjecture. This was also empirically verified in the specific case of addition, using the primitive polynomials already used for xor based linear recurrence. By using small bit widths of $w = \{2, 3, 4\}$ it is possible to brute-force calculate the period for multiple four and six tap polynomials with $r < 31$. For $w = 1$ the operation is just exclusive-or, which obviously has the correct properties, but for wider words the periods were all $2^{w-1}(2^r - 1)$. This theoretical and empirical evidence justifies the supposition that a multiple tap generator will have maximum period if the polynomial is primitive.

Figure 6.4 shows the Diehard scores for different feedback lengths as the number of taps is increased. It is immediately clear that moving from two taps to four taps has fixed the Birthday spacings problem. This is to be expected, as the reason that lagged Fibonaccis fail the Birthday test is because of the reliance on just two previous samples. Given a set of generated samples, the number of different values that can occur in the next set of samples is limited to just combinations of two values from the previous samples. This means that when the two sets of samples are eventually sorted and the gaps between them compared, some gaps are much more likely than others. When each sample is created from a combination of four others the possible values that can be created from the previous set of samples is far higher, so there is no obvious skew to the distribution of gaps found.

Moving to four taps has drastically improved the generators being tested, to the point where Diehard is no longer able to tell the difference between them. Intuitively this suggests that six or eight taps would also improve the generators, but a more powerful test must be applied, so TestU01 is needed. Using Diehard and SmallCrush a four tap additive generator with a feedback length of 20 passes both. However, if Crush is then applied it fails 7 out of the 120 tests. Of particular interest is that it has failed a much more aggressive version of the

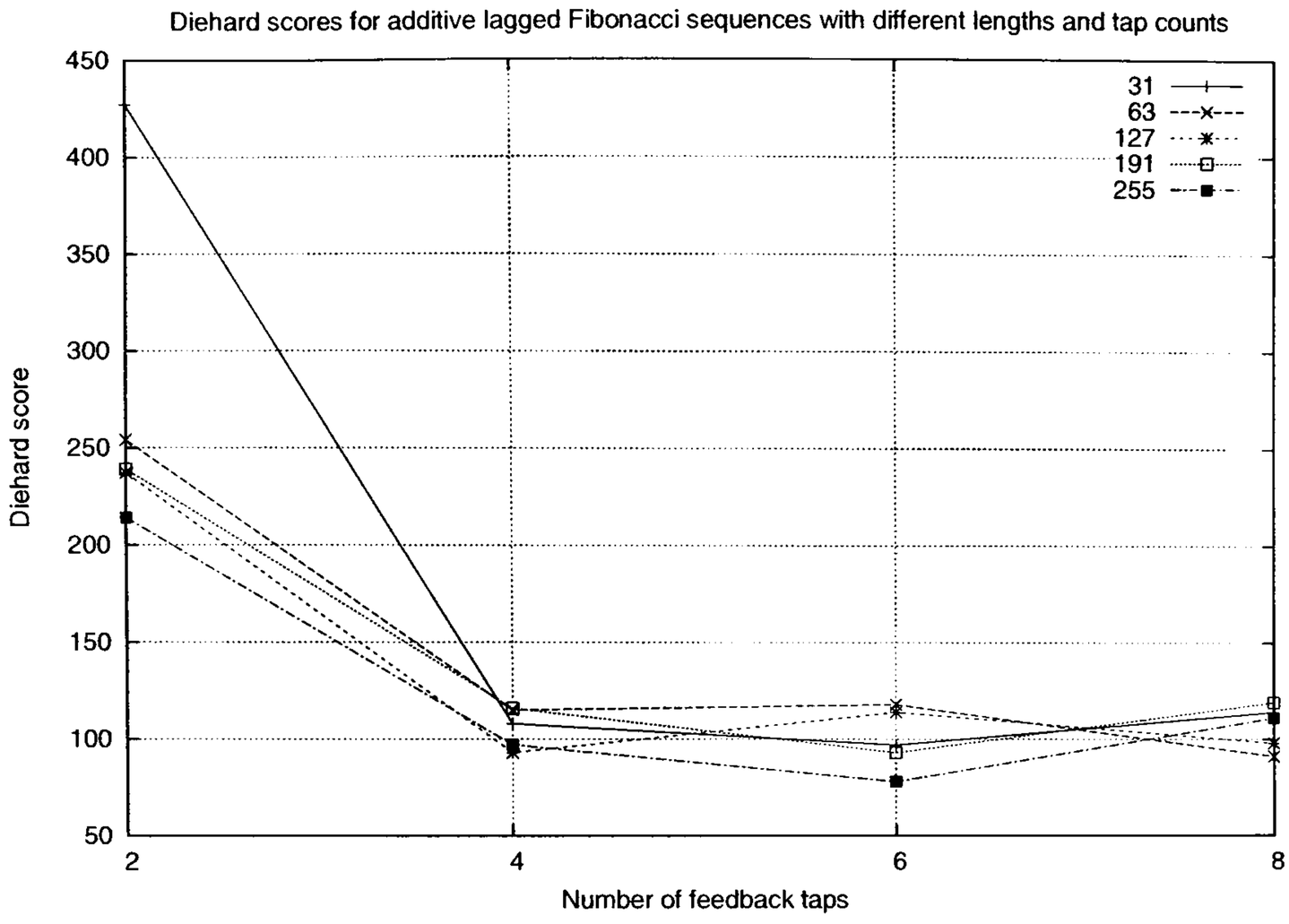


Figure 6.4: Diehard scores for additive lagged Fibonacci generator with different lengths and tap counts.

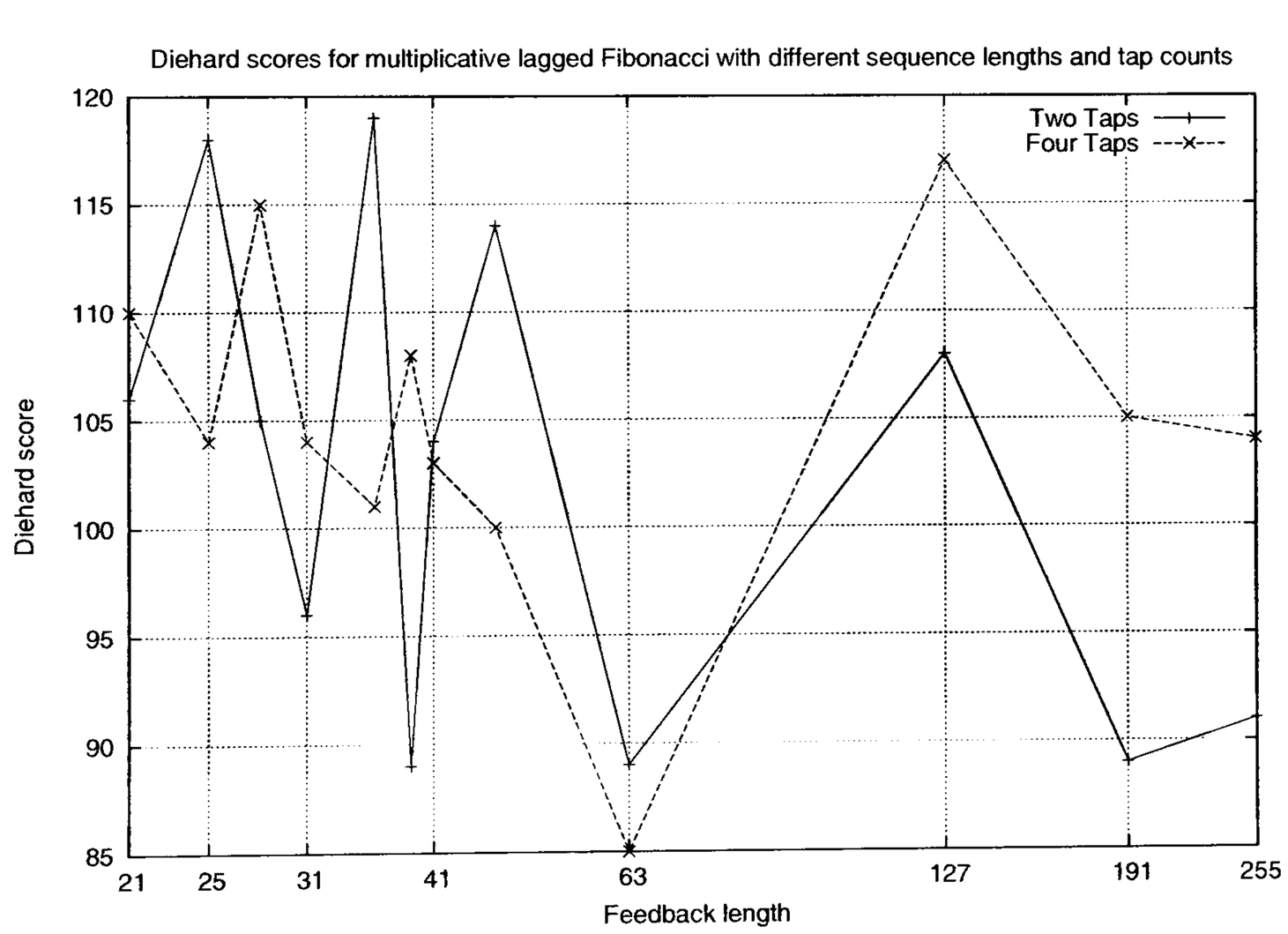


Figure 6.5: Diehard scores for multiplicative lagged Fibonacci generator with different lengths and tap counts.

Birthday test. However, if Crush is applied to a six tap version of the same generator then, while it does not pass completely, only two of the tests fail, and do not fail as catastrophically. In particular it now passes the new Birthday test. The same behaviour is also found for slightly larger feedback lengths, until eventually the four tap version starts to improve, and the six tap version starts passing the tests completely. This provides some empirical justification for the hypothesis that increasing the number of taps will always improve statistical quality.

6.3.4 Selecting Good Generator Parameters

The analysis presented in the previous sections provides five criteria for selecting lagged Fibonacci parameters:

- Multiplicative generators provide the best randomness for a given polynomial.
- Exclusive-or generators are an extremely poor choice due to the independence between different bit positions in the samples.
- Additive generators need at least four taps.
- Increasing the number of feedback stages increases random number quality.
- Increasing the number of feedback taps increases random number quality.

These criteria will now be used to create high quality random number generators for use in FPGAs. Three different design situations will be examined, targeting cases where three different resource types are available or restricted:

Multiplier based In some designs not all the DSP blocks will be needed, and in those cases it makes sense to use them in a multiplier based generator, reducing the amount of logic and storage needed.

Block RAM based If many block RAMs are available then an additive generator with a small number of taps can be selected, reducing the amount of overall logic needed, particularly as far less logic based FIFOs will be required.

Logic based If there are neither DSP or RAM blocks available then plain logic can be used to implement an additive generator with many taps, backed by a logic based FIFO.

A target for statistical quality is needed in order to determine if the selected parameters are adequate for long running simulations. Diehard, SmallCrush and Crush must all be passed. In addition, if the generator is to be suitable for long running and massively parallel simulations, it will need to pass BigCrush. BigCrush is the harshest standard battery currently in use, using almost 2^{38} samples, about 30,000 times the amount that Diehard uses.

The period of the generator must also be specified, which will be determined by putting an upper limit on the number of samples that may be consumed by a potential simulation. If it is assumed that each node in a simulation consumes samples at a rate of 1THz, there are one million nodes, and that the simulation runs for one year, this gives a total of 2^{75} samples consumed for each simulation run. Using the common rule of thumb that the period of the generator should be much larger than the square of the total number of samples used [119, 187], this mandates a minimum period of 2^{150}, and with an additional safety factor of 2^{16} this means that a lower limit of 2^{166} is required.

Multiplier based generator

Although multiplier based generators have performed well with two taps, in order to err on the side of caution a four tap solution will be used, requiring three modulo-2^{33} multipliers in a tree to create each new sample. The multipliers must be 33 bits wide in order to process the implicit one in the least-significant bit.

In both Virtex 2 and Stratix 2 devices the DSP blocks provide signed 18x18→36 bit multipliers [8, 247], so three DSP blocks are needed to implement each multiplier; each multiply is modulo-33, so the partial product of the most significant 18 bits does not need to be calculated. The partial products must then be added together using two 32 bit adders to form each result, before adding together the partial products using two 32 bit adders (the LSB will be zero). So the feedback stage requires 9 DSP blocks and six 32 bit adders.

If the target period of 2^{166} is chosen then a delay of 2^{166-31} is sufficient, due to the extra period conferred by the 32 bit wide words, so the FIFO must have a length of 135. Some examples of generators with primitive polynomials of length 135 with four taps are:

$$1 : (135, 7, 2, 1) \qquad 2 : (135, 11, 2, 1)$$
$$3 : (135, 116, 32, 16) \qquad 4 : (135, 82, 52, 16)$$

The first two polynomials are the first two primitive polynomials found while exhausting the entire search space for polynomials of degree 135 with five non-zero coefficients (the fifth weight is just 1 for x^0, and does not translate into a feedback tap). If good polynomials are searched for in this way then thousands of polynomials must be discarded before finding polynomials where the taps are spread out, so the search process was altered to only allow polynomials with taps more than a certain distance apart to be considered. Polynomial three is the first polynomial found where all taps are at least 16 places away from each other. The distances between the taps in the polynomial (including x^0 which represents the just generated sample) are spaces of 19, 84, 16 and 16. Ideally the distances between taps should all be different, so polynomial four was selected. In this case the tap distances are 53, 30, 36 and 16.

When BigCrush was executed on this generator one p-value outside the [0.01,0.09] range

was produced. The test failed is MultinomialBitsOver, with a value of 0.9971. This is a bit-oriented (rather than word-oriented) test, which forms t dimensional vectors from bits within a sample set, uses these vectors to fill a t dimensional hyper-cube, and then looks for a uniform distribution of points within the cube. On running the test again twice a good p-value was returned both times, so this generator is accepted.

The hardware characteristics of this generator in the Virtex 2 family are:

Slices	LUTs:Logic	LUTs:SRL16	LUTs:Routing	FFs	DSPs	Speed
517	98	353	0	515	9	155

Block RAM based generator

If block RAM can be used then the increased FIFO length should allow fewer feedback taps to be used, minimising the amount of extra logic required. A four tap generator was earlier found to be insufficient to pass Crush when a small FIFO length was used, but by using block RAMs the FIFO length can be of a comparable size to software. If a four tap generator is found to not pass BigCrush then a six tap generator can be tested.

The generator will be designed around the assumption that two block rams are available for use, and that these are using Virtex-2 sized block RAMs, allowing 512 x 32 bit words to be stored in each. In total this provides 1024 RAM based FIFO stages, in addition to any extra that are instantiated using logic, although logic based FIFOs will be kept to a minimum.

To avoid creating one extremely long FIFO span and three much shorter spans, the two block RAMs will be used as two FIFOs with a length of (at most) 512. This requires a polynomial with four taps, but with a maximum separation of 512 between three of the taps, and a separation less than 16 (the length of an SRL16) for the final two taps. A sequence length of 1015 was chosen, and the primitive polynomial (1015,1010,504,5), was found, which fits the criteria.

Running BigCrush on this polynomial produced three suspicious p-values outside the range [0.01,0.99]. The three tests are: MatrixRank, a test based on the linear independence of matrices constructed using the random sequence; RandomWalk, a test that use the samples to construct two dimensional random walks; and AutoCor, which looks at the auto-correlation between bits at different lags within an sub-string of bits. The three p-values are 0.0076, 0.9972 and 0.0058 respectively. These are slightly suspicious rather than damning, and on a second and third run all three tests produced acceptable p-values, so the generator was found to meet the design criteria.

The hardware characteristics of this generator in the Virtex 2 family are:

Slices	LUTs:Logic	LUTs:SRL16	LUTs:Routing	FFs		Speed
331	109	193	28	297	152	

The slice count is high, but still acceptable. Most of the area is still taken up by SRL16s, although it is not clear where they are used. The point where logic based shift registers are needed is to implement the short tap distances between the first and last pair of taps, which would require two sets of 32 SRL16s, although it is possible that some of shift registers were disrupted by the Handel-C compiler retiming optimiser.

Logic based generator

When no RAMs and no DPSs can be used the only choice is to create an additive generator with a short sequence length, but with lots of taps. As with the multiplicative generator a FIFO length of 135 is required, but because the additive generator is not as effective as the multiplier generator 8 taps will be used, so the logic used to calculate the next sample will require a tree of 15 32 bit adders.

A primitive polynomial of degree 135 with 8 taps must now be found, and as before the search process is modified slightly in order to quickly find an acceptable polynomial. The pattern of acceptable taps is organised so that the candidate polynomials have a different distance between every tap, and that the minimum distance between all taps is at least m. For an n degree polynomial the starting vector looks like this:

$$x^n + ... + x^{(4*m+21)} + x^{(3*m+10)}x^{(2*m+3)} + x^{(m+1)} + x^m + 1 \qquad (6.18)$$

Tne first polynomial found was (135,55,44,34,25,17,10,4), which passes BigCrush straight away, as there are no points outside the [0.01,0.09] range for any of the tests.

The hardware characteristics of this generator in the Virtex 2 family are:

Slices	LUTs:Logic	LUTs:SRL16	LUTs:Routing	FFs	Speed
1018	309	587	204	1201	140

Note that even though this generator and the DSP based generator have the same overall FIFO length, the number of SRL16s is much higher, 587 versus 353. This is because double the number of taps are used in this generator, so less of the shift registers can be used at their maximum length of 16, resulting in a need for more shift registers to implement the same total delay.

6.3.5 Evaluation

Table 6.6 shows the results of implementing these three generators in Handel-C for the Virtex-2 architecture. The same parametrised hardware design is used for all three results, and no generator-specific optimisations for speed are used. The designs are compiled using the Handel-C compiler, with retiming and architecture specific LUT packing enabled, then placed and routed using Xilinx tools using the default effort level.

	Period	Slices	LUTs	FFs	Rams	DSPs	Speed
Multiplier based	2^{166}	517	451	515	0	9	145
RAM based	2^{1046}	292	180	531	2	0	191
Logic based	2^{166}	687	477	853	0	0	143

Table 6.6: Comparison of resource usage and speed for lagged Fibonacci generator variants on the Virtex-2 architecture

	Slices(%)	RAMs(%)	DSPs(%)
Multiplier based	10.10	-	22.50
RAM based	6.46	5.00	-
Logic based	19.88	-	-
Xilinx LFSR	5.62	-	-

Table 6.7: Percentage of xc2v1000 resources used by each of the generator variants.

All the designs can run above 133MHz, which is generally the upper bound for clock speed on designs performing significant numerical processing such as simulations. The performance is not good enough for high speed communications applications, but in those cases a simpler bit-level generator is likely to be all that is needed. It is also highly likely that some generator specific optimisations could increase the clock speed significantly.

Table 6.7, summarises the resource usage of the three generators. This shows usage of the three main resource types as a percentage of the available resources in an xc2v1000 device. The attempt to tailor the generators to specific resource types has clearly worked, with each variant using a large amount of one specific resource.

Overall the block RAM based generator probably offers the best value, as it only requires two RAMs, the general logic use is not very high, and higher clock rates are achievable. By comparison the multiplier based generator uses a much higher percentage of its specific resource type, and while the logic based generator uses no special resources, it will take up a large area within a design. Both the multiplier and logic based generators are likely to slow down the compilation and place-and-route process due to the large amount of logic, although this affect could be mitigated with pre-placed cores. The non-RAM generators also require large numbers of shift-registers to be inferred and translated to SRL16s, which requires a significant amount of time in some compilers, although this too can be avoided with pre-compiled cores.

For comparison purposes, Table 6.7 also contains the equivalent area of a 32bit wide LFSR with period 2^{166}, implemented using the Xilinx core-generator. Because there are no primitive trinomials for this length, the default polynomial offered for this length actually has four taps, and so will be more random than most LFSRs, but this still fails Crush and Big-Crush on multiple tests.

The fact that the first choices for generator combinations all pass straight away suggests that there is scope for reducing the size of the generators: there exist six tap additive generators with 135 feedback length that also pass Big-Crush. However, this is falling in to the

Device	Inst. Count	Area% (Slices)	Speed (MHz)	Samples per sec
Virtex-II xc2v1000	1	6 %	191	191
	2	11 %	174	348
	4	25 %	180	720
	16	96 %	101	1616
Spartan-3 xc3s1000	1	4 %	155	155
	12	48 %	135	1620
Dual Athlon 2.66GHz MT19937	1	-	2660	52
	2	-	2660	99

Table 6.8: Area and speed for multiple instances of RAM based lagged Fibonacci generators on selected Virtex-II and Spartan-3 devices. Also included are generation speeds for a multi-threaded Mersenne Twister generator on a dual processor Pentium-4 2.66GHz.

trap of reducing the quality of the generator until it can *just* pass the current test battery. To be able to pass the next battery of tests the generators must be able to easily pass the current tests.

In Table 6.8 the performance of multiple generators within a single device is presented, showing the decrease in speed and increase in overall sample rate as the parallel instance count is increased. Only RAM based generators are shown, as these allow the greatest degree of replication in the two chosen devices. The first group of results shows results for the xc2v1000 Virtex-II device, while the second is for the low cost xc3s1000 Spartan-3 device. In the xc2v1000 the number of instances is limited to 16 by available logic (there are still 8 spare RAMs leftover), while in the xc3s1000 the limit of 12 is due to a lack of block RAMs. Because the xc2v1000 device is almost full with 16 instances the speed has dropped significantly, so the xc3s1000 actually offers the same overall sample rate, while still having half its logic and all its DSPs still available.

