

Imperial College London  
Department of Computing

# Profile-directed specialisation of custom floating-point hardware

Ashley W. Brown

March 2010

Supervised by Paul H J Kelly and Wayne Luk

Submitted in part fulfilment of the requirements for the degree of  
Doctor of Philosophy in Computing of Imperial College London  
and the Diploma of Imperial College London

# Declaration

I herewith certify that all material in this dissertation which is not my own work has been properly acknowledged.

Ashley W. Brown

# Abstract

We present a methodology for generating floating-point arithmetic hardware designs which are, for suitable applications, much reduced in size, while still retaining performance and IEEE-754 compliance. Our system uses three key parts: a profiling tool, a set of customisable floating-point units and a selection of system integration methods.

We use a profiling tool for floating-point behaviour to identify arithmetic operations where fundamental elements of IEEE-754 floating-point may be compromised, without generating erroneous results in the common case. In the uncommon case, we use simple detection logic to determine when operands lie outside the range of capabilities of the optimised hardware. Out-of-range operations are handled by a separate, fully capable, floating-point implementation, either on-chip or by returning calculations to a host processor. We present methods of system integration to achieve this error-correction. Thus the system suffers no compromise in IEEE-754 compliance, even when the synthesised hardware would generate erroneous results.

In particular, we identify from input operands the shift amounts required for input operand alignment and post-operation normalisation. For operations where these are small, we synthesise hardware with reduced-size barrel-shifters. We also propose optimisations to take advantage of other profile-exposed behaviours, including removing the hardware required to swap operands in a floating-point adder or subtractor, and reducing the exponent range to fit observed values.

We present profiling results for a range of applications, including a selection of computational science programs, Spec FP 95 benchmarks and the FFMPEG media processing tool, indicating which would be amenable to our method. Selected applications which demonstrate potential for optimisation are then taken through to a hardware implementation. We show up to a 45% decrease in hardware size for a floating-point datapath, with a correctable error-rate of less than 3%, even with non-profiled datasets.

# Acknowledgements

A great number of people have given me support and guidance during the creation of this thesis, which was only possible due to funding from the Engineering and Physical Sciences Research Council. My thanks go to my supervisor, Paul Kelly, who helped drive forward my area of research with his enthusiasm for the subject. He helped me turn an “I wonder if...” proposition into a reality. My thanks also go to Wayne Luk, my second supervisor, who provided additional funding and inspired corrections when I was in need of both.

I am grateful to the many people who have contributed to my education, in particular my family, who gave me an excellent start in life. Finally, my thanks must go to Ivanka, who has supported me through four years of late nights, and kept me going when things got tough.

# Contents

|          |                                                   |           |
|----------|---------------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                               | <b>11</b> |
| 1.1      | Contributions . . . . .                           | 13        |
| 1.2      | Motivation . . . . .                              | 14        |
| 1.3      | Approach . . . . .                                | 15        |
| 1.4      | Definitions . . . . .                             | 17        |
| <b>2</b> | <b>Background</b>                                 | <b>18</b> |
| 2.1      | IEEE floating-point & Derivatives . . . . .       | 19        |
| 2.1.1    | Special Cases . . . . .                           | 20        |
| 2.1.2    | Basic Implementation . . . . .                    | 21        |
| 2.1.3    | Accuracy and Stability . . . . .                  | 22        |
| 2.1.4    | Custom Formats . . . . .                          | 26        |
| 2.1.5    | Related Representations . . . . .                 | 29        |
| 2.2      | Alternative Representations . . . . .             | 30        |
| 2.2.1    | Fixed-Point . . . . .                             | 30        |
| 2.2.2    | CORDIC . . . . .                                  | 32        |
| 2.2.3    | Logarithmic and Residue Representations . . . . . | 33        |
| 2.3      | Hardware Optimisation Approaches . . . . .        | 33        |
| 2.3.1    | Reconfigurable Systems . . . . .                  | 34        |
| 2.3.2    | GPUs and Cell . . . . .                           | 35        |
| 2.4      | Related Concepts . . . . .                        | 36        |
| 2.4.1    | Iterative Compilation . . . . .                   | 36        |
| 2.4.2    | Speculation . . . . .                             | 37        |
| 2.4.3    | Hardware/Software Partitioning . . . . .          | 38        |
| 2.5      | Summary . . . . .                                 | 39        |
| <b>3</b> | <b>Speculative Reduction of Floating-Point</b>    | <b>40</b> |
| 3.1      | Base Hardware Architecture . . . . .              | 42        |
| 3.2      | Operand Alignment Reduction (OAR) . . . . .       | 43        |