The table also shows performance for the Mersenne Twister [148] software generator. Because random number generators are often used in multi-threaded simulations the performance for 1 and 2 concurrent threads is shown. The tests were performed on a dual Athlon 2.66GHz computer with 1GB of RAM. The sample rate of a single thread is almost one third that of a single xc2v1000 based generator, while about half that of a single xc3s1000 generator. If the fully utilised FPGAs are compared against the fully utilised computer (i.e. two software threads are used), the FPGAs both provide over 16 times the sample generation rate of the software implementation.

One point that has not been discussed is that many applications require less than 32 bits per sample. The techniques discussed here, including the three high quality generators, can all be scaled to smaller widths. There will be a loss in overall period: every bit removed halves the period, so for widths less than 16 it would be a good idea to pick larger polynomials to compensate, particularly with additive feedback. Some of the resource specific optimisations will also become less effective, though at a width of 16 it becomes possible to

use half the number of DSP blocks, or to use block RAMs in a narrower configuration.

The uniform generators presented here have been made available to software applications through the IGOL interface, and so can be used by any software applications. The main API supported by IGOL for random number generation is the GSL (GNU Scientific Library) interface, which allows external random number generators to be exposed to compatible applications. Through this interface applications can use hardware uniform random number generators implemented using the RC1000, RC2000 or UltraSonic. For all these platforms the speed at which random numbers can be generated in hardware far exceeds the rate at which they can be transferred back to the host: the maximum theoretical bandwidth of the PCI bus limits the RC1000 to just 30MSamples/s, while the RC2000 and UltraSonic are limited to 60MSamples/s.

When used through the GSL interface the speed of random number generation is further reduced due to function call overheads, with a sample rate of 41MSample/s achieved when an RC2000 is accessed from a software application running on an Athlon 2.66GHz. However, the GSL software generator also suffers from the same overheads, and even though a Mersenne Twister provides a 52MSample/s processing rate when used as a standalone generator, it can only provide 31MSamples/s if accessed through GSL. This means that the speedup when using hardware random number generation is only 1.3 times. However, the hardware random number generator uses only around 10% CPU time when it generates numbers at this rate, while the software generator must use 100%. This leads to a greater speedup in application processing rate than the sample generation rate suggests: in an application that uses a monte-carlo method to calculate Π a speedup of 2.3 times was achieved.

6.4 Summary

In this chapter the generation of high quality random numbers for simulations has been explored, and the ability of traditional uniform random number generators to meet these standards has been tested. Traditional Linear Feedback Shift Registers have been shown to produce poor quality sequences, which are inappropriate for use in long running simulations due to the many statistical flaws and correlations. In order to meet the need for high quality random numbers in hardware the Lagged Fibonacci generator is chosen, and three different parametrised generators are developed.

The key features of these hardware Lagged Fibonacci generators are:

- A period more than long enough for the most demanding of simulations.
- Statistical quality that passes the most stringent statistical batteries.
- Three different parametrisations, allowing generators to use three different types of resources.

- A raw generation rate over 16 times that of a dual Athlon 2.66GHz, even when a low cost Spartan-3 part is used.

The generators have also been exposed to software applications through the IGOL GNU Scientific Library plug-in, allowing existing software applications to use the hardware random number generators.

Chapter 7

Approximating Arbitrary Probability Distributions

Many algorithms and applications require random variates taken from non-uniform probability distributions. Applications include: Ray Tracing, where BRDFs (Bidirectional Reflectance Distribution Function) must be approximated for importance sampling [50, 116]; financial computing, such as calculating the prices of financial instruments [19, 251]; queuing simulations of communication networks and manufacturing processes [36, 188]; and Monte-Carlo evaluation of functions [156]. These applications all rely on a fast, efficient, and accurate supply of random samples. If these applications are to be implemented in hardware it is necessary to adapt existing variate generation algorithms from software where appropriate, and to develop new algorithms when they are not.

In this chapter a technique for the efficient hardware generation of random variates from any distribution is described. The generation method consists of two parts: a hardware template for a simple yet powerful random variate sampling unit based on a composite distribution approximation; and a process for generating a composite approximation for any target probability distribution. Unlike other FPGA based methods for sampling probability distributions, this technique is completely general, and does not require alterations to design logic or run-time reconfiguration in order to change the output distribution. The method also has other advantages, such as the ability to select between a set of different distributions in each cycle, morphing of distributions at run-time, and support for generating antithetic variates.

The chief contributions are:

- A probability distribution independent hardware algorithm for sampling non-uniform distributions. The hardware realisation of the algorithm specifically addresses the strengths and weaknesses of FPGAs to allow an area efficient, high-speed implementation.

- A process for creating composite distribution approximations to be used in the hardware algorithm. This process allows exact measurement of approximation error, and can approximate any target distribution, whether analytically or empirically derived.
- Demonstration of the speed, area efficiency and accuracy of the method compared to other random number generation techniques, and discussion of the benefits and limitations of this technique.

In this chapter these three contributions are presented in order, starting with the hardware algorithm.

7.1 Hardware Algorithm

In this section the generator algorithm is explained, followed by an explanation of the hardware implementation of this algorithm and the options available, along with some of the advantages of the technique.

The proposed hardware distribution generator operates by taking multiple random variates with easy to generate distributions, and then combining them using a simple operation to provide a composite distribution. By selecting the component distributions and the combination method correctly, this composite distribution can be made to approximate a given target distribution. The quality of the approximation depends on factors such as the smoothness of the target distribution, the number and shape of the component distributions, and the way that they are combined.

While the combination method could in principle be any operator, for efficient implementation only selection will be considered here, i.e. each value produced by the composite distribution will be an untransformed sample taken from one of the component distributions. To control the selection process an independent random variate is used, making the selection stateless and independent of the component distributions.

Each generator uses a vector of n component random variates, $RC_1 ... RC_n$, with (as yet) unspecified distributions, and a selection random variate, RS, with a range of $[1, n]$. The value of the composite random variate C at step i, along with its overall PDF and CDF, can then be defined as:

$$C_i = RC_{k,i}, \text{ where } k = RS_i \tag{7.1}$$

$$P(C = x) = \sum_{k=1}^{n} P(RS = k) \times P(RC_k = x) \tag{7.2}$$

$$P(C < x) = \sum_{k=1}^{n} P(RS = k) \times P(RC_k < x) \tag{7.3}$$

Because selection is used to combine the distributions, only one component distribution needs to be sampled to produce each output sample from the composite distribution. This means that the vector of component variates can be generated by using a single uniform random variate, UC, and selecting from a vector of transformation functions, $FC_1 ... FC_n$,

chosen such that $RC_x \sim FC_x(UC)$. In the same way the selection variate RS can be split into a uniform variate, US, and a transformation function FS. Initially it will be assumed that each component distribution is identically weighted, and FS is reduced to the identity function. So the ith value generated by C is:

$$C_i = FS_{us}(uc), \text{where } us = US_i, uc = UC_i \tag{7.4}$$

7.1.1 Hardware Architecture

The motivation for transforming the task of sampling one distribution into that of selection and sampling from amongst many component distributions is that this allows a very area efficient hardware implementation. The component distributions can be chosen such that they are implemented purely in terms of adds and bitwise logic, with the selection reduced to memory lookups. The hardware implementation of this algorithm does not require any complex logic, and can be fully pipelined, providing high clock rates.

As input to the composite distribution generator two uniform random sources US and UC are needed. A problem when generating uniform random numbers in hardware is that it is very easy to implement uniform random number generators which are distributed $\sim U[0, 2^w)$, but it is difficult to generate any other range. In software a $U[0, 2^w)$ variate can be transformed into a $U[0, t)$ variate (where $t < 2^w$) by using an iterative rejection method, but in pipelined hardware this is not a viable solution as it would have an unpredictable throughput: in some cycles a random variate would not be output, introducing bubbles into later pipeline stages. Using a modulus operation to reduce the range will take large amounts of logic, a lookup table uses large amounts of memory, and both will skew the transformed variate away from true uniform. The only efficient conversion of $U[0, 2^w)$ variates available is to $U[0, 2^{w-k})$ variates, using a bitwise-and with $2^{w-k} - 1$.

A consequence of this range limitation is that n (the number of component distributions) must be a binary power. Similarly all the functions $FC_1 ... FC_n$ must use a $U[0, 2^w)$ variate as input. This an unfortunate restriction, but by embracing the limitations a very simple and area efficient generator can be created. Each FC function will apply a bitwise mask to UC, adjusting the variance, then add an offset, altering the mean. This allows any distribution of the form $U[a, a + 2^{w-k})$ to be used for the component distributions, by applying the transformation $FC(x) = fc_{offset} + (x \otimes fc_{mask})$ (where $\otimes$ indicates bitwise-and). The (fc_{offset}, fc_{mask}) pairs can be stored in a ROM or RAM lookup table. The stored offset value will need to be the same width as the output width, but space can be saved in the table by storing just the width of the mask, then expanding it before use. The required element width of the table is then $rw + \lceil \ln_2 mw \rceil$, where rw is the width of the offset (and generator output value), which limits the range of values that the composite can produce; and mw is the width of the mask, which limits the maximum variance of component distributions (and determines the number of random bits that UC must supply).

Figure 7.1 shows the hardware architecture needed to approximate a target distribution using a composite of rectangular component distributions. The six components are:

1. A $U[0, n)$ uniform variate source used to select which of the n component distributions will be sampled in each cycle.

2. A lookup table containing the n $(fc_{offset}, fc_{width})$ pairs which control the transformation used to sample from a rectangular distribution with the desired mean and variance.

3. An expander which takes fc_{width} and expands it to the right number of ones, producing fc_{mask}.

4. A $U[0, 2^{mw})$ uniform variate source used to provide uc, the value used to generate a sample from within the chosen target distribution.

5. A bitwise-and between the expanded mask and uc, transforming the variate from $U[0, 2^{mw})$ to $U[0, 2^{fc_{width}})$.

6. An adder which adds the mean adjustment constant fc_{offset} to the variance adjusted variate, transforming it to a $U[fc_{offset}, fc_{offset} + 2^{fc_{width}})$ variate.

The sequence of values produced by this sampling process is distributed according to C, and assuming that the selection of component distributions is correct, will have the required target distribution. Few distributions can be exactly approximated with this method, as most distributions cannot be tiled exactly with a finite number of rectangles. However, by using a large number of rectangles the distribution of the composite will approach that of the target distribution to within some margin of error.

7.1.2 Alternate Component Distributions

Rectangular distributions allow a very space efficient implementation to be used, but have the disadvantage that approximations to continuous PDFs will actually consist of many small up and down steps. With n components an approximation can have at most $2n$ steps, so when the width of the output value is large relative to n it will be necessary to approximate gentle probability gradients with large areas of constant probability followed by abrupt jumps up and down. To solve this problem the composite must be formed from component distributions with sloping sides rather than abrupt steps.

One option is to use symmetric triangular distributions. This distribution can be generated by taking two uniform variates and adding them together. Give two continuous uniform

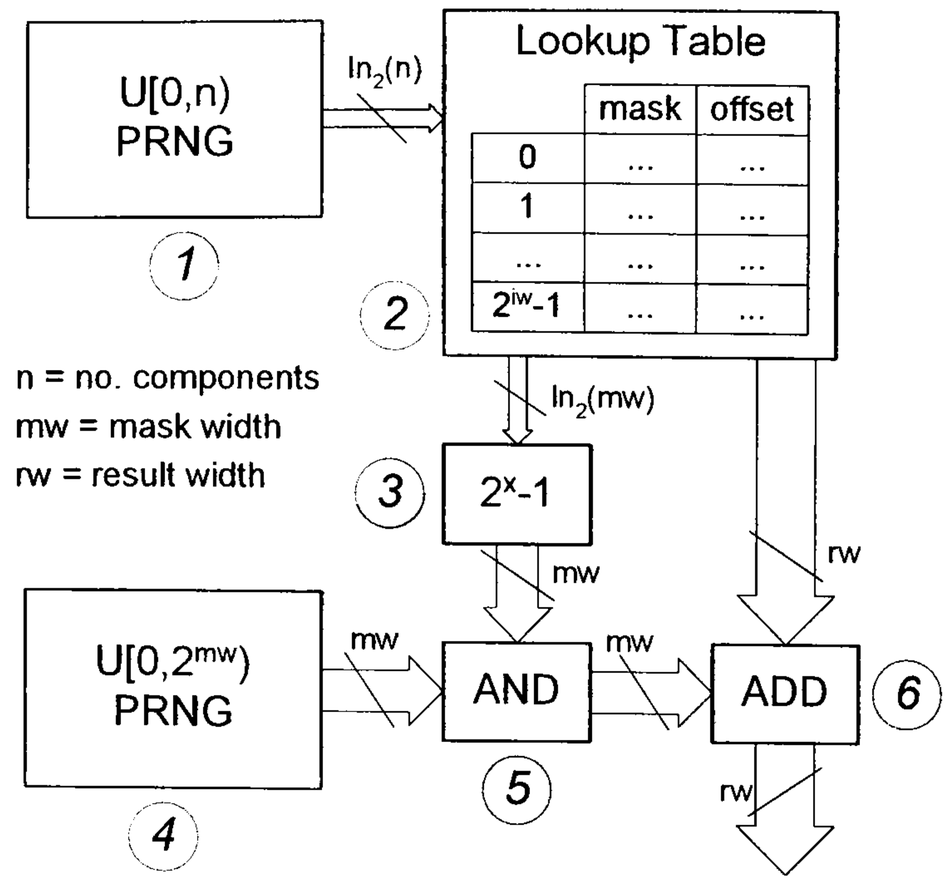


Figure 7.1: Architecture of hardware distribution generator

variates $X, Y \sim U(0, a)$ the PDF and CDF of $T = X + Y$ will be:

$$P(T = x) = \begin{cases} x/a^2 & \text{if } 0 \leq x \leq a \\ (2a - x)/a^2 & \text{if } a < x < 2a \end{cases} \tag{7.5}$$

$$P(T \leq x) = \begin{cases} x^2/2a^2 & \text{if } 0 \leq x \leq a \\ 1 - (2a - x)^2/2a^2 & \text{if } a < x < 2a \end{cases} \tag{7.6}$$

(7.7)

In hardware this means that if two a bit wide uniform variates are added together then the resulting distribution will be a discrete triangular distribution ranging from 0 to $2a - 1$, with a single mode at a. Bit masking can again be used to adjust the variance of the rectangular distributions, by masking both of the uniform variates. For w uniform random input bits a range of triangular distributions with widths from 1 up to $(w+1)/2$ can be generated. Using triangular distributions requires double the number of random bits and an adder, but allows smooth slopes in the output PDF.

An advantage of triangular elements is that they scale up to higher output bit widths. This means that if a gentle gradient is approximated at a chosen output width then the triangles should still provide a smooth slope if the output range and triangle widths are doubled. This is not always the case, for example the peaks of PDFs will resolve into a row of tiny points as the range resolution is increased, but over most areas of the approximated distribution the slopes in the PDF will remain slopes at higher output resolutions.

Unfortunately the triangular distribution, while supporting slopes, cannot produce the abrupt vertical PDF steps available through rectangular components. The closest that can be achieved is a degenerate triangle with no width, but this will have a very large height, and it

is impossible to build up a slope on just one side of it: any attempt to place a smooth slope on one side of the step triangle will result in part of the slope "leaking", poking through the desired step as well as building up the slope.

A possible solution is to transform the symmetric triangular distribution into an asymmetric right-angled "wedge" triangle. For example:

$$T' = \begin{cases} a - T & \text{if } T < a \\ T - a + 1 & \text{if } T \geq a \end{cases} \tag{7.8}$$

will create a wedge that decreases from left to right, while the additive inverse will produce its reflection. Unfortunately this requires many extra operations, requiring at least two more adders and control circuits to determine when to flip the distribution horizontally.

An area efficient way of allowing both slopes and steps is to generalise the technique used to generate triangular distributions by adding two different uniform distributions together. Given two continuous uniform variates $X \sim U(0, a)$ and $Y \sim U(0, b)$ where $a \leq b$ then the PDF and CDF of $M = X + Y$ will be:

$$P(M = x) = \begin{cases} x/ba^2 & \text{if } 0 \leq x < a \\ 1/b & \text{if } a \leq x < b \\ \frac{(a-x)}{ba^2} & \text{if } b \leq x < a + b \end{cases} \tag{7.9}$$

$$P(M \leq x) = \begin{cases} x^2/2ab & \text{if } 0 \leq x < a \\ \frac{a^2}{2ab} + \frac{(x-a)}{b} & \text{if } a \leq x < b \\ 1 - \frac{(2a-x)^2}{2ab} & \text{if } b \leq x < a + b \end{cases} \tag{7.10}$$

(7.11)

The two parameters a and b describe a family of trapezoids, or truncated pyramids. This is shown in Fig. 7.2. The larger of the two parameters, here called the major width, controls the overall variance of the distribution, and indirectly the height. The smaller parameter, the minor width, controls the "pointiness" of the distribution, allowing both rectangular and triangular distributions to be used, as well as a range of intermediate shapes.

There is very little extra hardware cost for supporting trapezoid distributions rather than just symmetric triangular distributions. The main change is that the lookup table containing the mask and offsets now needs to hold two masks, increasing the width of each element to $rw + 2\lceil \ln_2 mw \rceil$. This also means that two independent mask expanders are now needed. Apart from these two changes no extra hardware is required, and steps and slopes are both now available.

7.1.3 Algorithm Extensions

When element tables are stored in on-chip RAM blocks the dual access ports of most such RAMs[8, 237] allows certain advantages and improvements. The first advantage is that

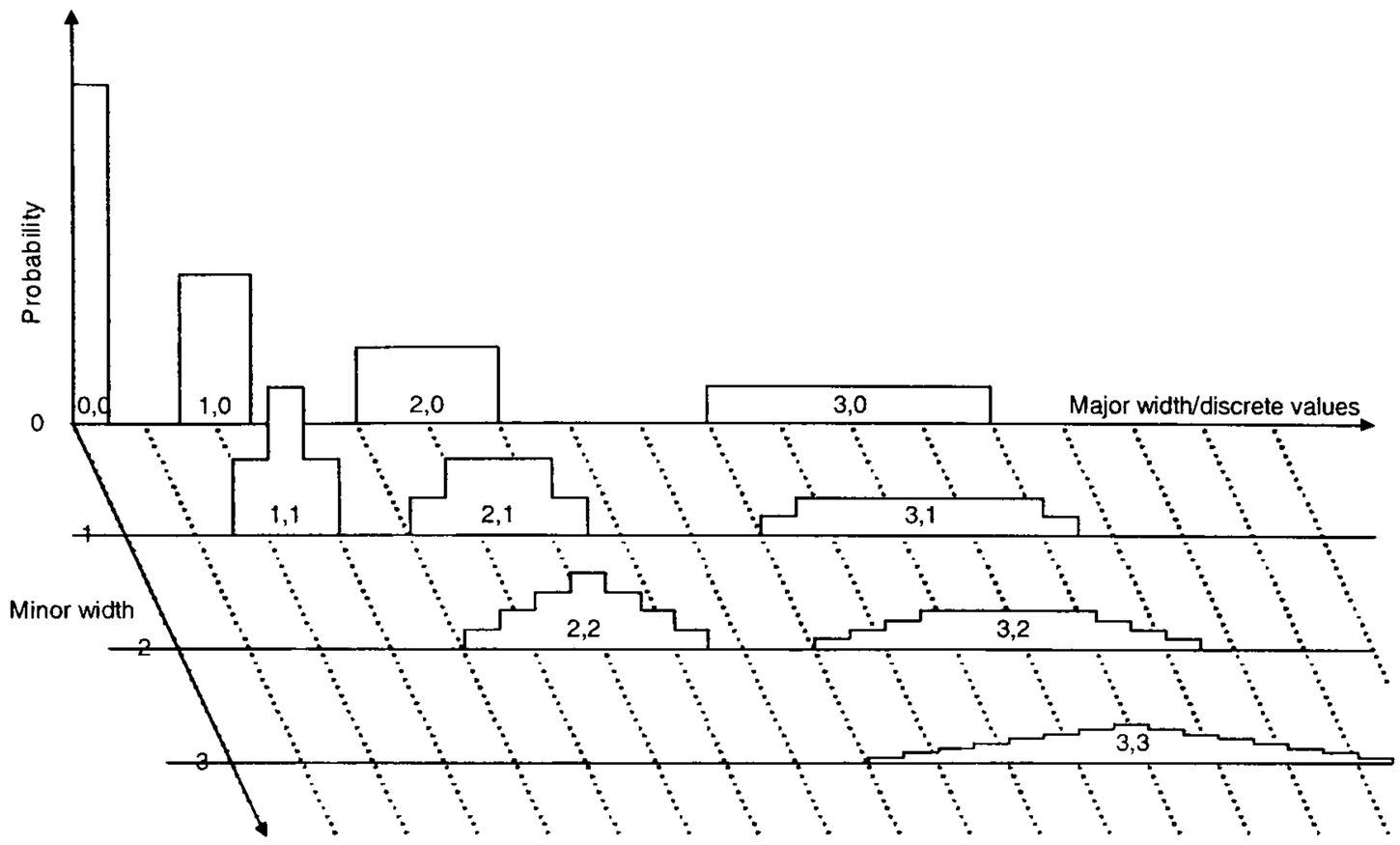


Figure 7.2: Families of trapezoid component distributions

each element lookup table can be shared between two distribution approximators, halving the number of block RAMs needed per unit. The two units can share the same table, generating the same distribution, or can operate out of different regions of the RAM to provide alternate distributions. If two units share the same element table then it is also possible to create pseudo-antithetic variables, which are important for variance reduction in Monte-Carlo simulations.

Variance reduction is a way of reducing the number of simulation runs needed to achieve an accurate result, by introducing controlled correlations between random variables used in the simulation [118]. Antithetic variables are one example of this technique [44, 248], reducing variance in the answer calculated through simulation by producing correlated pairs of random numbers to be used at the same step in different simulation runs. The paired simulation runs will tend to cancel each other out (without affecting the accuracy of the answer), because a large movement in one will be matched by a similar movement in the other direction. Antithetic variables are easily produced when a distribution is sampled by transforming a uniform variate: if samples are produced by passing a uniform number $X \sim U(0,1)$ through a transformation function F then a pair of antithetic variables can be produced by taking $V_1 = F(X)$ and $V_2 = F(1-X)$.

Antithetic variates can be produced in a hardware composite by having two generator units sharing a common element lookup table. If the elements in the common table are sorted by offset, then elements at low indexes will produce values lower in the output range, and elements at high indexes will produce values higher in the range. The exact values produced will depend on the shape of the elements as well as their offset, but overall the lookup table can be seen as an approximation to function F above, while X is the uniform

variable used to provide the index. If both units share the same index variable, with one using X and one $1 - X$, then the two units will produce pseudo-antithetic variates. The strength of the correlation will depend on the exact nature of the element table, but should provide at least some variance reduction in most applications.

Another way that table driven approximations can be used is to provide data dependent distributions. By maintaining distinct areas within the element lookup table a different distribution can be sampled on each cycle. For example if a 2^k sized table is split into 4 sections then two of the table index bits can be used to select between distributions with no extra hardware or computational overhead. These distributions might be unrelated, or the index can be used as a run-time distribution parameter. For example CCD (Charge Coupled Device) pixel noise is partially dependent on light level [84], so when modelling the effects of this noise the value of incoming pixels can be used to affect the noise applied to them.

Another feature of the generator architecture is that the distribution can be changed at run-time by altering the constants stored in the table. This can be done even while the generator is running, either by dedicating one of the ports in a dual-ported table RAM to modifications, or by modifying the generator circuitry so that when a new constant is written into the RAM the generator will use that value for the current output cycle rather than reading from the table. As the table entries must be altered one-by-one the distribution produced will morph from the original distribution to the new one, and by carefully selecting the order in which the elements are changed the intermediate distributions can be made to produce smooth transitions. This feature can be used to make time-varying distributions, for example by gradually replacing coefficients in the table from a larger external RAM bank or storage.

For applications where a sharp transition between distributions is needed the element table can be double-buffered, with constants loaded serially into an alternate RAM. Once the constants are loaded the generator can flip between the RAMs, changing the distribution in a single cycle. Using dual-ported RAM this can be achieved even with a single RAM by partitioning the table into two parts.

The simplicity of the algorithm and its hardware implementation allows generators to be both small and fast, but in order to be used a means of organising the component distributions so that they form an approximation to the target distribution is needed. In the next section the method developed for creating the approximations is explained.

7.2 Approximation of Target Distributions

Generating the data tables used to control the composite generators is a critical process, as without a method for tiling the target PDF with the set of component distributions the hardware generator architecture is useless. In this section the methods used to create the data tables for generators are introduced, and some features of a typical approximated PDF are shown. In the following text it is most useful to think about the composite and component

distributions graphically, with the composite distribution having area one, and containing n component distributions with area (and so selection probability) $1/n$.

7.2.1 Approximation Algorithm

In the generator hardware architecture the components occur with fixed probability and the number of components to be used is fixed. If a component's distribution has the right width but the wrong height it cannot be stretched vertically to improve the fit: the only way of reducing the height of the component is to increase the width. The area (or total probability), of each element must be preserved. Equally, if a good approximation has been reached using $2^n - 1$ elements it is impossible to stop at that point, as there is no efficient way of wrapping the index round at any number of elements other than 2^n. The only way of achieving a good fit is by having large amounts of overlap between elements, stacking them on top of each other where needed to achieve height. Creating any kind of analytical solution to the problem appears impossible: perhaps for very simple target distributions a solution is possible, but attempting to find an solution for the optimal approximation of an arbitrary distribution using exactly 2^n rectangles is computationally infeasible.

For an approximation over an r bit range the number of possible configurations for a rectangular element with widths varying between 2^r and 2^0 is:

$$\sum_{i=0}^{i=r} 2^r - 2^i + 1 \approx r2^r \tag{7.12}$$

When the width of the rectangle is large the number of possible configurations is lower, as the rectangle can only shift a certain amount within the generator range without exceeding the range limits, but as the rectangle becomes narrower and taller the number of possible offsets increases. If n elements need to be positioned, then the number of possible configurations for all elements is $\approx r2^r n!$. With heuristics this number could be reduced, but it is still too large to brute force for useful values of r and n, particularly as the evaluation of each configuration is non-trivial.

A simple simulated annealing [165] approach was the first method for producing generation configurations that was tried, but this did not work: even when the system temperature was lowered very slowly, the solutions contained many sub-optimal features that were obvious through visual inspection of the resulting graphs.

Instead a simpler strategy was devised that produces acceptable results in a acceptable amount of time, based on either an automatic or semi-interactive process. Rather than trying to incorporate a method for avoiding local minima into the search algorithm, the strategy is simply to select a single element within the composite, then try to reposition that element to provide a lower overall error. If the error is lower the new composite is selected, otherwise the change is discarded. So the overall search strategy is to select any candidate configuration that has a lower error than the current configuration, regardless of whether this results

in getting trapped in a local minima.

The element to be repositioned is selected using one of three processes:

- **Worst Element** For each element the approximation error is calculated if that element is removed. The element which causes the largest reduction in error due to its absence is repositioned to minimise overall error. If no better configuration is found then the next worst element is considered, and so on. This is particularly effective during the initial stages of curve fitting.

- **Worst Point** The contribution of each point in the range to the error metric is calculated, then the points are sorted by decreasing error. All the elements which contribute some probability to the worst point are then repositioned from largest contributor to smallest contributor. If no better configuration is found then the second worst point is considered, and so on. This can be useful when trying to improve the tails of distributions.

- **Random Element** An element is selected at random, then repositioned to minimise overall error. This is a good polishing technique when approximation quality is critical, but it is very slow when large numbers of elements are used: during the final stages of approximation only a small set of elements can be moved into a better position, so most of the time the randomly selected element cannot be moved.

This very simple strategy works well in practise: even without any human input the best approximation found will be much better than that provided by the simple simulated annealing or that a human could create by hand. Sometimes the solution found will be good enough by itself, but if the approximation is not close enough to meet requirements then the human operator can perturb a small set of elements, or change the search strategy, forcing the algorithm to start re-optimising the curve. Alternatively, the system itself can attempt to determine when the search process has stalled and take action, based on heuristics and adjustable thresholds.

Figure 7.3 shows the application used to implement this process. First the user selects a target distribution, either from a set of predefined distributions or by loading an analytically or empirically derived distribution from a file. Then the parameters of the composite approximation are set, such as type and number of elements, and the width of the range. The application will then start the search algorithm, updating the visual display of the graph as the solution is found. During this process the user can guide the process by modifying the error metric or altering the element selection strategy. Once an acceptable solution has been found the approximation can be tested against the target PDF for goodness-of-fit, or exported as a Handel-C description for a random number generation core.

The application supports a large number of error metrics which can be changed at runtime to alter the search process. The two most effective have been found to be sum of relative

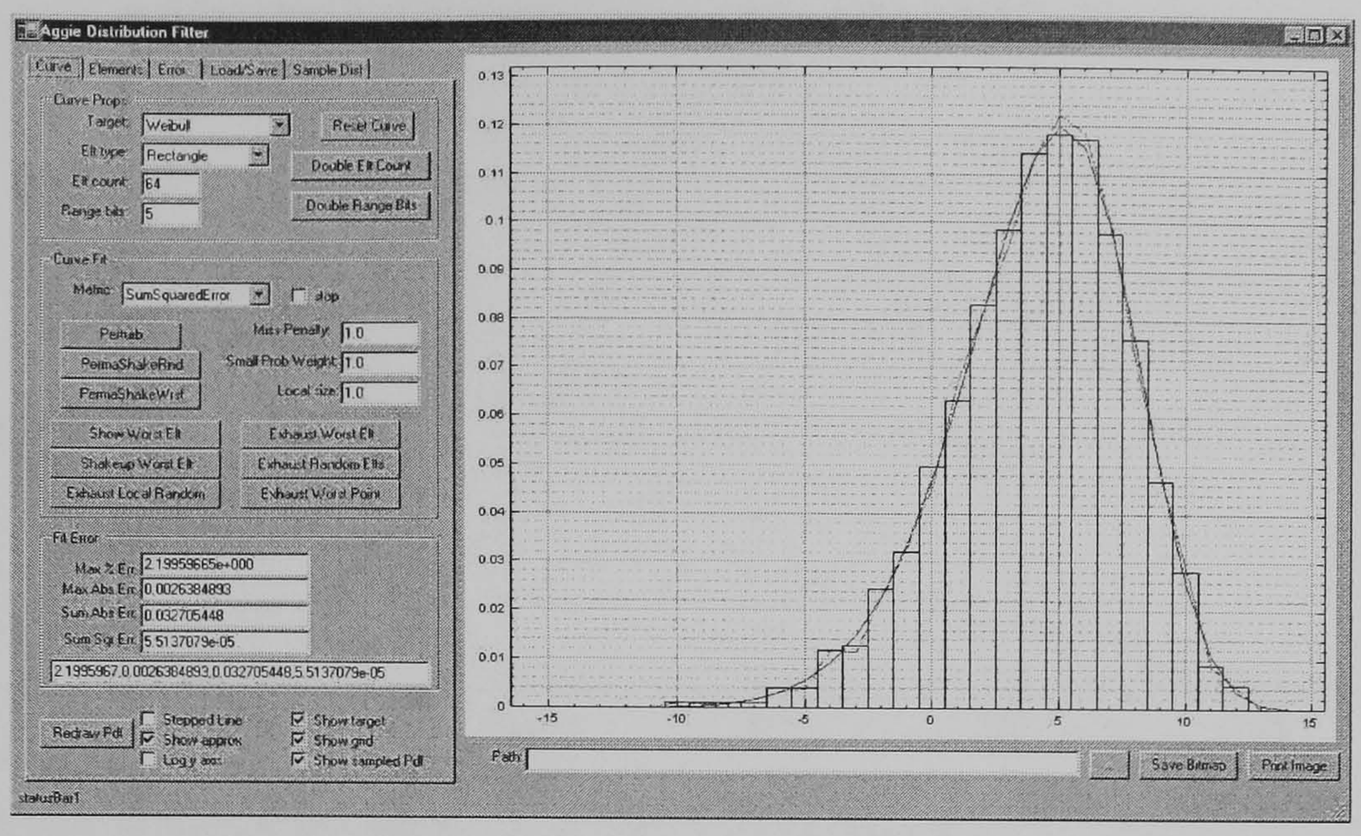


Figure 7.3: Interactive distribution approximation application.

error and sum of cubed error, respectively:

$$\sum_i \frac{|o_i - e_i|}{e_i} \qquad\qquad \sum_i |o_i - e_i|^3 \tag{7.13}$$

where o is the array of approximated PDF probabilities, while e is the equivalent array of probabilities from the target PDF. The sum of relative error has been found to be very effective at approximating the tails of distributions, but if very few elements are used will often neglect the peaks. The sum of cubed error behaves in the opposite way, and will concentrate on the largest absolute errors, often found near the distribution peaks, but will often ignore the tails once they become close to zero probability.

Figure 7.4 shows an approximation to the Gaussian distribution, using 1024 rectangles over an 8 bit range. The Gaussian variance is scaled so that all the discrete values in the range correspond to an integer, so 32 units on the scale represents one standard-deviation (σ). Once in hardware the integers will be interpreted as fixed point numbers. The stepped line shows the exact probability of each output value under the approximation, while the smooth line shows the target distribution probability. The target distribution is quantised into the same output values as the approximation, but it is shown as a continuous PDF to make the two distinguishable in the graph.

Figure 7.5 zooms in on the peak of the curve, and also shows the curve of the target PDF. The slight irregularities in the curve are now visible as uneven steps. Also note that the distribution is not perfectly symmetric, which could be critical for some applications. Constraints such as symmetry or perfect mean/variance can be applied in the search process if these properties are needed (for example, elements can be displaced as symmetric pairs).

In Figure 7.6 a close-up of the tail of the distribution is shown. This demonstrates one of

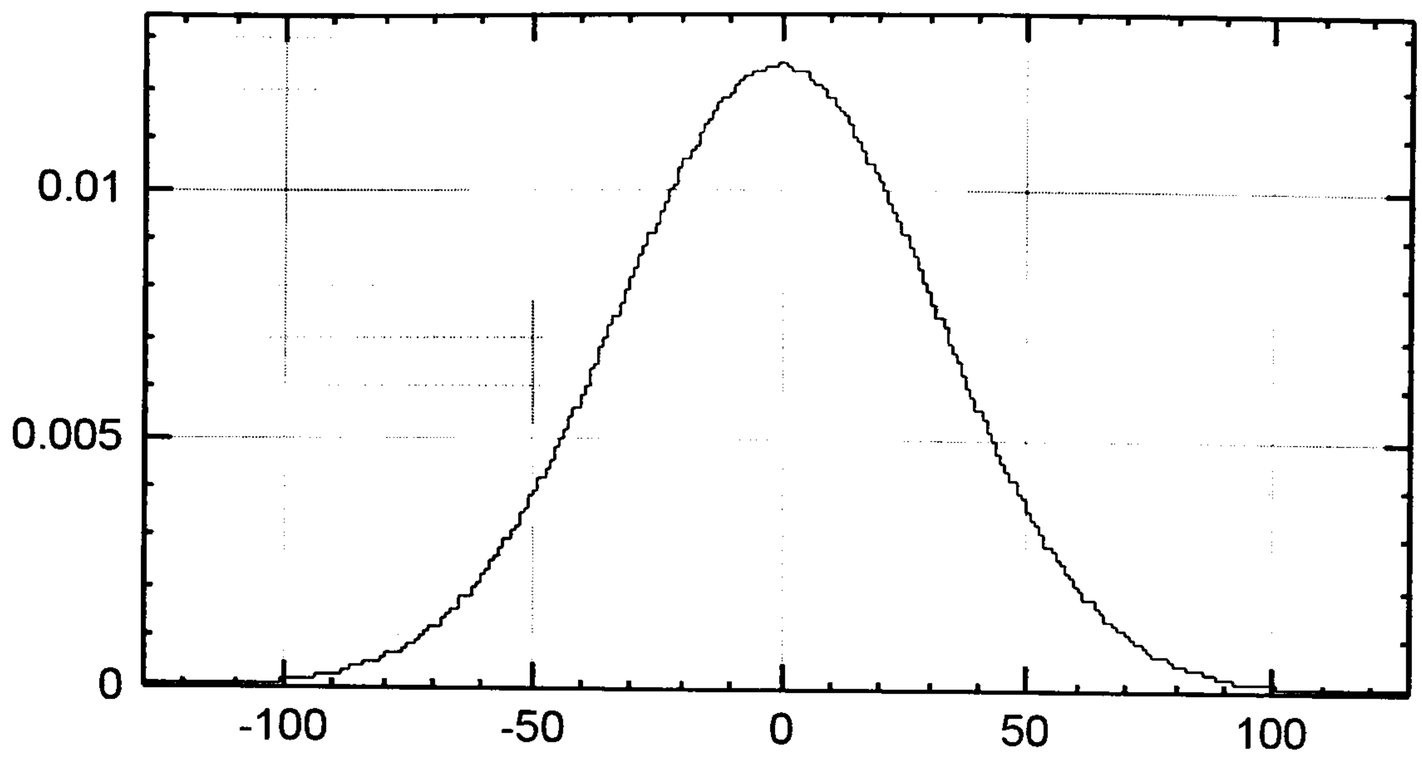


Figure 7.4: PDF of approximation to a Gaussian distribution over $\pm 4\sigma$ using 1024 rectangles with an 8 bit range.

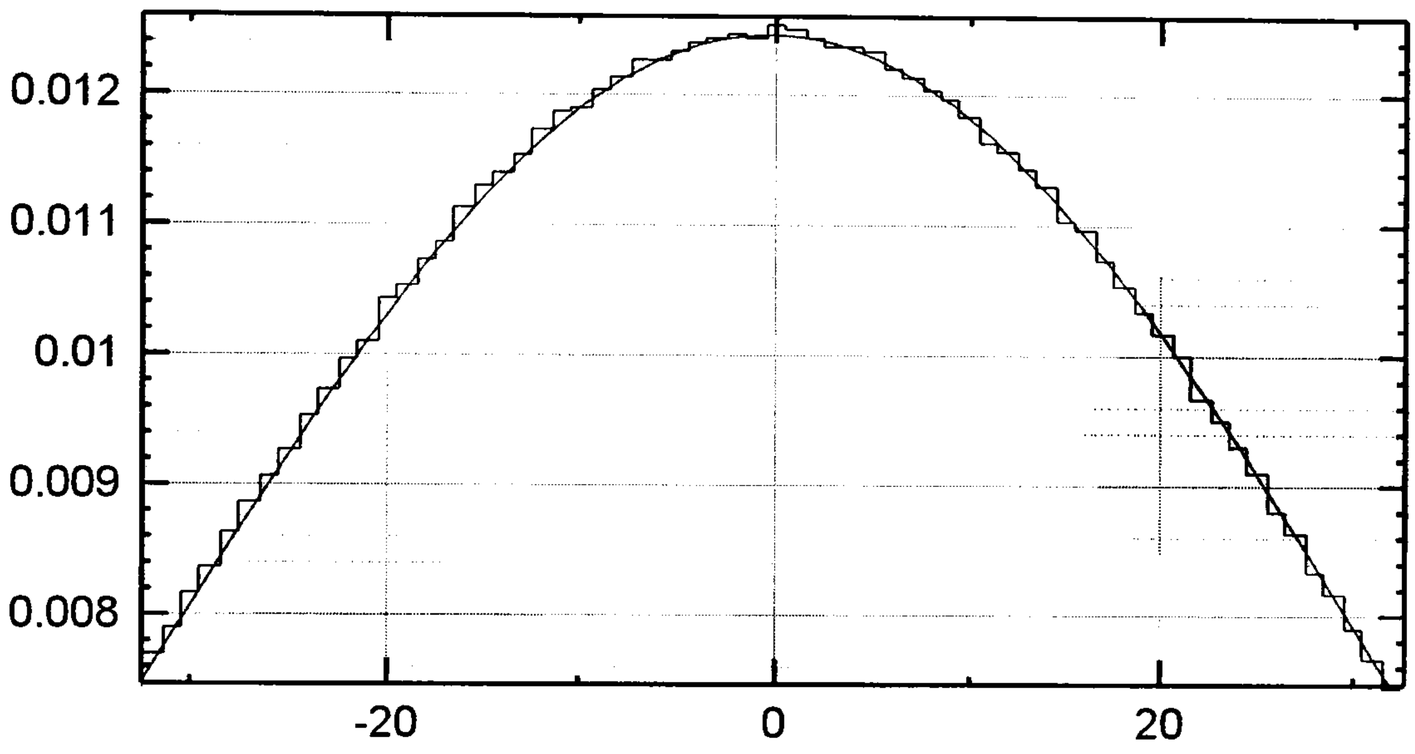


Figure 7.5: Zoomed section of peak of PDF from Fig. 7.4 over $\pm 1\sigma$.

the problems with this techniques when attempting to approximate low probability values, as eventually the element shape with smallest possible height is reached. At this point the relative error of the approximation starts to increase sharply, as the approximation error approaches the target value. The approximated tail can be extended out further by increasing the number of elements, which reduces the height of all elements, allowing thinner shapes to be used. Unfortunately doubling the number of elements only halves the minimum probability that can be produced. Between 4σ and 5σ the Gaussian probability drops by two orders of magnitude, so it would require roughly four times as much RAM to extend the tail by one σ. One solution to this problem is to use non-uniform probabilities on elements.