|          |                                                     |           |
|----------|-----------------------------------------------------|-----------|
| 3.3      | Normalisation Reduction . . . . .                   | 46        |
| 3.4      | Exponent Reduction . . . . .                        | 47        |
| 3.5      | Summary . . . . .                                   | 50        |
| <b>4</b> | <b>System Integration</b>                           | <b>51</b> |
| 4.1      | Hardware Options . . . . .                          | 52        |
| 4.1.1    | Optimised Floating-Point Arithmetic Unit . . . . .  | 52        |
| 4.1.2    | Optimised Data-path . . . . .                       | 53        |
| 4.2      | Error Recovery . . . . .                            | 55        |
| 4.2.1    | Iteration Indexed Re-execution Buffer . . . . .     | 55        |
| 4.2.2    | In-design Correction . . . . .                      | 56        |
| 4.3      | Summary . . . . .                                   | 59        |
| <b>5</b> | <b><i>FloatWatch: A Floating-Point Profiler</i></b> | <b>60</b> |
| 5.1      | The Tool . . . . .                                  | 62        |
| 5.2      | Alternative Uses . . . . .                          | 66        |
| 5.2.1    | Accuracy Loss . . . . .                             | 66        |
| 5.2.2    | Precision Loss from Cancellation . . . . .          | 66        |
| 5.2.3    | Denormal Numbers and Zero Values . . . . .          | 68        |
| 5.3      | Results . . . . .                                   | 70        |
| 5.3.1    | MORPHY . . . . .                                    | 70        |
| 5.3.2    | SpecFP95 Benchmarks . . . . .                       | 71        |
| 5.3.3    | FFMPEG and Runtime Reconfiguration . . . . .        | 73        |
| 5.4      | Summary . . . . .                                   | 76        |
| <b>6</b> | <b>Case Study: SPEC FP95 <i>102.swim</i></b>        | <b>77</b> |
| 6.1      | Analysis . . . . .                                  | 78        |
| 6.2      | System Architecture . . . . .                       | 88        |
| 6.3      | Results . . . . .                                   | 91        |
| 6.3.1    | Single Data-path . . . . .                          | 92        |
| 6.3.2    | Accelerator/Processor Modelling . . . . .           | 95        |
| 6.3.3    | Overlapped Execution . . . . .                      | 99        |
| 6.3.4    | Parallel Data-paths . . . . .                       | 102       |
| 6.3.5    | Non-profiled Data . . . . .                         | 102       |
| 6.4      | Summary . . . . .                                   | 105       |

|          |                                                    |            |
|----------|----------------------------------------------------|------------|
| <b>7</b> | <b>Conclusion</b>                                  | <b>106</b> |
| 7.1      | Review of Contributions . . . . .                  | 107        |
| 7.2      | Future Enhancements . . . . .                      | 109        |
| 7.2.1    | Additional Optimisations . . . . .                 | 109        |
| 7.2.2    | Power Optimisation . . . . .                       | 111        |
| 7.2.3    | Algorithmic Modifications . . . . .                | 112        |
| 7.2.4    | Dynamic Reconfiguration . . . . .                  | 112        |
| 7.2.5    | Modelling and Design Space Exploration . . . . .   | 114        |
| 7.2.6    | <i>FloatWatch</i> . . . . .                        | 115        |
| 7.3      | Limitations . . . . .                              | 116        |
| 7.3.1    | Application-Specific Integrated Circuits . . . . . | 116        |
| 7.3.2    | Quality of Datasets . . . . .                      | 117        |
| 7.4      | Concluding Remarks . . . . .                       | 117        |