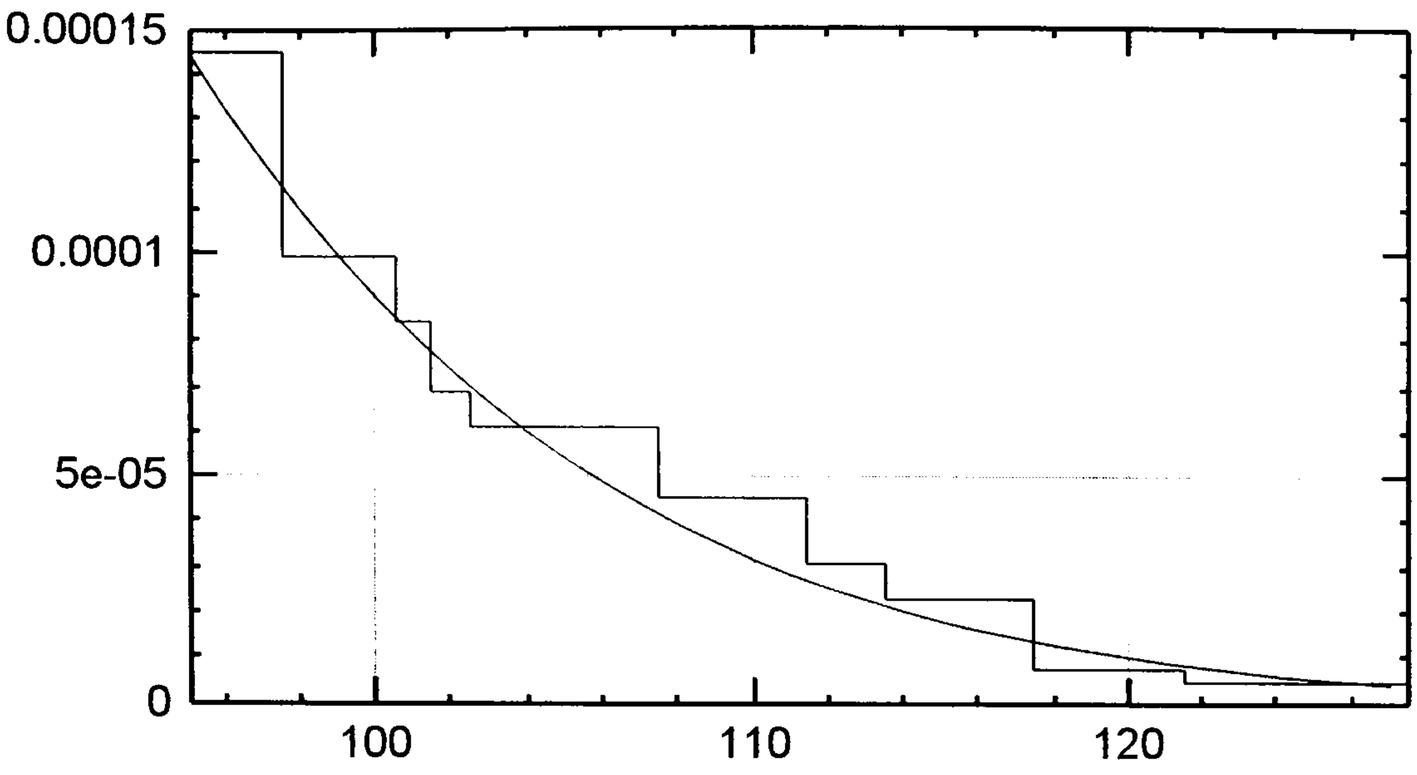


Figure 7.6: Zoomed section of tail of PDF from Fig. 7.4 from 3σ to 4σ.

Approximation using Non-uniform Element Probabilities

To better match the tails of continuous distributions is is necessary to reduce the minimum probability of some elements, as this will reduce the area contributed by that element to the overall PDF. There are two ways of doing this: the simplest is to choose an area-efficient algorithm for generating a non-uniform distribution of element indices, and then to apply the approximation process using this fixed set of probabilities. The alternative is to allow the approximation process to vary the weights in order to achieve the best fit, then to create an index generator that follows this optimal distribution.

Using a fixed set of probabilities is the most direct of the two choices, and adds no extra overhead to the search algorithm compared with approximations using equal-probability elements. Each element in the composition is assigned an initial probability weighting, which is used to scale the height it contributes when the composition's PDF is calculated. This weight is maintained during the optimisation process as the elements are re-positioned: weights are not exchanged between elements or altered during the search process. It might appear that elements with small probabilities would get hidden in amongst the larger elements, and so would never actually make it to the tails, but by using a tail specific error metric, such as sum of relative error, the small elements are "sucked" out of the mass of larger elements and positioned at the tails. During approximation the algorithm can be seen to build the distribution starting at the tails (where the relative error is the highest), then work its way inwards towards the higher probability areas.

The fixed weight set used here is the triangle distribution, generated in the same way as for a triangular element of a composition. For an n bit wide index this provides a dynamic range (highest weighting vs. lowest weighting) of $n : 1$. A minor disadvantage of this method is that one element index will never by generated (as $2(2^{n-1} - 1) < (2^n - 1)$), but when using block RAMs this will be only 1 out of 512 or 1024 elements, and so is not a

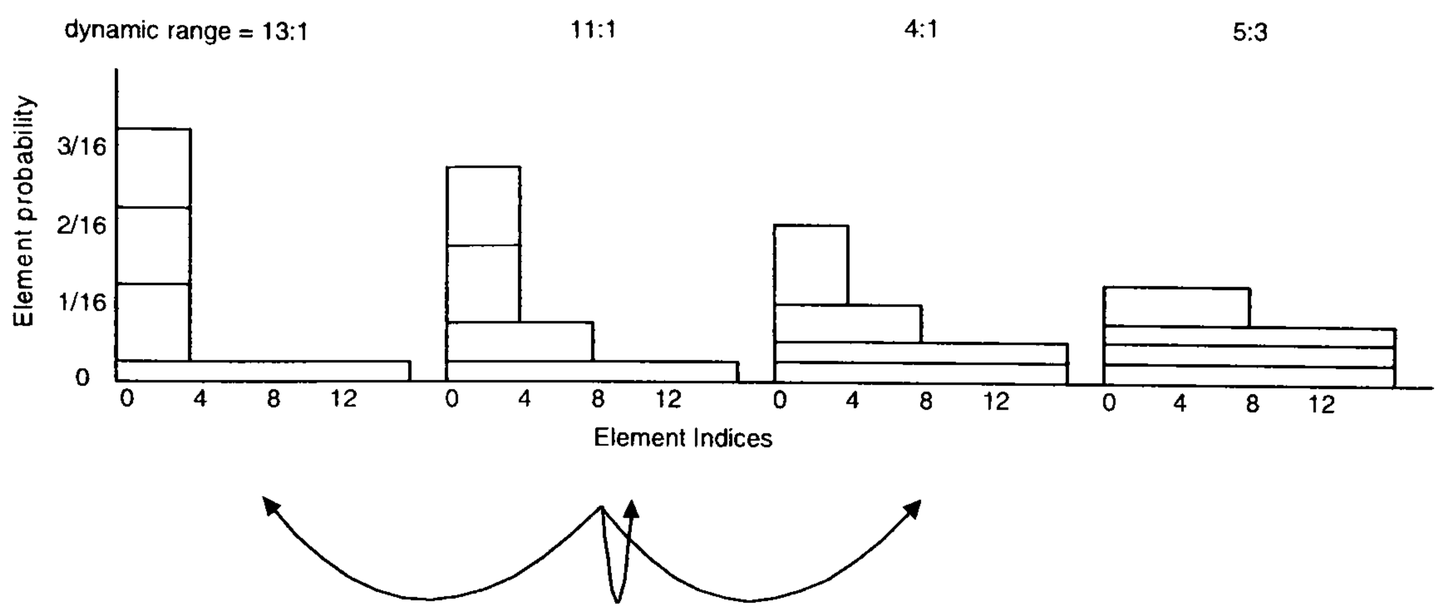


Figure 7.7: Example element index distributions using four elements to provide a four bit index, showing two possible configuration changes.

critical loss when compared with the ease with which this distribution can be generated.

Varying element probabilities during approximation makes the search process more complex, as every variation in the index generator distribution changes the global shape of the final approximation PDF, increasing the number of possible configurations. In order to reduce the search space and to ensure that the set of element probabilities can be efficiently implement in hardware, a small composite rectangle approximation is used to generate the indices used by the overall composite generator. As it does not matter in what order the index probabilities occur, only the set of probabilities, the index generator only uses rectangles with zero offset. This reduces the index generator to a single ROM, a mask expander and a bitwise and operation.

In order to make the search process as efficient as possible, not all configurations of probabilities supported by the rectangular index generator are allowed, instead only a fixed set S is allowed. In addition, for any particular set of index probabilities s_i the approximation process is only allowed to move to a small subset of the complete set of allowed configurations, $N(s_i) \subset S$. After moving from s_i to a new configuration $s_i \in N(s_i)$, the approximator may then move to a different subset of configurations $N(s_j) \subset S$, where $s_i \in N(s_j)$ and $N(s_j) \neq N(s_i)$, so the original configuration may be selected again if necessary. The set of allowable moves $N(s)$ is chosen so that the change in probability for each index does not change significantly with each possible move, but by making multiple moves it is possible to achieve great changes in index probabilities if necessary.

Figure 7.7 shows four candidate index probability distributions, using a four rectangle distribution to provide a four bit index. The arrows indicate possible movements between configurations that the approximator might be allowed to make, which result in the probability of each index remaining roughly the same. Note that it is always permissible for the optimiser to retain the current configuration. The set of permissible moves for each configuration is determined by choosing a maximum permissible relative probability change for each element, which is then used by the approximation software to create an adjacency

graph to allow efficient movement between configurations.

An example of approximating the exponential distribution using four different element weighting methods is shown in Figure 7.8 and Figure 7.9, demonstrating the effect on the distribution of the tail when using 64 triangle elements. The uniform probability version can only provide a non-zero probability up to zero (on the adjusted scale), but then has no way to provide a lower probability element and so the PDF drops to zero. The slight bump right at the end is the tip of the thinnest triangle that can be produced with uniform weighting. Using fixed triangular weighting provides an order of magnitude reduction in the lowest approximation probability over uniform weighting, increasing the extent of the approximated tail. There are now three separate bumps before the PDF drops to zero, produced by three separate triangles of decreasing probability that have been stacked to create the tail of the curve.

Using a 64 rectangle composite with dynamic index probabilities allows much lower probability values in the extremes of the tail, but decreases the accuracy of the rest of the curve. The approximation was created using the relative error metric, which has forced the approximation process to concentrate on the very low probability portions of the curve, at the expense of higher probability sections. This effect is even more pronounced when using a 1024 rectangle composite to create the indexes. The minimum probability is now two orders of magnitude below the uniformly weighted approximation, but the shape of the higher probability areas is now very distorted.

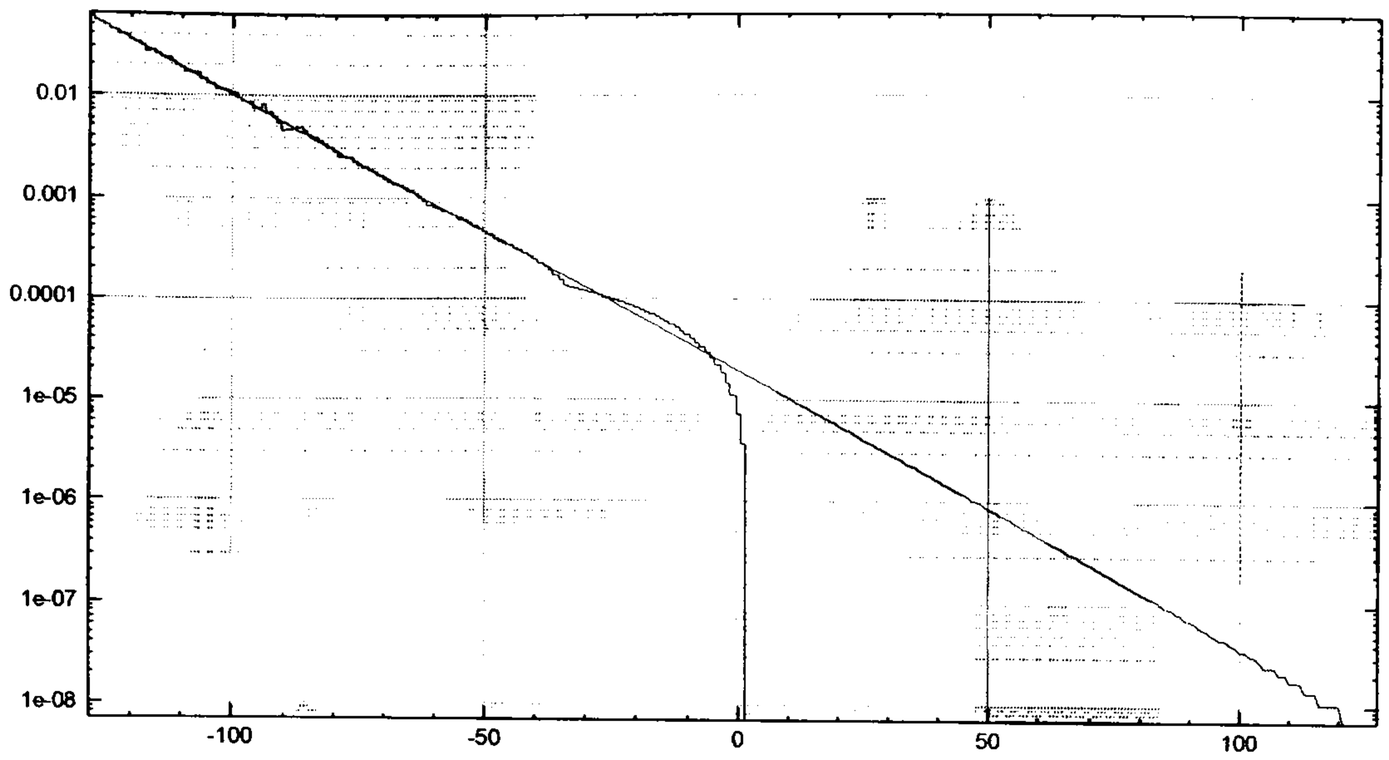
7.3 Results

In this section the hardware performance and resource usage of the approximation unit is measured, and a speed and quality comparison with other generators is made.

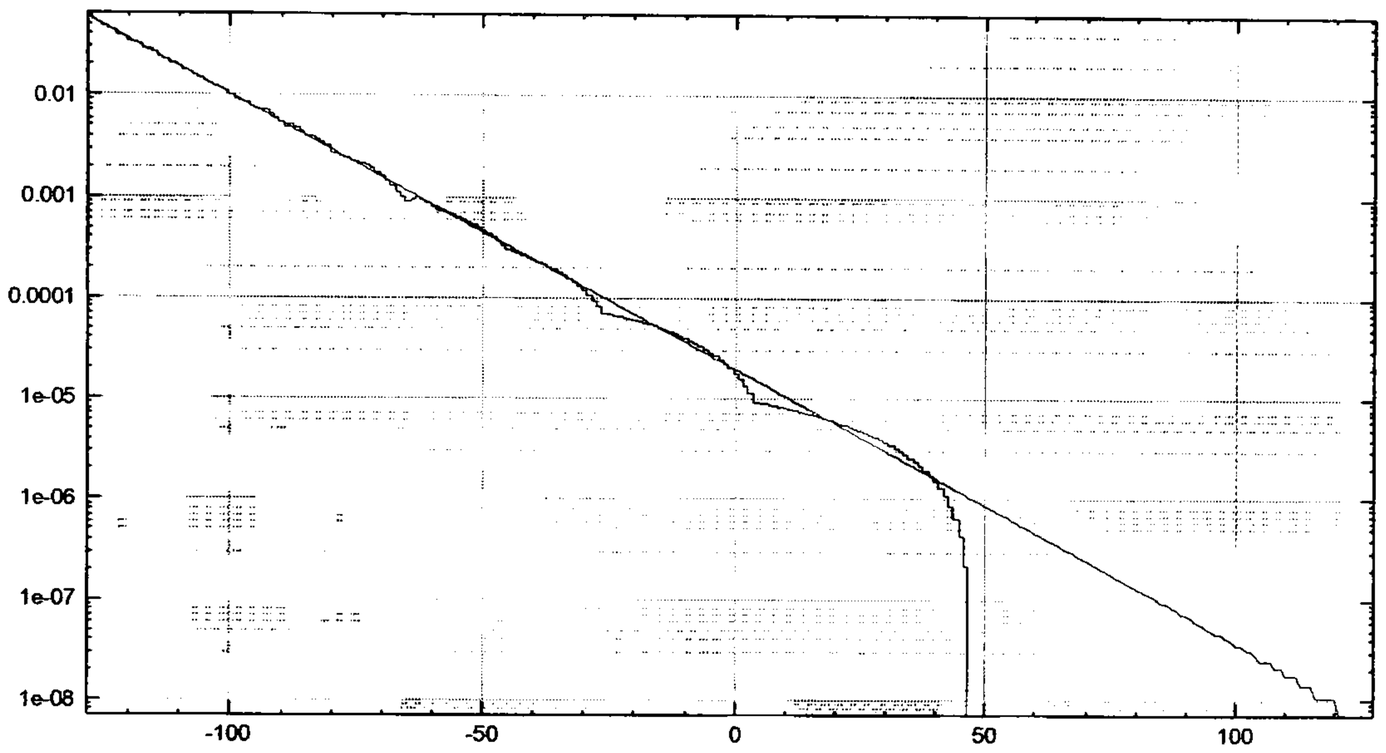
7.3.1 Hardware Performance

The hardware architecture was programmed using Handel-C, synthesised using Synplify, and then placed and routed using ISE 6.1 to produce Virtex-2 configurations. Table 7.1 shows the hardware speed and resource usage for different combinations of parameters. Note that the results include the area required for all uniform random numbers required by each unit. These are provided by a XorShift based generator [138] with a period of 2^{128}, which uses 113 slices when instantiated in isolation.

The first group of generators uses small LUT based ROMs to hold the element tables rather than block RAMs, and represent the case where only a very crude approximation is required. The second group uses a single block RAM with a priority on number of elements rather than output range, while the third group prioritises output width. The speed remains around 200Mhz for all configurations without any specific optimisations for speed, meaning that each generator will be able to provide 200M random samples per second.



(a) Uniform weighting

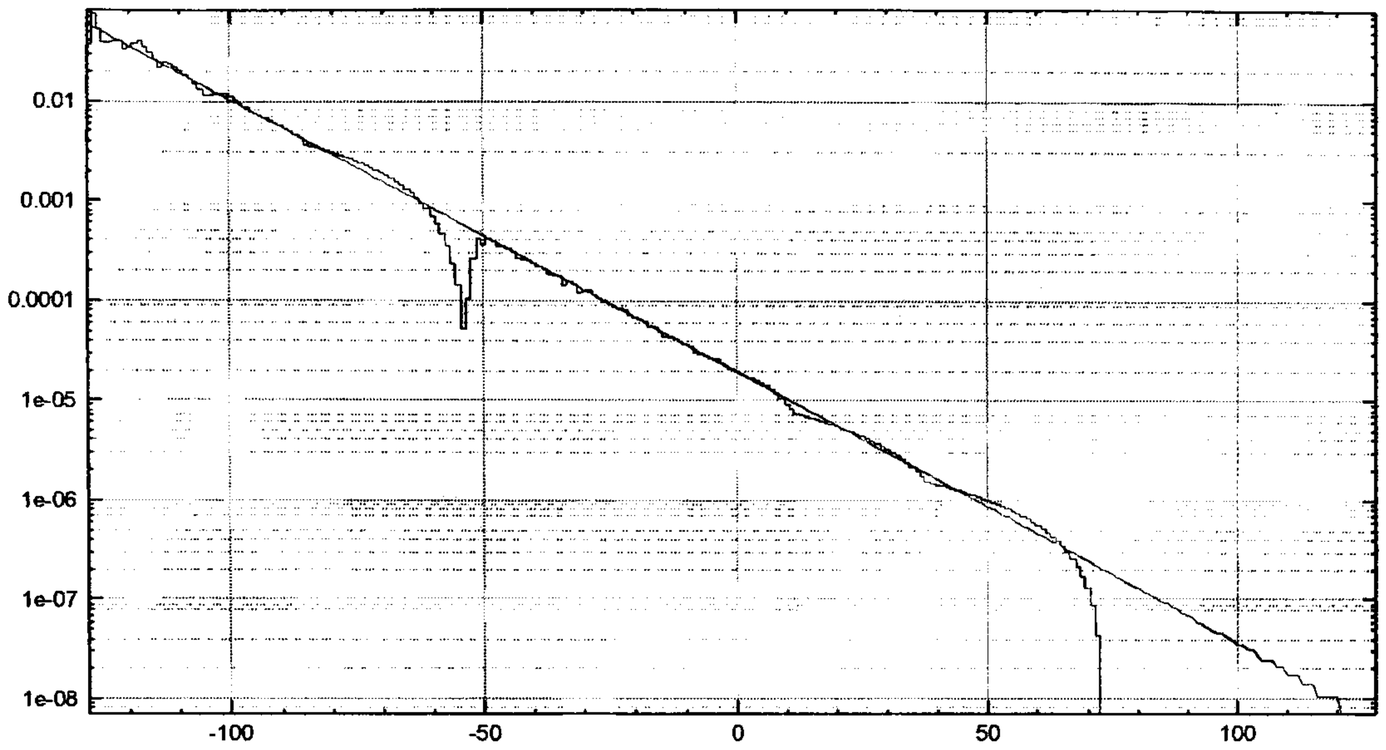


(b) Triangular weighting

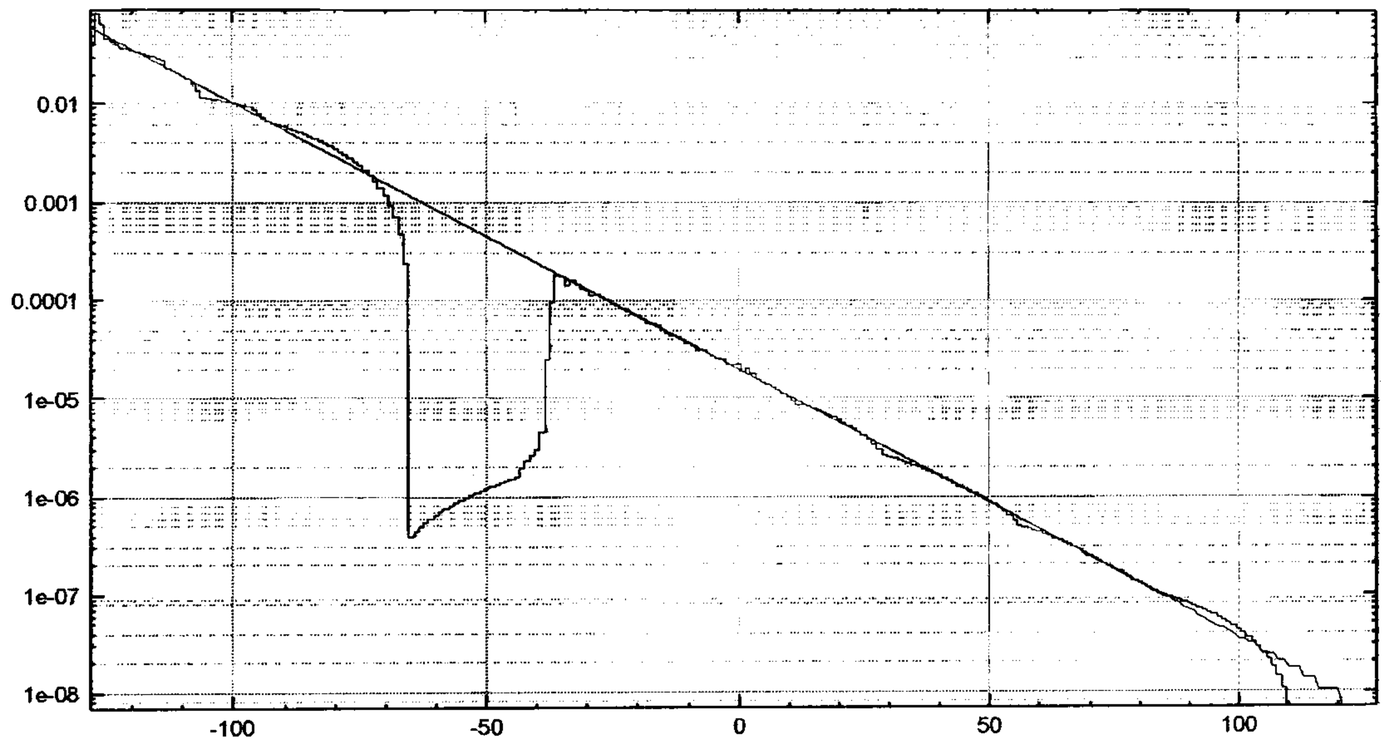
Figure 7.8: Using non-uniform weightings to increase the length of the tail of an exponential approximation. The composite consists of 64 triangles. The first approximation uses uniform weighting, and second uses triangular weighting.

Shape	Count	Range	Slices	MHz	RAMs	DSPs	LUTs	FFs
Rectangle	16	10	121	220	0	0	85	217
Triangle	16	10	136	210	0	0	89	244
Trapezoid	16	10	137	211	0	0	90	244
Rectangle	1024	14	144	215	1	0	105	237
Triangle	1024	14	160	212	1	0	104	271
Trapezoid	1024	10	188	208	1	0	120	310
Rectangle	512	24	267	205	1	0	156	452
Triangle	512	24	253	198	1	0	157	449
Trapezoid	512	24	270	195	1	0	220	526

Table 7.1: Performance and resource usage of approximation units in Virtex-II architecture.



(a) Rectangular composite with 64 elements



(b) Rectangular composite with 1024 elements

Figure 7.9: Using non-uniform weightings to increase the length of the tail of an exponential approximation. The composite consists of 64 triangles. The first approximation uses dynamic 64 element non-uniform weighting, and the second uses 1024 element non-uniform weighting..

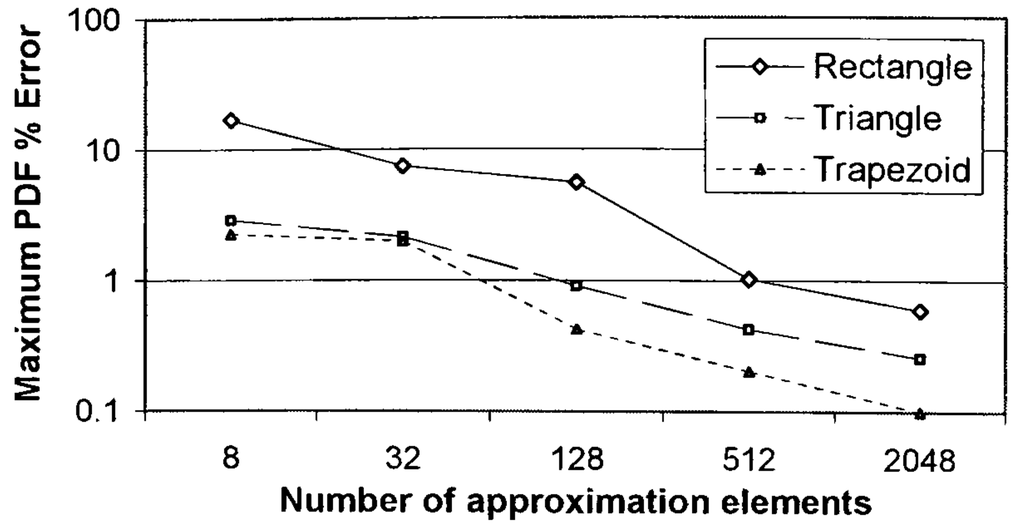


Figure 7.10: Graph showing the difference in approximation accuracy when using different component types to approximate the Weibull distribution. The maximum absolute error is shown as a percentage of the largest probability in the PDF.

7.3.2 Approximation Quality

Here we present results on the quality of the samples that composite generators can provide. It is worth bearing in mind that the accuracy of the approximation is highly dependent on the process used to create it, and it is not currently known how close the current system comes to the optimal approximation. For that reason the results presented here should be seen as benchmarks of the current approximation system, rather than of composite generators themselves. Improved methods for creating approximations may drastically improve quality, without requiring any changes to the underlying generator architecture used in hardware.

Unless otherwise specified the approximations used in this section were generated automatically without any human interaction. An arbitrary time limit was also imposed on the approximation process, dependant on the number of elements being used. For an approximation with k elements, the approximation system was allowed $\sqrt{k}$ minutes of CPU time on a 2.4GHz Athlon with 2GB of RAM. This means that, for example, approximations with 8 elements have about 3 minutes CPU time, while those with 2048 elements get about 45 minutes. The χ^2 tests used here were performed by bucketing n samples at the range resolution, then coalescing adjacent buckets until the expected count per bucket exceeds $n/n^{0.4}$.

Figure 7.10 shows the maximum PDF error found while approximating a Weibull distribution for different component shapes and numbers of components (the PDF of the Weibull distribution is shown in Figure 7.3). The error shown is the maximum absolute error expressed as a percentage of the largest probability in the target PDF. The rectangular approximation is at a disadvantage with any smooth continuous distribution like the Weibull, and this shows as a very high error for 8 blocks, but gradually improving for larger numbers of blocks. The triangular and trapezoid approximations start off at a lower error, due to the ability to match the coarse features of the distribution even with a low number of components. Both the rectangular and trapezoid approximations roughly half the maximum PDF error for each quadrupling of the component count, while the triangular approximation does

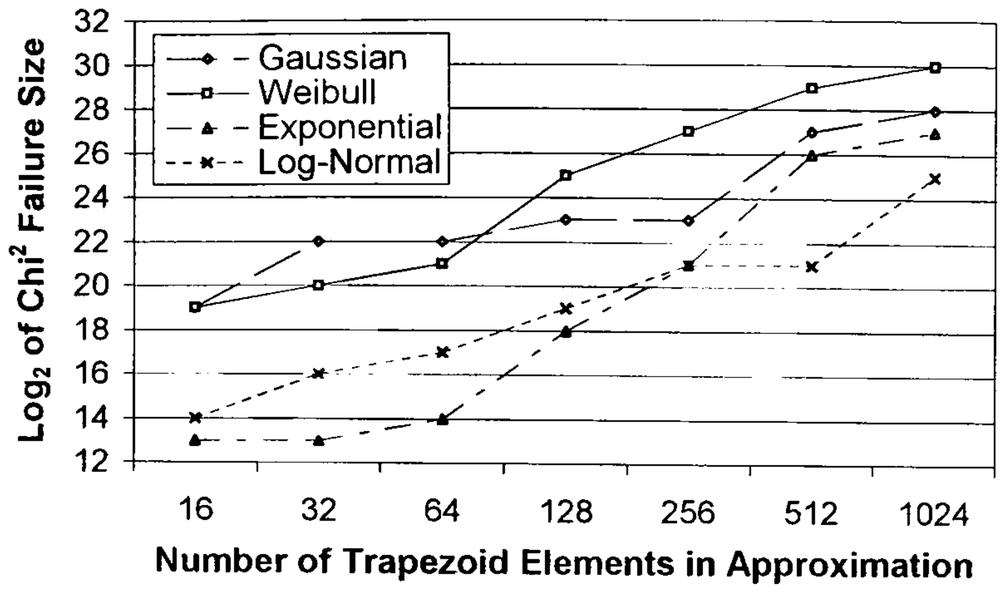


Figure 7.11: Graph showing the number of samples that can be taken from trapezoid approximations to four distributions without failure of the χ^2 test (at the 5% significance level). Sample sizes of increasing binary powers were tested and are shown in the graph as $\log_2$ the tested sample size.

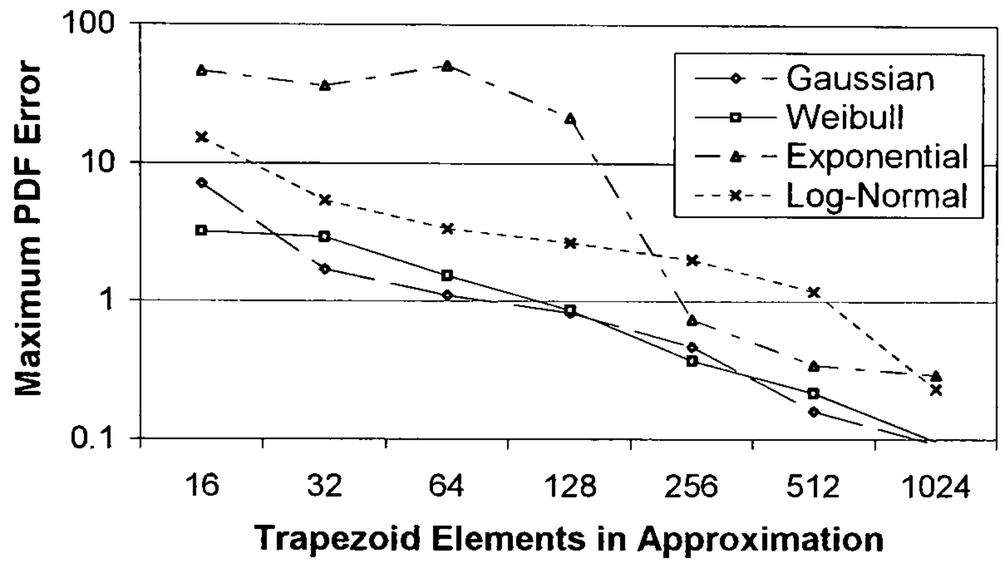


Figure 7.12: Graph showing the accuracy of trapezoid approximations with different numbers of elements for four different target distributions. The maximum absolute error is shown as a percentage of the largest probability in the PDF.

slightly worse, with the error dropping by about a third each time the component count is quadrupled.

In Figure 7.11 the χ^2 test is used to determine how many samples can be take from four different distributions before failure. All the distributions were approximated using trapezoid components. The approximation to the Weibull distribution appears to be much more effective than all the others, possibly because the mode is more extended and smoother than the other distributions. The Exponential distribution causes difficulty for the approximation system, as the system could not place triangles at the mode to approximate the steep slopes without the triangle extending into negative numbers (where the Exponential PDF is zero), which leads to a poor approximation where the PDF has the highest probability. The maximum PDF error as percent of maximum PDF for the same set of generators is also shown in Figure 7.12, showing that until 256 components are used the approximation error is large.

To explore how effective the method can be if accuracy concerns override speed and area,

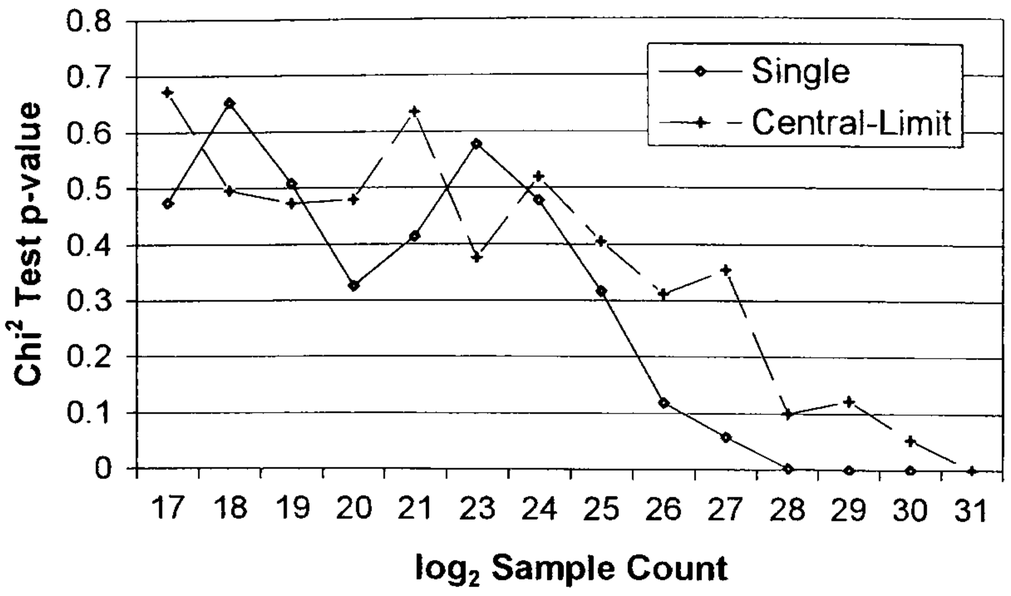


Figure 7.13: Graph of p-values for χ^2 test of 2048 trapezoid approximation to the Gaussian distribution. In the single sample case one value is taken from the approximation, while in the Central-Limit case two separate values were generated in parallel using the same table and combined through subtraction.

a more complex approximation to the Gaussian distribution is developed. This uses 2048 trapezoids, with a 16 bit output range, requiring 3 block RAMs to implement the 2048x24 bit table. The approximation process was also allowed enough CPU time to reach a steady state (about four hours), and where necessary the search process was manually tweaked. This results in a distribution that extends to about $\pm 5.1\sigma$, beyond which it is impossible to approximate smaller probabilities. The maximum absolute PDF error is 3.4×10^{-07}, or 0.34% of the peak probability.

Figure 7.13 shows the p-values for the χ^2 test for normality, applied to the 2048 element generator for increasing batch sizes. The "Single Sample" results show that the test is passed for sample sizes up to about 50 million samples. For larger sample sizes the p-value falls below 0.05, and are no longer normally distributed.

In the "Dual Sample" tests two samples are combined together to provide each sample, taking advantage of the Central Limit Theorem. to improve quality. Rather than combination through addition, the samples are combined using subtraction, as this removes any asymmetries of the single distribution. The combined PDF is non-zero within the range $\pm 5.2\sqrt{2}\sigma$, (about $\pm 7.2\sigma$), and is much more accurate, with the generator just on the verge of failing the χ^2 test at 1 billion samples.

The same 2048 trapezoid generator (without Central-Limit accumulation) was also used to investigate the generation of antithetic variates, as described in Section 7.1.3. The components are sorted by offset, and for successive outputs the components were chosen as i and $2047-i$, then the chosen component distributions were sampled independently. One million pairs of output samples are generated, then split into two pointwise-correlated ordered sets a and b. The correlation coefficient for the set of pairs (a_i, b_i) is -0.698, indicating a very strong correlation between the two. The correlation coefficient between all sets of pairs in (a_i, b_{i+n}), where $n \neq 0$, shows no significant correlation, demonstrating that the correlation

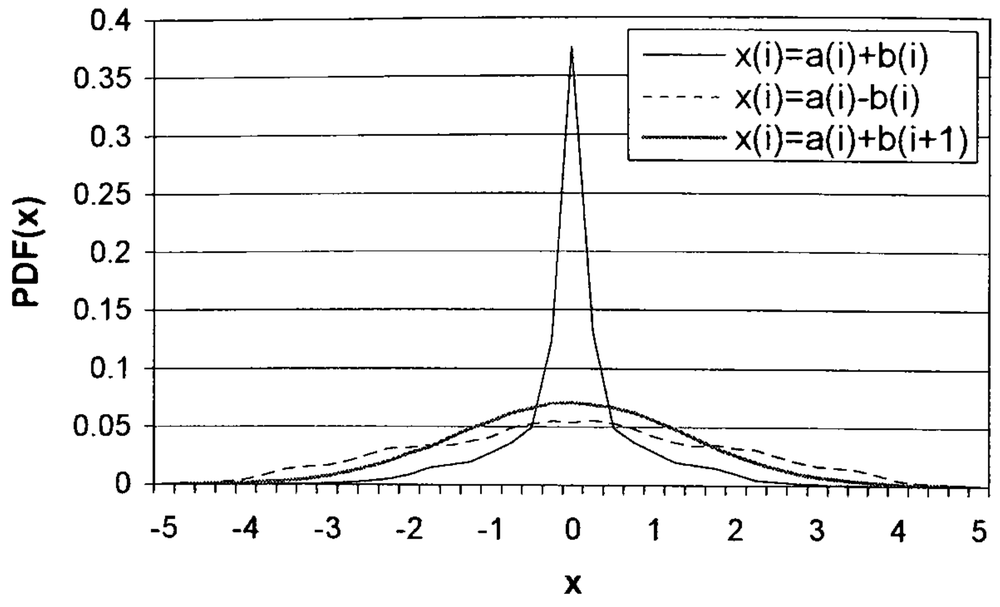


Figure 7.14: PDFs of the sums of pseudo-antithetic variates generated using a 2048 trapezoid approximation to the Gaussian distribution. Two point-wise correlated lists of variates a and b are combined using addition and subtraction, showing the change in the variance of the resulting PDF. If the sum of offset pairs from a and b are combined then there is no correlation, and the resulting PDF is Gaussian.

only occurs for pairs of samples generated in tandem.

The correlation is shown in another way in Figure 7.14, where the PDF of the sum of the correlated pairs is shown. When the pairs are added ($a_i + b_i$) together the variance of the resulting PDF is 0.6, and has a strong spike around zero, but if the pairs are subtracted ($a_i - b_i$) then the variance is increased to 3.4, producing a flattened PDF. If the sum of samples from different output pairs ($a_i + b_{i+1}$) are taken then the PDF is Gaussian, with the expected variance of 2.

Although the production of antithetic Gaussian variates is actually very simple due to the distribution symmetry, this technique can be applied to any distribution, whether it is analytically or empirically defined. While the correlation between variables will never be perfect, the technique will provide a level of "free" variance reduction for almost any distribution without requiring analysis or specialised hardware.

7.3.3 Comparison with Alternate Methods

Table 7.2 shows a comparison between methods for creating Gaussian samples, including previous hardware implementations, component generator based hardware using trapezoids (referred to as "Trapezoids" for the rest of this section), and software implementations. The existing hardware and software methods are all specialised (and optimised) for the Gaussian distribution, while except for the CLT component generator, all the component generators are generic and can produce any distribution. All the generators are implemented in the Virtex-II family, except for the Ziggurat, which uses Virtex-II Pro. The "Dual" Trapezoid generator is the case where two generators are sharing a dual-ported RAM. The reported resource usage, speeds, and number of samples before χ^2 were extracted from the papers where they were presented, except for the χ^2 failure point for the Box-Muller, which was

Class	Previous Hardware			Trapezoids				Software	
Method	BM	Wal	Zig	Mono	Mono	Dual	CLT	Zig	Wal
Ref.	[236]	[124]	[250]	-	-	-	-	[141]	[223]
Elts	-	-	-	2^9	2^{11}	2^{11}	2^{11}	-	-
Width	24	24	35	24	16	16	17	-	-
MHz	245	155	168	195	193	196	194	-	-
Slices	455	895	891	270	256	411	451	-	-
RAMs	5	7	4	1	3	3	3	-	-
DSPs	5	4	2	0	0	0	0	-	-
MS/s	245	155	168	195	193	392	194	35	36
χ^2 Fail	2^{17}	$2^{32}+$	$2^{30}+$	2^{26}	2^{27}	2^{27}	2^{30}	$2^{32}+$	$2^{32}+$

Table 7.2: Comparison of different methods for generation Gaussian random numbers (Zig = Zigurrat, Wal = Wallace, BM = Box-Muller).

reported by a third party [124]. The software speeds were measured on a 2.66GHz Athlon with 2GB of RAM.

In terms of slices the Trapezoids are all smaller than the other hardware methods, although the CLT based generator is only just smaller than the Box-Muller. However, the Box-Muller uses 5 block multipliers, and all three non-Trapezoids use at least two multipliers, and the Trapezoids also use less block RAMs than the Gaussian generators. When slices, multipliers and block RAMs are all considered together it is clear that the Trapezoids use much less area, and will put less pressure on resource usage in any design that uses random number generators.

The speed of the Trapezoids also compares well with the other hardware methods. Although the Box-Muller is actually 50MHz faster, it uses explicit placement and device specific features to achieve the best speed. A more appropriate comparison is with the Wallace and Ziggurat designs, which were implemented using Handel-C and VHDL respectively, and so use a very similar tool chain to that used to create the composite generators. The Wallace is approximately 40MHz slower than the composite, or about 80% of the Trapezoid speed. Although the Ziggurat appears to be slightly faster than the Wallace, it should be noted that this was implemented using a Virtex-II Pro, which provides a small intrinsic speed advantage over the Virtex-II [239], and so on equal terms is probably about the same speed. Many applications that consume non-uniform random samples will need to apply numerical operations to the generated samples, so it is unlikely that 200MHz will be a bottleneck, but the "Dual" generator, where the RAM table is shared between two generators, can provide 392 MSamples/s.

Comparing the generators on quality is very difficult, as methodologies vary between papers. The testing performed on the Wallace generator is by far the most extensive, and it clearly passes the χ^2 test convincingly for more than 2^{32} samples, producing extremely high quality samples. In the same paper independent testing is also performed on the Box-Muller generator (the original documentation does not include any results), and it is concluded that

the generator has serious deficiencies, failing at just 2^{17} samples. This mainly seems to be cause by a large spike in the PDF at the origin, and is likely a symptom of a bug rather than any flaw in the architecture or methodology.

It is claimed that the Ziggurat method passes the χ^2 test for 10^9 samples over 512 buckets [250]:

> The χ^2_{511} statistic is 422.539. For a 95% level of confidence, the critical value is 565, showing that our hardware Ziggurat design produces good quality results.

However, while the statistic is below the 95% threshold, it is also below the 5% threshold of 460. The probability of a test statistic less than 422 is only 0.0017, which while not an outright failure is also not a convincing pass. This may indicate that the generator is suffering from "super-normality" with a PDF that is too perfect to be random, although it could be also be a statistical blip or a typo.

The quality of the Trapezoid generators is their weakest point: even with the Central-Limit extension only 2^{30} samples can be generated before the χ^2 test is failed. At 200Mhz this represents only about 5.5 seconds worth of output. Although the quality is better than that of the Gaussian specific Box-Muller generator, a bug-free Box-Muller would be expected to provide samples of at least or higher quality. However, for many applications the provided quality may be good enough, and future improvements to the approximation process can be expected to improve the quality.

When compared with software generators the main advantage of the Trapezoid generators is that they provide over twice the sample generation rate. However, for other distributions the difference may be much greater, as not all distributions have received the attention and optimisation of the Gaussian. Analytically defined distributions specific to a single task or empirically defined distributions will be much slower in software, having to rely on one of the arbitrary distribution generation methods available in software.

7.3.4 Integration with IGOL

The hardware generators described in this chapter have been made available to software applications through the IGOL framework. The functionality is available both as a random number generator through GSL (GNU Scientific Library), and as an additive noise filter for video and audio.

The GSL non-uniform random number generation interface supports many distributions such as the Normal, Exponential and Log-Normal distributions, but does not provide a mechanism to allow third-party generators to be accessed. To allow IGOL to intercept the calls the default generators were replaced with a level of indirection that either called the existing generator method, or redirected to an IGOL Operation Instance. For applications that use the GSL this is transparent, and requires no changes.

The GSL random number interface returns just a single value per function call, so to reduce the overhead a caching system is used, whereby random numbers are constantly

transferred from hardware into a memory based FIFO by a background thread. When the application requests a random number, the next sample in the FIFO is returned, reducing the cost of generation to a pointer increment, compare and memory access, rather than a full round trip to hardware. The rate of random number generation is limited only by the speed of the data transfer bus, with a theoretical peak of 60 MSamples/s for the RC2000 and UltraSonic, and 30 MSamples/s for the RC1000. The sustained generation rates when accessed through the GSL interface were 45 MSamples/s and 26 MSamples/s respectively.

If the hardware generation rate on the RC2000 is compared with the software generation rate using an Athlon 2.66GHz, then a speedup of 1.36 was observed for the Normal distribution, 1.8 for the Log-Normal distribution, and 1.28 for the Exponential distribution. Generation through hardware also requires less CPU time: on average the hardware generator used 10% of CPU time to achieve its peak generation speed. The remaining 90% of CPU time can be used by the application, as the hardware transfers are performed on a background thread.

A Monte-Carlo simulation using the Black-Scholes pricing model [19, 134] was implemented using GSL as the source of Log-Normal random numbers. When software generation was used the random number generation process occupied 53% of total execution time. When the same application was executed using hardware random number generation on the RC2000 via IGOL the random number generation overhead was reduced to 9% of execution time, providing an overall speedup in execution speed of 1.78 times.

The random number generators are also exposed in IGOL as additive noise generators for use in video and audio applications. Gaussian noise is commonly used in both fields to simulate the effects of noise in communications channels or capture devices such as video cameras. Through IGOL these noise sources are exposed in two ways: as a source of pure noise, with no inputs, or as a noise addition process, with one input and one output.

When used as a noise source the filters output video frames or audio samples that consist solely of Gaussian noise, which can be used either as an aesthetic effect [129], or as a building block in video processing [2]. This is implemented in the same way as the GSL random number source; random numbers are continually transferred from the hardware device to a memory buffer via a background thread, making full use of the bandwidth to hardware.

When used as an additive noise transform the filter adds Gaussian noise to the input data, with the amplitude of the noise selected through a parameter. This filter is implemented using two different components, allowing IGOL to select the best method at run-time. In the first mode the filter component contains both a software and hardware component, with the hardware acting as a source of random numbers that are constantly transferred in a background thread, and the software component actually adding the noise to each data sample. In the second mode the filter component contains just a hardware configuration which is managed by IGOL. Input data is transferred to hardware, then noise is added, before data is transferred back.

The first mode provides optimal performance when no other IGOL hardware filters are applied to the data, as maximises the bandwidth available from hardware, performing the addition of input data and noise using vectorised software instructions. The second mode can provide better performance where two or more IGOL filters are applied to the same data, using one of the hardware sharing mechanisms explained in Chapter 5. Under advanced plug-ins such as the Direct-Show plug-in, the IGOL framework is able to select the best mode at run-time.

When Gaussian noise is applied to a 640x360 video stream using DirectShow (using the same testing methodology applied in Chapter 5), with no other filters applied, the processing rate is 36 frames per second in software, compared with 120 frames when an UltraSonic is used. The large difference in speed is because a Gaussian sample is needed for each channel of each pixel, requiring three sequential calls to the Gaussian noise generator in software. However in hardware the three values can be calculated in parallel by three separate generators, and then combined with the pixels using vectorised packed arithmetic [99]. This provides a speedup of 3.3 times when the hardware random number generators and IGOL are used together.

7.4 Summary

This section has demonstrated the viability of a method for hardware approximations of arbitrary probability distributions in hardware. The hardware architecture has been shown to be competitive with traditional methods for non-uniform random number generation in hardware, with lower area requirements and higher speed than most Gaussian specific random number generators. A technique for creating approximations to arbitrary probability distributions has also been presented, and statistical quality for sample sizes up to 1 billion variates has been shown. The key advantage of this technique over other hardware non-uniform random number generation techniques is that the hardware design is independent of the distribution to be produced, and the distribution can be changed at run-time.

The hardware random number generators can be used both as a source of non-uniform variates for simulations in hardware, and for use within software simulations. Using the IGOL framework applications can use such hardware generators without requiring any code changes, through numeric libraries such as the GNU Scientific Library, or by utilising IGOL Operation Instances directly. The absolute speedup in random number generation rate is limited by the data-transfer bandwidth of hardware accelerators, but the overall speedup of applications is greater because less CPU time is required to generate each sample. A speedup of 1.78 times was observed in a monte-carlo simulation when Log-Normal samples were generated via the IGOL plug-in for the GNU Scientific Library, and a speedup of 3.3 times the software rate was seen when adding Gaussian noise to video frames.

Chapter 8

Conclusion

The two main topics addressed in this thesis are the management and use of hardware acceleration from software applications, and random number generation in FPGAs. These two methods can be combined to create an accelerated source of random numbers for software simulations.

In Chapter 3 a methodology for describing data processing operations and the relationships between them was presented. This provides a two level system of specification and implementation, allowing applications to request a data processing operation without needing to know how that operation is implemented. This allows applications to be given the best performing implementation of the operation available given the resources that are free. By defining the operations first and providing a reference implementation the development and testing of software applications can be started before the hardware components they will eventually use are complete.

In Chapter 4 the IGOL framework was presented, a framework for data processing which implements the ideas presented in Chapter 3. This framework supports the transparent use of hardware and software data processors, automatically selecting the best available implementation at run-time. A shared registry component provides a common rendezvous point for all parties that use operation meta-data, such as the IGOL activator. The activator is able to select the optimal software or hardware implementation on behalf of applications, both when instantiated, and during processing. A number of different plug-ins for existing applications are presented, including a method for accessing hardware random number generators through the GSL (GNU Scientific Library) interface.

In Chapter 5 the performance of IGOL video operations using two different FPGA expansion boards, the RC1000 and UltraSonic, was compared to software implementations. For non-trivial filters speedups of up to 6 times were achieved using the RC1000, and up to 7 times on the UltraSonic. The use of generic video optimisation techniques was also demonstrated, such as segmented processing which increased the frame rate of RC1000 based filters by a factor of between 1.15 and 1.42 times without requiring any changes to the

hardware implementations. Finally three types of co-operative device sharing were demonstrated. On the UltraSonic this was achieved using a dedicated hardware path, providing an almost constant frame rate as more filters are applied. On the RC1000 two forms of device sharing were used, one based on the use of pre-compiled combined configurations, providing a speedup of over 5 times when two filters are applied, and another which reconfigures between different filters while avoiding redundant data transfers, which provides around 2.5-3 times the performance of software without requiring changes to the hardware implementations.

In Chapter 6 the randomness of existing hardware based random number generators was examined, and it was found that providing high quality randomness using these methods requires large amounts of hardware. Lagged Fibonacci generators were proposed as a method for hardware uniform random number generation, and three very high quality random number generators were designed, optimised to make use of RAM, DSP or logic resources. All three generators pass BigCrush, the most stringent test for randomness currently available. These uniform random number generators are made available to software applications through the IGOL GSL interface, providing software applications with a hardware accelerated source of high quality random numbers.

In Chapter 7 a technique for approximating arbitrary probability distributions was described that is particularly well suited to hardware. Unlike other distribution specific techniques, the generated distribution can be modified at run-time without needing to recompile hardware. This is achieved by approximating the target distribution using a composite of multiple smaller distributions of rectangular, triangular or trapezoid shape. The approximation is created using a custom optimisation strategy, which allows the emphasis of specific aspects of the target curve, such as the overall shape or the length of the distribution tails. In concert with the XorShift random number generator this technique allows for very lightweight, yet flexible probability distribution generators, using less than 200 slices and a single block-ram in the Virtex-4. The use of non-uniform weight distribution between elements in the composite is also examined, which allows more accurate approximations in the tails of distributions. An approximation to the Gaussian distribution, approximated using the sum of two composite approximations, is compared with other hardware Gaussian distribution generators. While it cannot achieve quite the same level of accuracy as the best Gaussian specific generators, it uses the least resources and provides the highest sample rate.

As a framework IGOL can be evaluated in terms of metrics such as processing performance, but one aspect of its success or failure is the way in which it has been used. IGOL is intended to be useful beyond its role as a research project, and it must also be evaluated it on this criteria.

One of the goals of IGOL was to encourage the use of reconfigurable hardware in everyday applications, allowing users with no knowledge of FPGAs to benefit from faster processing rates. In this the system has been very successful, particularly in the field of image and video processing. Hardware accelerated image processing components can be transparently

accessed in eight different applications: if hardware is available then it is automatically used, otherwise software processing is used. No action is required on the part of the user, and no hint is given that hardware acceleration is in use, apart from the increased processing speed.

A second goal is to encourage the re-use of hardware designs, by packaging them up and deploying them, with the minimum of software required from the hardware designer. In this the system was also successful, as in the majority of cases no software at all must be written: all that the hardware designer needs to do is to describe the hardware bitstream using the IGOL meta-data language, package the meta-data and bitstream together, then the hardware component can be deployed to all available machines. Wherever the component is deployed the local IGOL registrar will then automatically load and use the component with no further work required.

One area in which IGOL has disappointed has been that of specification, as the system was intended to support a strong model of functional specifications. Although the specification model works perfectly, it has not proved as much of a benefit as hoped. The IGOL system has been used by the author and by third parties, for writing both software and hardware components, and the only times when the functional specifications were properly used was in case studies of IGOL. When a real task was performed using IGOL it was often found to be too difficult to write adequate constraint based specifications, or if a reference implementation was specified then no test data was specified.

However, although the functional aspects of Operation Specifications were not found to be practical, the split between Specification and Implementation was, and even third parties using the system typically first created an Operation Specification, then created a reference software Operation Implementation, followed by hardware Implementations. The ability to have multiple versions of a hardware component installed on a machine was also greatly appreciated, as a known good version could be published as machine scope for demos, while the version being actively developed and debugged could be restricted to used scope.

8.1 Future Work

8.1.1 Data Processors

Network support in IGOL is currently limited, and should be extended in order to achieve the maximum utilisation of available hardware resources. There are currently quite a few ways that parts of IGOL can be used across the network, for example an operation instance can be used on a remote machine using DCOM (the automatic remoting system used by COM), but these require the application to explicitly request that the operation be created on a specific remote machine. The performance is also not optimal, because the DCOM network protocol is not optimised for streaming large amounts of data.

A customised protocol for streaming data to and from remote operations could use network bandwidth more intelligently. If the network connection to a remote computer is a

1GB/s link then high data transfer rates can be achieved, but it is important that the channel is always full of data. This requires a heavily pipelined protocol to make sure that the limitation is bandwidth rather than the latency of network calls.

In conjunction with the pipelined networking there should also be more general pipelining of data through IGOL operations. Most applications only support synchronous processing, so do not allows more than one frame or data packet to be processed at once, but advanced applications such as DirectShow are capable of having multiple frames being processed at the same time. IGOL should take advantage of this by applying the same pipelining of data transfer and processing that was used in the segmented processing optimisation, but applying it to whole frames, instead of segments within frames.

Optimisation of operation instances at the moment is limited in its scope, as the proxy only tries to move to faster implementation that are already known. There is no attempt to try to find a faster implementation somewhere else on the network, or to try to compile a faster implementation. IGOL should also support optimisation of the application, PC or even network cluster as a whole, rather than relying on each operation trying to acquire the fastest piece of hardware. Ideally a system that maximised performance for the whole system could be implemented, by coordinating all IGOL components to allow allocation of resources where they will provide the best speed-up.

One of the intentions behind the rich operation meta-data was to provide operation instances which could obey certain contracts, for example an operation might guarantee to provide a certain minimum processing speed, or a minimum latency. The operation activator could use this information to build a schedule that provided fall-back options, so that if the current implementation became unavailable then there would still be a backup implementation. The reason that the implementation might become unavailable is because the remote machine crashed, or the network goes down, or a higher priority operation instance kicks the executing implementation off the hardware. This backup might be less efficient in a global sense, e.g. the backup option has to use 90% of CPU time on the local machine as opposed to 5% of FPGA time on a remote device, but would still allow the application to meet its targets.

Even without providing performance guarantees this mechanism could still be useful. For example if a CPU-bound operation has been executing on a remote machine for 10 hours, but the network drops 10 minutes before it completes, then the processing task must be started from scratch. A solution would be for the local operation proxy to periodically take snapshots of the operation's, then if the connection is lost all that will be lost is the processing time since the last snapshot. This can be restored on a local implementation of the operation, and the computation completed. Many of these future directions are related to research into Grid infrastructure and services, which provide support for process checkpointing, networking and resource allocation.

Integration of IGOL into a grid framework would provide many advantages, such as allowing processing tasks to automatically migrate to machines with the best available hard-

ware. However, this would require IGOL to hook into the Grid infrastructure to allow it to detect events such as processing checkpoints.

8.1.2 Random Numbers

The Lagged Fibonacci generators are over-engineered so that they easily pass a specific test battery (Big-Crush), but it is unclear by exactly how much they exceed requirements. It would be useful to determine how much the design parameters (feedback length, tap count, operator) can be reduced until the different tests in the battery start to fail. The tests in TestU01 are also parametrised, so it is possible to make the tests in Crush and Big-Crush more difficult. Increasing the difficulty of the tests until failure would give another view on how good the generators actually are. This would allow the tuning of the generator parameters until all three generators are of roughly equivalent quality, and provide some insight into which method has the highest intrinsic quality, for example by comparing the area of generators with equal statistical quality.

The three generators presented also all use simple feedback logic, based on homogeneous word-length mathematical operators. This makes comparison and testing against software easier, but FPGAs offer the possibility of more interesting feedback logic at little or no cost. One question is whether combining different operators in a generator improves or reduces quality, for example a four tap generator might use $(a + b) \oplus (c * d)$, or some other combination of operators. If only one multiplication is needed to gain all the benefits then it might be possible to drastically reduce the number of DSP blocks needed.

Selecting tap polynomials based on the number of shift registers needed to implement that polynomial offers another possibility for optimising the hardware. By trying to select a polynomial such that the taps maximise the amount of each SRL16 in use, i.e. almost every SRL16 is configured to be of length 16, it might be possible to reduce the large amount of slices currently needed to implement the logic based generator, where many of the shift register spans are much less than 16.

8.1.3 Distribution Approximation

The optimisation strategy used to find approximations to target distributions works well, but has some problems. First, it does not scale well to a very large number of elements, or to very wide ranges. The number of configurations considered per second roughly halves every time either of these parameters is doubled, so trying to find a solution using a range of 2^{16} and 4096 elements the algorithm is almost impossible. A second problem is that the process of repositioning just one element seems, both intuitively and empirically, to provide worse solutions as the number of elements increases. More than one element needs to be moved at a time in order to escape local minima, but it is not clear how the correct elements to be moved can be identified.

One solution is to develop a new search algorithm which use many more heuristics and optimisations to cut down on the number of "silly" positions for elements. For example the current system will seriously consider placing an element in regions where the probability of the target distribution is zero. Another option is to move some of the search logic into hardware, as the logic required to evaluate the fitness criteria should fit on a modern FPGA. A good direction might be a hybrid system, whereby the software directs the search process, while the FPGA evaluates the error metric at the chosen points.

In this thesis only rectangles, triangles and trapezoids are used, but there are many more basic shapes that could be generated. Particularly interesting would be the use of wedges. These could be generated by generating a triangle as normal, but then shifting the centre of the triangle to zero then taking the absolute value of the value, producing a wedge shape. This could be very useful for approximating the tails of distributions, particularly if the wedge can be given an extremely low weighting.

Bibliography

[1] Adobe Systems Incorporated. *XMP Specification*, 2004.

[2] N. Ahuja and B. J. Schachter. Image models. *ACM Computing Surveys*, 13(4):373–397, 1981.

[3] P. Alfke. Efficient shift registers, LFSR counters, and long pseudo-random sequence generators. Technical report, Xilinx, Inc., 1996.

[4] Alpha Data. *ADM-XRC SDK User Guide 4.3.1*, 2003.

[5] Altera Corporation. *Linear Feedback Shift Register Megafunction : Solution Brief 11*, 1996.

[6] Altera Corporation. *Nios Embedded Processor : Software Development Reference Manual*, 2003.

[7] Altera Corporation. *Cyclone II Device Handbook, Volume 1*, 2005.

[8] Altera Corporation. *Stratix II Device Handbook, Volume 1*, 2005.

[9] A. L. Ananda, B. H. Tay, and E. K. Koh. A survey of asynchronous remote procedure calls. *ACM SIGOPS Operating Systems Review*, 26(2):92–109, 1992.

[10] R. Andraka and R. Phelps. An FPGA based processor yields a real time high fidelity radar environment simulator. In *Conference on Military and Aerospace Applications of Programmable Devices and Technologies*, 1998.

[11] Annapolis Micro Systems, Inc. *WILDSTAR-II Host Software Reference Manual*, 2005.

[12] K. Beck. *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[13] D. Beckett. *RDF/XML Syntax Specification (Revised) (W3C Recommendation)*. W3C, 2004.

[14] J. R. Bell. Algorithm 334: Normal random deviates. *Communications of the ACM*, 11(7):498, 1968.

[15] B. Bergen and P. Donnelly. *Inside DirectX*. Microsoft Press, Redmond, WA, 1998.

[16] T. Berners-Lee. *Notation 3 : An RDF language for the Semantic Web*, 2001.

[17] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

[18] J. D. Bingham and M. Serra. Solving hamiltonian cycle on FPGA technology via instance to circuit mappings. In *Workshop on Engineering of Reconfigurable Hardware/Software Objects*. Springer, 2000.

[19] F. Black and M. S. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economics*, 81:637–659, 1973.

[20] E. Boutillon, J.-L. Danger, and A. Ghazel. Design of high speed AWGN communication channel emulator. *Analog Integrated Circuits and Signal Processing*, 34(2):133–142, 2003.

[21] V. M. Bove and J. A. Watlington. Cheops: A reconfigurable data-flow system for video processing. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(2):140–149, 1995.

[22] M. Bowen. *RC1000-PP Function Reference Manual*. Celoxica Ltd., 2000.

[23] D. Box. *Essential COM*. Addison-Wesley Professional, first edition, 1997.

[24] G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *Annals of Mathematical Statistics*, 29(2):610–611, 1958.

[25] S. Breitinger, R. Loogen, and Y. Ortega-Mallen. Towards a declarative language for parallel and concurrent programming. In D. Turner, editor, *Functional Programming*, Workshops in Computing, Glasgow, 1995. Springer.

[26] R. P. Brent. Uniform random number generators for supercomputers. In *Australian Supercomputer Conference*, pages 95–104, 1992.

[27] R. P. Brent. On the periods of generalized fibonacci recurrences. *Mathematics of Computation*, 63(207):389–401, 1994.

[28] R. P. Brent. A fast vectorised implementation of Wallace's normal random number generator. Technical Report TR-CS-97-07, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia, Apr. 1997.

[29] R. P. Brent. Some comments on C. S. Wallace's random number generators. To appear in The Computer Journal, 2003.

[30] R. P. Brent. Note on Marsaglia's xorshift random number generators. *Journal of Statistical Software*, 11(5):1–4, 2004.

[31] D. Brickley and R. Guha. *RDF Vocabulary Description Language 1.0: RDF Schema (W3C Recommendation)*. W3C, 2004.

[32] D. Brickley and L. Miller. *FOAF Vocabulary Specification*, 'pages about things' edition, 2005.

[33] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *International Journal in Computer Simulation*, 4(2):155–182, 1993.

[34] B. Bullis and E. Sanders. *Adobe Premiere 6.5 Software Development Kit*. Adobe Systems Incorporated, 2002.

[35] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[36] J. A. Buzacott and J. G. Shanthikumar. *Stochastic Models of Manufacturing Systems*. Prentice Hall Professional Technical Reference, 1992.

[37] D. Caporossi. FPGA designers are heavyweights too. *Electronic Design*, August 2003.

[38] Celoxica Ltd. *Handel-C Language Reference*, 1999.

[39] Celoxica Ltd. *Platform Developer's Kit : DSM Manual*, 2005.

[40] Celoxica Ltd. *Platform Developer's Kit : PAL Manual*, 2005.

[41] S. Chappell. RC1000 performance: DMA v memory mapping performance on the RC1000. Technical report, Celoxica Ltd., 2001.

[42] J. Chen, J. Moon, , and K. Bazargan. Reconfigurable readback-signal generator based on a field-programmable gate array. *IEEE Transactions on Magnetics*, 40(3):1744–1750, 2004.

[43] J. Chen, J. Moon, and K. Bazargan. A reconfigurable FPGA-based readback signal generator for hard-drive read channel simulator. In *Design Automation Conference*, pages 349–354, New York, USA, 2002. ACM Press.

[44] R. C. Cheng. The use of antithetic control variates in computer simulations. In *Winter Simulation Conference*, pages 313–318, Piscataway, NJ, USA, 1981. IEEE Press.

[45] J. Clark. Introducing generics in the CLR. *MSDN Magazine*, September 2003.

[46] P. C. Clements. A survey of architecture description languages. In *International Workshop on Software Specification and Design*, page 16, Washington, DC, USA, 1996. IEEE Computer Society.

[47] P. D. Coddington. Random number generators for parallel computers. *The NHSE Review*, 1(2), 1996.

[48] A. Compagner and A. Hoogland. Maximum-length sequences, cellular automata, and random numbers. *Journal of Computational Physics*, 71(2):391–428, 1987.

[49] C. A. Constantinides, A. Bader, T. H. Elrad, P. Netinant, and M. E. Fayad. Designing an aspect-oriented framework in an object-oriented environment. *ACM Computing Surveys*, 32(1es):41, 2000.

[50] R. L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, 1986.

[51] R. Cooper. *UltraSONIC API Reference*. Sony Broadcast & Professional Europe, third edition, 2002.

[52] C. Courbis and A. Finkelstein. Towards aspect weaving applications. In *International Conference on Software Engineering*, pages 69–77, New York, USA, 2005. ACM Press.

[53] J. G. F. Coutinho and W. Luk. Source-directed transformations for hardware compilation. In *International Conference on Field-Programmable Technology*, pages 278–285. IEEE Computer Society Press, 2003.

[54] C. P. Cowen and S. Monaghan. A reconfigurable Monte-Carlo clustering processor (MCCP). In D. A. Buell and K. L. Pocek, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 59–65, Napa, CA, 1994. IEEE Computer Society Press.

[55] D. R. Cox and W. L. Smith. *Queues*. Methuen's Monographs on Statistical Subjects. Chapman and Hall, 1991.

[56] G. Czajkowski. Application isolation in the Java virtual machine. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 354–366, New York, USA, 2000. ACM Press.

[57] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag Berlin Heidelberg, 2002.

[58] J.-L. Danger, A. Ghazel, E. Boutillon, and H. Laamari. Efficient FPGA implementation of gaussian noise generator for communication channel emulation. In *IEEE International Conference on Electronic Circuits and Systems*, pages 366–368. IEEE Computer Society Press, 2000.

[59] B. Dawes and D. Abraham. *Boost C++ libraries*. `http://www.boost.org`, 2005.

[60] D. Denning, I. J, and M. Devlin. A key agile 17.4Gbit/sec camellia implementation. In *International Conference on Field Programmable Logic and Applications*, pages 546–554. Springer-Verlag, Berlin, 2004.

[61] L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag New York, 1996.

[62] J. Diaz, E. Ros, S. Mota, R. Carrillo, and R. Agis. Real time optical flow processing system. In *International Conference on Field Programmable Logic and Applications*, pages 617–626. Springer-Verlag, Berlin, 2004.

[63] S. Duplichan. PPSearch : A primitive polynomial search program. http://users2.ev1.net/ sduplichan/primitivepolynomials/.

[64] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, 1998.

[65] K. Entacher. Bad subsequences of well-known linear congruential pseudorandom number generators. *ACM Transactions on Modelleling and Computer Simulation*, 8(1):61–70, 1998.

[66] H. Epsom. *UltraSONIC Developers Guide*. Sony Broadcast & Professional Europe, first edition, 2002.

[67] M. Feeley and J. S. Miller. A parallel virtual machine for efficient scheme compilation. In *ACM Conference on LISP and Functional Programming*, pages 119–130, New York, USA, 1990. ACM Press.

[68] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, second edition, 1990.

[69] C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, 2003.

[70] B. N. Freeman-Benson. Kaleidoscope: mixing objects, constraints, and imperative programming. In *European Conference on Object-Oriented Programming*, pages 77–88, New York, USA, 1990. ACM Press.

[71] V. Frost, W. J. LaRue, and K. Shanmugan. Efficient techniques for the simulation of computer communications networks. *IEEE Journal on Selected Areas in Communications*, 6:146–157, 1988.

[72] E. Gamma, R. Helm, R. Johnson, and J. Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[73] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Network Parallel Computing*. The MIT Press, 1994.

[74] D. Geist, G. Biran, T. Arons, M. Slavkin, Y. Nustov, M. Farkas, K. Holtz, A. Long, D. King, and S. Barret. A methodology for the verification of a "system on chip". In *Design Automation Conference*, pages 574–579, New York, USA, 1999. ACM Press.

[75] M. George and P. Alfke. Linear feedback shift registers in virtex devices. Technical report, Xilinx, Inc., 2001.

[76] R. S. Germain, B. Fitch, A. Rayshubskiy, M. Eleftheriou, M. C. Pitman, F. Suits, M. Giampapa, and T. C. Ward. Blue matter on blue gene/L: massively parallel computation for biomolecular simulation. In *EEE/ACM/IFIP International Conference on Hardware/software Codesign and System Synthesis*, pages 207–212, New York, USA, 2005. ACM Press.

[77] The GNU Project. *The GNU Scientific Library*, 2005.

[78] M. Goresky and A. Klapper. Efficient multiply-with-carry random number generators with maximal period. *ACM Transactions on Modeling and Computer Simulations*, 13(4):310–321, 2003.

[79] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification*. Addison-Wesley, second edition, 2000.

[80] G. Govindu, L. Zhuo, S. Choi, and V. K. Prasanna. Analysis of high-performance floating-point arithmetic on FPGAs. In *International Parallel and Distributed Processing Symposium*, pages 149–152. IEEE Computer Society Press, 2004.

[81] J. Grant and D. Beckett. *RDF Test Cases (W3C Recomendation)*. W3C, 2004.

[82] C. Gross. *Building COM Components on UNIX*. Microsoft, 1998.

[83] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *International Conference on VLSI Design*, page 461. IEEE Computer Society Press, 2003.

[84] T. Hardy, M. Jamel Deen, and R. Murowinski. Low-frequency noise in proton damaged LDD MOSFET's. *IEEE Transactions on Electron Devices*, 46:1339–1346, 1999.

[85] W. L. Harrison III and Z. Ammarguella. The Design of Automatic Parallelizers for Symbolic and Numeric Programs. In *Parallel Lisp: Languages and Systems*, volume 441, pages 235–253, Sendai, Japan, June 5–8, 1988. Springer, Berlin.

[86] P. H. Hartel and L. Moreau. Formalizing the safety of Java, the Java virtual machine, and Java card. *ACM Computing Surveys*, 33(4):517–558, 2001.

[87] P. Hayes. *RDF Semantics (W3C Recommendation)*. W3C, 2004.

[88] S. D. Haynes, J. Stone, P. Y. K. Cheung, and W. Luk. Video image processing with the Sonic architecture. *IEEE Computer*, 33(4):50–57, 2000.

[89] D. Heath, R. Jarrow, and A. Morton. Bond pricing and the term structure of interest rates: A new methodology for contingent claims valuation. *Econometrica*, 60(1):77–105, 1992.

[90] P. Hellekalek. Don't trust parallel monte carlo! In *Proceedings of the Workshop on Parallel and Distributed Simulation*, pages 82–89, Washington, DC, USA, 1998. IEEE Computer Society.

[91] W. Hörmann. A note on the quality of random variates generated by the ratio of uniforms method. *ACM Transactions on Modeling and Computer Simululation*, 4(1):96–106, 1994.

[92] W. Hörmann and J. Leydold. Continuous random variate generation by fast numerical inversion. *ACM Transactions on Modeling and Computer Simulation*, 13(4):347–362, 2003.

[93] P. D. Hortensius, R. D. McLeod, and H. C. Card. Parallel random number generation for VLSI systems using cellular automata. *IEEE Transactions on Computers*, 38(10):1466–1473, 1989.

[94] P. D. Hortensius, R. D. McLeod, and B. W. Podaima. Cellular automata circuits for built-in self test. *IBM Journal of Research and Development*, 34(2-3):389–405, 1990.

[95] L. Huelsbergen. A representation for dynamic graphs in reconfigurable hardware and its application to fundamental graph algorithms. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 105–115, New York, USA, 2000. ACM Press.

[96] IEEE. *IEEE Standard 1076: VHDL Language Reference Manual*, 1991.

[97] IEEE. *IEEE standard Specifications for the Implementation of 8X8 Inverse Discrete Cosine Transformers*, 1997.

[98] Intel Corp. *IA-32 Intel Architecture Software Developers Manual*, 2001.

[99] Intel Corp. *Intel Pentium 4 Processor Optimization : Reference Manual*, 2001.

[100] Internet Engineering Task Force. *The Atom Syndication Format*, internet-draft edition, 2005.

[101] ISO/IEC. IT – security techniques – hash-functions – part 3: Dedicated hash-functions. International Standard ISO/IEC DIS 10118-3, International Standards Organization, 1997.

[102] M. Izatt, P. Chan, and T. Brecht. Ajents: towards an environment for parallel, distributed and mobile Java applications. In *ACM Java Grande Conference*, pages 15–24, New York, USA, 1999. ACM Press.

[103] J. Jacky. Specifying a safety-critical control system in Z. *IEEE Transactions on Software Engineering*, 21(2):99–106, Feb. 1995.

[104] M. Jeruchim. Techniques for estimating the bit error rate in the simulation of digital communication systems. *IEEE Journal on Selected Areas in Communications*, 2:153–170, 1984.

[105] C. W. Johnson. Literate specification: Using design rationale to support formal methods in the development of human-machine interfaces. *Human-Computer Interaction*, 11(4):291–320, 1996.

[106] A. Kageyama, M. Kameyama, S. Fujihara, M. Yoshida, M. Hyodo, and Y. Tsuda. A 15.2 TFlops simulation of geodynamo on the earth simulator. In *ACM/IEEE Conference on Supercomputing*, page 35, Washington, DC, USA, 2004. IEEE Computer Society.

[107] H. J. Kim and W. H. Mangione-Smith. Factoring large numbers with programmable hardware. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 41–48, New York, USA, 2000. ACM Press.

[108] A. J. Kinderman and J. F. Monahan. Computer generation of random variables using the ratio of uniform deviates. *ACM Transactions on Mathematical Software*, 3(3):257–260, 1977.

[109] R. Knop. Remark on algorithm 334 [g5]: Normal random deviates. *Communications of the ACM*, 12(5):281, 1969.

[110] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1981.

[111] I. Kraljic, G. Qunot, and B. Zavidovique. From real-time emulation to ASIC integration for image processing applications. *EEE Transactions on VLSI Systems*, pages 391–404, 1996.

[112] A. S. P. Kumar. Prototype and validation platform for Leon based multiprocessor socs. Master's thesis, Department of Computer Science & Engineering, Indian Institute of Technology Delhi, 2002.

[113] Y.-S. Kwon and C.-M. Kyung. Performance-driven event-based synchronization for multi-FPGA simulation accelerator with event time-multiplexing bus. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1444–1456, 2005.

[114] D. E. Lackey, P. S. Zuchowski, and J. Koehl. Designing mega-ASICs in nanogate technologies. In *Design Automation Conference*, pages 770–775, New York, NY, USA, 2003. ACM Press.

[115] D. Lampret. *OpenRISC 1000 Architecture Manual*. OpenCores, 2005.

[116] J. Lawrence, S. Rusinkiewicz, and R. Ramamoorthi. Efficient BRDF importance sampling using a factored representation. *ACM Transactions on Graphics*, 23(3):496–505, 2004.

[117] P. Leach, M. Mealling, and R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace (Request for Comments)*. The Internet Engineering Task Force, 2005.

[118] P. L'Ecuyer. Efficiency improvement and variance reduction. In *Winter Simulation Conference*, pages 122–132, San Diego, CA, USA, 1994. Society for Computer Simulation International.

[119] P. L'Ecuyer. *Handbook on Simulation : Random Number Generation*, chapter 4. Wiley, 1998.

[120] P. L'Ecuyer. Tables of maximally equidistributed combined LFSR generators. *Mathematics of Computation*, 68(225):261–269, 1999.

[121] P. L'Ecuyer and R. Simard. TestU01 random number test suite. `http://www.iro.umontreal.ca/~simardr/indexe.html`.

[122] A. Lee. *VirtualDub Filter SDK*. `http://virtualdub.org`, 2005.

[123] D.-U. Lee, W. Luk, J. D. Villasenor, and P. Y. Cheung. A gaussian noise generator for hardware-based simulations. *IEEE Transactions On Computers*, 53(12):1523–1534, december 2004.

[124] D.-U. Lee, W. Luk, J. D. Villasenor, G. Zhang, and P. H. Leong. A hardware gaussian noise generator using the wallace method. *IEEE Transactions on VLSI Systems*, 13(8):911–920, 2005.

[125] J. Lee, L. F. Lai, and W. T. Huang. Task-based specifications through conceptual graphs. *IEEE Expert: Intelligent Systems and Their Applications*, 11(4):60–70, 1996.

[126] 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 *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 170–179. IEEE Computer Society Press, 2001.

[127] A. Lerner. Graphics: Manipulate digital images in internet explorer with the DirectX transform SDK. *MSDN Magazine*, March 2001.

[128] J. Leydold. Short universal generators via generalized ratio-of-uniforms method. *Mathematics of Computation*, 72(243):1453–1471, 2003.

[129] P. Litwinowicz. Processing images and video for an impressionist effect. In *ACM Conference on Computer Graphics and Interactive Techniques*, pages 407–414, New York, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[130] E. Lord, M. Devlin, N. Harold, and C. Sanderson. Accelerating BER analysis using an FPGA based processing platform. In *IEEE Military Communications Conference*, pages 218–223. IEEE Computer Society Press, 2002.

[131] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat. An efficient implementation of Java's remote method invocation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 173–182, New York, USA, 1999. ACM Press.

[132] T. Mackinnon, S. Freeman, and P. Craig. Endotesting: Unit testing with mock objects. In *eXtreme Programming and Flexible Processes in Software Engineering*, 2000.

[133] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 3–14, New York, USA, 1996. ACM Press.

[134] S. A. Maidanov. Monte carlo european options pricing implementation using various industry library solutions. Technical report, Intel Corp., 2003.

[135] G. Marsaglia. A current view of random number generators. In *Computer Science and Statistics Symposium*, pages 3–10, 1985.

[136] G. Marsaglia. The Diehard random number test suite. `http://stat.fsu.edu/pub/diehard/`, 1997.

[137] G. Marsaglia. Multiply-with-carry (MWC) generators. Document included with Diehard package (`http://stat.fsu.edu/pub/diehard/`), 1997.

[138] G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 2003.

[139] G. Marsaglia and J. Marsaglia. Evaluating the Anderson-Darling distribution. *Journal of Statistical Software*, 9(2):1–5, Feb. 2004.

[140] G. Marsaglia and W. W. Tsang. A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions. *SIAM Journal on Scientific and Statistical Compututing*, 5(2):349–359, June 1984.

[141] G. Marsaglia and W. W. Tsang. The ziggurat method for generating random variables. *Journal of Statistical Software*, 5(8):1–7, 2000.

[142] P. Martin. An analysis of random number generators for a hardware implementation of genetic programming using FPGAs and Handel-C. Technical report, University of Essex, 2002.

[143] T. Maruyama. Real-time computation of the generalized hough transform. In *International Conference on Field Programmable Logic and Applications*, pages 980–985. Springer-Verlag, Berlin, 2004.

[144] The MathWorks. *Simulink Reference*, 2005.

[145] R. Matousek, M. Tichy, Z. Pohl, J. Kadlec, C. Softley, and N. Coleman. Logarithmic number system and floating-point arithmetics on FPGA. In *International Conference on Field Programmable Logic and Applications*, pages 627–636, London, UK, 2002. Springer-Verlag.

[146] M. Matsumoto and Y. Kurita. Twisted GFSR generators. *ACM Transactions on Modeling and Computer Simulation*, 2(3):179–194, 1992.

[147] M. Matsumoto and Y. Kurita. Twisted GFSR generators II. *ACM Transactions on Modeling and Computer Simulation*, 4(3):254–266, 1994.

[148] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, Jan. 1998.

[149] D. Mattson. Evaluation of synthesizable CPU cores. Master's thesis, Chalmers University of Technology, Gothenburg, 2004.

[150] E. M. Maximilien and L. Williams. Assessing test-driven development at ibm. In *International Conference on Software Engineering*, pages 564–569, Washington, DC, USA, 2003. IEEE Computer Society.

[151] J. M. McCollum, J. M. Lancaster, D. W. Bouldin, and G. D. Peterson. Hardware acceleration of pseudo-random number generation for simulation applications. In *IEEE Southeastern Symposium on System Theory*, pages 299–303, 2003.

[152] E. Meijer and C. Szyperski. Overcoming independent extensibility challenges. *Communications of the ACM*, 45(10):41–44, 2002.

[153] O. Mencer. PAM-Blox II: design and evaluation of c++ module generation for computing with FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 67–76. IEEE Computer Society Press, 2002.

[154] O. Mencer, Z. N. Huang, and L. Huelsbergen. HAGAR: efficient multi-context graph processors. In *International Conference on Field Programmable Logic and Applications*, pages 915–924. Springer-Verlag Berlin, 2002.

[155] O. Mencer, D. J. Pearce, L. W. Howes, and W. Luk. Design space exploration with A Stream Compiler. In *International Conference on Field-Programmable Technology*, pages 270–277. IEEE Computer Society Press, 2003.

[156] N. Metropolis and S. Ulam. The monte carlo method. *Journal of the American Statistical Association*, 44:335–341, 1949.

[157] M. M. Meysenburg and J. A. Foster. Random generator quality and GP performance. In *International Conference on Genetic and Evolutionary Computation*, pages 1131–1126. ACM Press, 1999.

[158] Microsoft. *ECMA-335: Common Language Infrastructure*. European Computer Manufacturers Association, 2002.

[159] Microsoft Corp. *VB.NET Language Specification*, 2000.

[160] Microsoft Corp. *Microsoft DirectShow 9.0 SDK : DirectShow Reference*, 2005.

[161] L. Moll and M. Shand. Systems performance measurement on PCI pamette. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 125–133, Napa, CA, April 1997.

[162] G. E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics Magazine*, 38(8):114–117, April 19 1965.

[163] C. C. Morgan and B. A. Sufrin. Specification of the Unix filing system. *IEEE Transactions on Software Engineering*, 10(2):128–142, Mar. 1984.

[164] T. H. Myer and I. E. Sutherland. On the design of display processors. *Communications of the ACM*, 11(6):410–414, 1968.

[165] S. Nahar, S. Sahni, and E. Shragowitz. Simulated annealing and combinatorial optimization. In *Design Automation Conference*, pages 293–299, Piscataway, NJ, USA, 1986. IEEE Press.

[166] A. Negoi and J. Zimmermann. Monte Carlo hardware simulator for electron dynamics in semiconductors. In *International Annual Semiconductor Conference*, pages 557–560, Sinaia, Romania, 1996.

[167] R. E. Noonan and R. H. Prosl. Unit testing frameworks. In *SIGCSE Technical Symposium on Computer Science Education*, pages 232–236, New York, USA, 2002. ACM Press.

[168] Nullsoft. *Winamp Software Development Kit*, 2004.

[169] The Open Group. *DCE 1.1: Remote Procedure Call*, 1997.

[170] S. K. Park and K. W. Miller. Random number generators: good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.

[171] E. K. Pauer, P. D. Fiore, and J. M. Smith. Algorithm analysis and mapping environment for adaptive computing systems: Further results. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 264–265. IEEE Computer Society Press, 1999.

[172] PCI SIG. *PCI Local Bus Specifications (Revision 2.2)*, 1998.

[173] K. Pearson. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can reasonably supposed to have arisen from random sampling. *Philosophical Magazine, Series 5*, 50:157–175, 1900.

[174] K. Periyasamy and J. Chidambaram. Software reuse using formal specification of requirements. In *Conference of the Centre for Advanced Studies on Collaborative Research*, page 31. IBM Press, 1996.

[175] M. Pierce, G. Fox, C. Youn, S. Mock, K. Mueller, and O. Balsoy. Interoperable web services for computational portals. In *ACM/IEEE Conference on Supercomputing*, pages 1–12, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[176] J. L. Pino, E. A. L. S. Ha, and J. T. Buck. Software synthesis for DSP using ptolemy. *Journal on VLSI Signal Processing*, 9(1):7–21, 1995.

[177] M. Pinto, L. Fuentes, and J. M. Troya. DAOP-ADL: an architecture description language for dynamic component and aspect-based development. In *International Conference on Generative Programming and Component Engineering*, pages 118–137, New York, USA, 2003. Springer-Verlag New York, Inc.

[178] F. Plasil. Enhancing component specification by behavior description: the SOFA experience. In *International Symposium on Information and Communication Technologies*, pages 185–190. Trinity College Dublin, 2005.

[179] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Prentice Hall International Series in Computer Science. Prentice-Hall, 1991.

[180] D. K. Pradhan and M. Chatterjee. GLFSR - a new test pattern generator for built-in-self-test. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(2):238–247, 1999.

[181] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1997.

[182] R. Pucella. Towards a formalization for COM part I: the primitive calculus. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 331–342, New York, USA, 2002. ACM Press.

[183] QEMU. *QEMU Internals*, 2005.

[184] G. Qunot, I. Kraljic, J. Serot, and B. Zavidovique. A reconfigurable compute engine for real-time vision automata prototyping. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 91–100. IEEE Computer Society Press, 1994.

[185] M. Reaz, S. Z. Islam, M. A. M. Ali, and M. Sulaiman. FPGA realization of back-propagation for stock market prediction. In *International Conference on Neural Information Processing*, volume 2, pages 960–964, 2002.

[186] L. M. Reyneri, F. Cucinotta, A. Serra, and L. Lavagno. A hardware/software co-design flow and IP library based on simulink. In *Design Automation Conference*, pages 593–598, New York, USA, 2001. ACM Press.

[187] B. D. Ripley. *Stochastic Simulation*. Wiley Series in Probability and Statistics. Wiley, 1987.

[188] M. N. O. Sadiku and M. Ilyas. *Simulation of Local Area Networks*. CRC Press, 1994.

[189] D. Schmitz, P. Khosla, and T. Kanade. The CMU reconfigurable modular manipulator system. Technical Report CMU-RI-TR-88-07, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 1988.

[190] M. Schoeberl. JOP: A Java optimized processor. In Z. T. R. Meersman and D. Schmidt, editors, *Workshop on Java Technologies for Real-Time and Embedded Systems*, volume 2889 of *Lecture Notes in Computer Science*, pages 346–359, Catania, Italy, November 2003. Springer.

[191] S. D. Scott, S. Seth, and A. Samal. A synthesizable VHDL coding of a genetic algorithm. Technical Report Technical Report UNL-CSE-97-009, University of Nebraska-Lincoln, 1997.

[192] S. P. Seng, W. Luk, and P. Y. K. Cheung. Flexible instruction processors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 193–200, New York, USA, 2000. ACM Press.

[193] S. P. Seng, W. Luk, and P. Y. K. Cheung. Run-time adaptive flexible instruction processors. In *International Conference on Field Programmable Logic and Applications*, pages 545–555, London, UK, 2002. Springer-Verlag.

[194] R. Sethi, J. D. Ullman, and A. V. Aho. *Compilers. Principles, techniques, and tools.* Addison-Wesley, 1986.

[195] B. Shackleford, M. Tanaka, R. J. Carter, and G. Snider. FPGA implementation of neighborhood-of-four cellular automata random number generators. pages 106–112, New York, USA, 2002. ACM Press.

[196] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, 1995.

[197] D. Shreiner. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2.* Addison-Wesley Professional, third edition, 1999.

[198] J. Siegel. *CORBA 3 Fundamentals and Programming.* John Wiley & Sons, second edition, 2000.

[199] S. Singh and R. Slous. Accelerating Adobe Photoshop using the XC6200 FPGA. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 236–244, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[200] E. H. Spafford. The internet worm program: an analysis. *ACM Computer Communication Review*, 19(1):17–57, 1989.

[201] M. Steenstrup, M. A. Arbib, and E. G. Manes. Port automata and the algebra of concurrent processes. *Journal of Computer and System Sciences*, 27(1):29–50, 1983.

[202] D. B. Stewart, R. A. Volpe, and P. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. Technical Report CMU-RI-TR-93-11, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, July 1993.

[203] D. Stiliadis and A. Varma. FAST: a simulation testbed for ATM networks. In *IEEE International Computer Conference*, page 32, Washington, DC, USA, 1996. IEEE Computer Society.

[204] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, third edition, 1997.

[205] H. Styles and W. Luk. Customising graphics applications: Techniques and programming interface. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 77–90. IEEE Computer Society Press, 2000.

[206] C. Sweeney and B. Blyth. *RC1000-PP Hardware Reference Manual 2.2*. Celoxica Ltd., 2000.

[207] Synplicity Inc. *Synplify Reference Manual*, 2004.

[208] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.

[209] C. Szyperski. Component technology: what, where, and how? In *International Conference on Software Engineering*, pages 684–693, Washington, DC, USA, 2003. IEEE Computer Society.

[210] A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, 1996.

[211] O. Tammelin. *Jeskola Buzz*. `http://www.buzzmachines.com/`, 2000.

[212] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, fourth edition, 2002.

[213] S. Tezuka, P. L'Ecuyer, and R. Couture. On the lattice structure of the add-with-carry and subtract-with-borrow random number generators. *ACM Transactions on Modeling and Computer Simulation*, 3(4):315–331, 1993.

[214] D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic, 1991.

[215] F. Y. Tim Lindholm. *The Java Virtual Machine Specificaiton*. Addison-Wesley, 1996.

[216] D. Turner and I. Oeschger. *Creating XPCOM Components*. 2003.

[217] R. D. Turney, A. M. Reza, and J. G. R. Delva. FPGA implementation of adaptive temporal kalman filter for real time video filtering. http://www.xilinx.com/products/logicore/dsp, 1999.

[218] K. D. Underwood and K. S. Hemmert. Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 219–228. IEEE Computer Society Press, 2004.

[219] USB Implementers Forum, Inc. *Universal Serial Bus Revision 2.0 specification*, 2000.

[220] G. van Rossum. Python reference manual. Technical Report CS-R9525, Centrum voor Wiskunde en Informatica, 1995.

[221] J. A. Waicukauski and E. Lindbloom. Fault detection effectiveness of weighted random patterns. In *IEEE Test Conference*, pages 245–255. IEEE Computer Society Press, 1988.

[222] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly, third edition, 2000.

[223] C. S. Wallace. Fast pseudorandom generators for normal and exponential variates. *ACM Transactions on Mathematical Software*, 22(1):119–127, 1996.

[224] X. Wang and B. E. Nelson. Tradeoffs of designing floating-point division and square root on virtex FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines*, page 195. IEEE Computer Society Press, 2003.

[225] X. Wang and S. G. Ziavras. Parallel direct solution of linear equations on FPGA-based machines. In *International Conference on Parallel and Distributed Real-Time Systems*, page 8. IEEE Computer Society Press, 2003.

[226] X. Wang and S. G. Ziavras. Performance optimization of an FPGA-based configurable multiprocessor for matrix operations. In *International Conference on Field-Programmable Technology*, pages 303–306. IEEE Computer Society Press, 2003.

[227] X. Wang and S. G. Ziavras. Parallel LU factorization of sparse matrices on FPGA-based configurable computing engines. *Concurrency and Computation: Practice and Experience*, 16:319–343, 2004.

[228] P. Wegner. Concepts and paradigms of object-oriented programming. *SIGPLAN OOPS Messenger*, 1(1):7–87, 1990.

[229] M. J. Wichura. Algorithm AS 241: The percentage points of the normal distribution. *Applied Statistics*, 37(3):477–484, 1988.

[230] R. Wilson, N. Mokhoff, and N. Mokhoff. Design gap remains, even as ISD fizzles. *EETimes*, page 4, 2002.

[231] S. Wiltamuth and A. Hejlsberg. *C# Language Specification*. Microsoft Corp., 2000.

[232] M. J. Wirthlin and B. L. Hutchings. DISC: the dynamic instruction set computer. In J. Schewel, editor, *FPGAs for Fast Board Development and Reconfigurable Computing*, pages 92–103, Bellingham, WA, 1995. SPIE – The International Society for Optical Engineering.

[233] M. J. Wirthlin, B. L. Hutchings, and K. L. Gilson. The nano processor: A low resource reconfigurable processor. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 23–30, Los Alamitos, CA, 1994. IEEE Computer Society Press.

[234] S. Wolfram. Random sequence generation by cellular automata. *Advances in Applied Maths*, 7(2):123–169, 1986.

[235] S. Woop, J. Schmittler, and P. Slusallek. Rpu: A programmable ray processing unit for realtime ray tracing. In *International Conference on Computer Graphics and Interactive Techniques*. ACM Press/Addison-Wesley Publishing Co., July 2005.

[236] I. Xilinx. Additive white gaussian noise (AWGN) core. CoreGen documentation file, 2002.

[237] Xilinx, Inc. *Virtex-II Platform FPGAs: Complete Data Sheet*, 2000.

[238] Xilinx, Inc. *Lightfoot 32-bit Java Processor Core Datasheet*, 2002.

[239] Xilinx, Inc. *Virtex-II Pro Platform FPGAs: Complete Data Sheet*, 2002.

[240] Xilinx, Inc. *CORE Generator Guide*, 2003.

[241] Xilinx, Inc. *Spartan-3 1.2V FPGA Family: Complete Data Sheet*, 2003.

[242] Xilinx, Inc. *Virtex-4 Family Overview*, 2004.

[243] Xilinx, Inc. *MicroBlaze Processor Reference Guide*, 2005.

[244] Xilinx, Inc. *Virtex-4 Embedded Tri-Mode Ethernet MAC User Guide*, 2005.

[245] Xilinx, Inc. *Virtex-4 RocketIO Multi-Gigabit Transceiver: User Guide*, 2005.

[246] Xilinx, Inc. *Xilinx System Generator v7.1 User Guide*, 2005.

[247] Xilinx, Inc. *XtremeDSP for Virtex-4 FPGAs User Guide*, 2005.

[248] W.-N. Yang and W.-W. Liou. Combining antithetic variates and control variates in simulation experiments. *ACM Transactions on Modeling and Computer Simulation*, 6(4):243–260, 1996.

[249] M. Yoshimi, Y. Osana, T. Fukushima, and H. Amano. Stochastic simulation for biochemical reactions on FPGA. In *International Conference on Field Programmable Logic and Applications*, pages 105–114. Springer-Verlag, Berlin, 2004.

[250] G. L. Zhang, P. H. Leong, D.-U. Lee, J. D. Villasenor, R. C. Cheung, and W. Luk. Ziggurat-based hardware gaussian random number generator. To appear at FPT 05, 2005.

[251] G. L. Zhang, P. H. W. Leong, C. H. Ho, K. H. Tsoi, D.-U. Lee, R. C. C. Cheung, and W. Luk. Reconfigurable acceleration for monte carlo based financial simulation. In *International Conference on Field-Programmable Technology*, pages 215–224. IEEE Computer Society Press, 2005.

[252] L. Zhuo and V. K. Prasanna. Sparse matrix-vector multiplication on FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 63–74, New York, USA, 2005. ACM Press.

Appendix A

IGOL Operation Specification Schema

Operation Specifications are the way that data processing components in IGOL are described, incorporating both structural and functional characterisics (see Section 3.2.1). The schema used to describe these specifications is based on a simple method for describing properties of specifications, along with some production rules that are used to expand these specifications in cases where specifications are derived from other specifications through extension or restriction.

A.1 Operation Specification Syntax

Data within the schema is represented using a very simple syntax for making statements about objects, using the following syntax shown in BNF (Backus Naur Form) [194]:

```
<statement> ::= <relation> | <object> ';'
<relation> ::= <id> '=' ( ``<text>'' | <number> | <object> ) ;
<object> ::= <id> ( '{' <statement> * '}' ) ?
<full-id> ::= <id-part> ( ':' <id-part> ) *
<local-id> ::= ( ':' <id-part> ) *
<id-part> ::= [a-zA-Z0-9-/] | '*'
```

An example of statements made in this syntax are:

```
subject1 {
relation1 = object1 ;
relation3 = part1 {
  relation2 = object2 ;
} ;
```

```
  relation4 = subject1:* ;
};
```

This fragment makes two statements about *subject1* and one statement about an object within *subject1*, called *subject1:part1*. When objects are declared within other objects in this way the identifier of the enclosing object is prefixed to that of the internal object, so the above fragment is equivalent to:

```
subject1 {
  relation1 = object1 ;
  relation3 = :part1 ;
  relation4 = subject1:* ;
};
subject1:part1 {
  relation2 = object2 ;
};
```

The identifier *part1*, used in *relation3* uses a local identifier, indicating that the identifier should be prefixed with the containing identifier to get the full identifier.

The information contained in this syntax can also be read as triples of the form *(subject,relation,object)*, so the examples above expand to:

```
(subject1, relation1, object1)
(subject1, relation3, subject1:part1)
(subject1, relation4, subject1:*)
(subject1:part1, relation2, object2)
```

Asterisks within identifiers are wildcards, and will match any other identifier that has all the same non-wildcard parts. For example *object1:*:ex* matches *object1:part2:ex* and *object1:part3:ex*, but not *object1:part1:te*. When an identifier ends in a wildcard, the wildcard can match any number of id parts as long as all preceding non-wildcard components are the same, so *object1:** can match *object1:part1*' and *object1:part1:a*. Wherever a wildcard identifier appears in a set of statements the associated triple will be expanded for matching identifiers, so for example the above example also implicitly contains the statement

```
(subject1, relation4, subject1:part1)
```

as *subject1:part1* is used in a non-wildcard form elsewhere in the description.

A.2 Specification Schema

Certain types of relation have special meaning to IGOL, and identify that statements are being made about a specific type of components. These can be split into different classes of information, and the most critical within each class are shown below.

A.2.1 Specification Relationships

(*a*, **type**, *b*) : Identifies the type of object *a* as the type corresponding to identifier *b*. Well known types include (but are not limited to) **opSpec**, **opImpl**, and **input**, **output**, **param**, and **funcSpec**.

(*a*, **extends**, *b*) : Indicates that object specification *a* is an extension (or sub-type) of *b*. Both *a* and *b* should have type **opSpec**.

(*a*, **implements**, *b*) : Indicates that object *a* implements specification *b*, where *a* has type **opImpl** and *b* has type **opSpec**.

(*a*, **restricts**, *b* : Indicates that object *a* is a restriction (i.e. a partial implementation) of specification *b*, where *a* has type **opSpec** and *b* has type **opSpec**.

(*a*, **restriction**, *b* : Expresses a restriction applied to object *a* during the expansion of a partial specification.

A.2.2 Port Structure

In order for applications to interact with components at run-time specifications must provide ports to represent inputs, outputs and parameters.

(*a*, **input**, "*b*") : Identifies object *b* as an input port of object *a*. Similar relations **output** and **param** identify output and parameter ports.

(*a*, **minInputCount**, *i*) : Specifies the minimum number of ports of each type that are allowed under component specification *a* is the integer *i*. **minOutputCount** and **minParamCount** similarly restrict output and parameter ports.

(*a*, **inputCount**, *i*) : Specifies the exact number of input ports of each type that the component must have. If both are present this must be compatible with the **minInputCount**. **outputCount** and **paramCount** restrict output and parameter port counts.

(*a*, **dataType**, *dt*) : Specifies the data-type of port *a* as *dt*, for example **RGB24** or **video**.

(*a*, **minValue**, *r*) : Specifies the the minimum value that values on port *a* can take. This is used to restrict the values of parameters, such as specifying that a parameter that represents an angle must be between 0 and 360. Corresponding constraint **maxValue** restricts the maximum value allowed.

A.2.3 Functionality

Specifications need to contain functionality both in order to indicate how to create an Operation Implementation, but also to allow functional specifications within Operation Specifications. The functionality may be embedded as source code within the specification that

can be compiled, or as activation information that allows the creation or loading of a component. Possible properties are not limited to those below, and developers are free to attach other information (as long as the relation identifiers are globally unique).

(*a*, **function,** "*b*") : Identifies a **funcSpec** *b* the defines the expected functionality of **opSpec** *a*.

(*a*, **platform,** "*b*") : Identifies the platform that **funcSpec** *a* uses, such as: **native**, which is raw C++ that can be compiled and loaded; **COM** or **dotNET**, which indicates that the component can be instantiated and loaded via the COM or .NET frameworks; or the name of a hardware platform such as **rc1000/xcv2000e** or **ultrasonic/xcv1000e**.

(*a*, **format,** "*b*") : Identifies the format or language that the **funcSpec** *a* is described in. This is needed in addition to the platform, as there are often many choices of format for each platform, such as different types of source code, and different levels of compilation. Typical values include **cpp**, **csharp**, **handelc**, **assembly** and **bitfile**.

(*a*, **activationInfo,** "*b*") : For components using the COM or .NET frameworks this contains the information needed to instantiate the component, for example the GUID [117] for COM, or the fully qualified type name of the .NET class.

(*a*, **source,** "*b*") : This is used to hold source code when it is embedded directly into the specification. Used mostly for Operation Specifications rather than Operation Implementations.

(*a*, **path,** "*b*") : Where the functionality is not contained in source form within the specification, this points to the external file, such as an assembly or hardware configuration.

A.2.4 General Purpose

Many applications need to display information to users about available options and parameters. By annotating the Operation Specifications the applications can provide a better user experience by providing friendly names and using specialised methods for controlling certain parameters.

(*a*, **name,** "*b*") : Provides a human-readable display name of object *a* for use in user interfaces.

(*a*, **description,** "*b*") : Provides a textual description of object *a* for use in user interfaces.

(*a*, **hidden,** "*b*") : Suggests to the application that the given object should not be displayed in user-interfaces, for example a parameter port that should only be adjusted under programmatic control.

(*a*, **stereotype,** *b*) : Provides hints to applications as to how a port should be treated, particularly with respect to user interfaces for parameter control.

A.3 Specification Extension and Restriction Mechanism

To allow for Operation Specification extension and restriction it is necessary to have a well defined mechanism for determining the properties of derived Operation Specifications. For example, consider the following two Operation Specifications:

```
videoEffect{ type = opSpec;
  :in:* { dataType = video ; }
  :out:* { dataType = video ; }
  inputMinCount = 1 ;     outputMinCount = 1 ;
}

videoTransform{
  extends =  videoEffect ;
  input = :in:input ;     output = :out:output ;
  inputCount = 1 ;     outputCount = 1;
}
```

The specification of *videoTransform* implies some rules that are not explicitly present, due its extension relationship with *videoEffect*. In order to discover all properties of videoTransform certain production rules are applied, which apply all constraints of videoEffect to videoTransform. These constraints can be determined by applying the following production rules to the currently known statements:

$$\forall\, child, parent, pred : (child, \mathbf{extends}, parent) \wedge (parent, pred, x) \implies (child, pred, x) \quad \text{(A.1)}$$

Which replicates all statements directly linked to the base specifications, such as the assertions about $inputCountMin$ and $outputCountMin$. A second rule does the same for all statements about parts of the specification:

$$\forall\, child, parent, pred, part : (child, \mathbf{extension}, parent) \implies (child : part, \mathbf{extends}, parent : part) \quad \text{(A.2)}$$

These two production rules make all information known about *videoEffect* also apply to *videoTransform*, resulting in the following expanded specification:

```
videoTransform{ type = opSpec;
extends = videoEffect ;
input = :in:input ;     output = :out:output ;
:in:* { dataType = video ; }
:out:* { dataType = video ; }
```

```
  :in:input { dataType = video ; }
  :out:output { dataType = video ; }
  inputCount = 1 ;     outputCount = 1;
  minInputCount = 1 ;     minOutputCount = 1;
}
```

In order to support restrictions and partial implementations it is necessary to support specifications that alter constraints that exist on the base specification. This is achieved by adding statements to the partial specification saying which statements need to be modified. During expansion the appropriate substitution will be made, so no false statements are copied into the partial specification. An example restriction is shown below:

```
rgbInTransform {
restricts = videoTransform ;
restriction = tmp1 {
rstSubj = rgbInTransform:in:* ;
rstPred = dataType ;
rstOldObj = video ;
rstNewObj = RGB24 ;
}
```

The changes are made by using the **restriction** relation to identify an object that links to the changes to be made, *:change1*. This object contains information about which statements should be altered during the copying process. The product rule used to control this expansion is:

$$
\begin{aligned}
&(partialSpec, \textbf{restricts}, baseSpec) \wedge (partialSpec, pred, obj) \implies \\
&\quad \text{if } \exists(partialSpec, \textbf{restriction}, (partialSpec, pred, obj, newObj)) \\
&\qquad (partialSpec, pred, newSubj) \\
&\quad \text{else} \\
&\qquad (partialSpec, pred, obj)
\end{aligned}
\tag{A.3}
$$

After applying this transform the *rgbInTransform* will contain the modified statements:

```
rgbInTransform {
  ...
  :in:* { dataType = RGB24 ; }
  :in:input { dataType = RGB24 ; }
    ...
}
```

A.4 Summary

The IGOL Operation Specification Schema provides a simple and convenient way to describe structure and functionality of components within the framework, and the ability to merge statements from different sources allows meta-data from different manifests to be merged without creating conflicts.