# List of Tables

|     |                                                                                                                      |    |
|-----|----------------------------------------------------------------------------------------------------------------------|----|
| 2.1 | Bit-widths for IEEE-754 floating-point types. . . . .                                                                | 19 |
| 2.2 | Floating-point Optimisation Options . . . . .                                                                        | 28 |
| 3.1 | Hardware savings with Operand Alignment Reduction for floating-point adders and subtractors . . . . .                | 44 |
| 3.2 | Logic element use for a floating-point adder, with Operand Alignment Reduction and error detection options . . . . . | 46 |
| 3.3 | Hardware savings with Normalisation Reduction for floating-point adders and subtractors . . . . .                    | 46 |
| 3.4 | Hardware savings with Normalisation Reduction for floating-point multipliers . . . . .                               | 47 |
| 3.5 | Logic element use for a floating-point adder, with Normalisation Reduction and error detection options . . . . .     | 48 |
| 3.6 | Hardware savings with Exponent Reduction for floating-point adders and subtractors . . . . .                         | 48 |
| 3.7 | Hardware savings with Exponent Reduction for floating-point multipliers . . . . .                                    | 48 |
| 3.8 | Hardware use for exponent adjustment logic . . . . .                                                                 | 49 |
| 3.9 | Custom Bias Table . . . . .                                                                                          | 50 |
| 5.1 | Characteristics of three test videos . . . . .                                                                       | 74 |
| 6.1 | Execution Time Profile . . . . .                                                                                     | 78 |
| 6.2 | Expected Error Rates from Profiling Data . . . . .                                                                   | 87 |
| 6.3 | Base metrics for software-only execution . . . . .                                                                   | 92 |
| 6.4 | Observed re-execution (RX) rates for optimisation combinations . . . . .                                             | 94 |
| 6.5 | Cyclone II logic cell usage for base and optimised configurations                                                    | 94 |

# List of Figures

|     |                                                                                                      |    |
|-----|------------------------------------------------------------------------------------------------------|----|
| 1.1 | Simplified optimisation results . . . . .                                                            | 12 |
| 1.2 | Basic Conceptual Overview . . . . .                                                                  | 16 |
| 1.3 | The optimisation tool-chain . . . . .                                                                | 17 |
| 2.1 | Generalised IEEE-754 layout. . . . .                                                                 | 19 |
| 2.2 | A simple floating-point adder, without special cases . . . . .                                       | 23 |
| 2.3 | Distribution of hardware resources for optimised and unoptimised floating-point subtractors. . . . . | 24 |
| 2.4 | A simple floating-point multiplier, without special cases . . . . .                                  | 24 |
| 2.5 | Errant behaviour of $f(x) = \frac{1-\cos(x)}{x^2}$ . . . . .                                         | 27 |
| 2.6 | IEEE-754 with block floating-point and dual fixed-point layouts. . . . .                             | 29 |
| 3.1 | Optimisation opportunities for a floating-point adder . . . . .                                      | 41 |
| 3.2 | Staged-multiplexer design for variable shifter . . . . .                                             | 41 |
| 3.3 | Operand Alignment Example . . . . .                                                                  | 44 |
| 3.4 | Functional diagram of error detection hardware for Operand Alignment Reduction . . . . .             | 45 |
| 4.1 | Software and hardware stacks with error detection/correction . . . . .                               | 51 |
| 4.2 | Processor integration methods . . . . .                                                              | 52 |
| 4.3 | Flow Graph from the Swim Benchmark . . . . .                                                         | 54 |
| 4.4 | Functional Diagram of the Iteration Indexed Re-execution Buffer . . . . .                            | 56 |
| 4.5 | In-design correction . . . . .                                                                       | 58 |
| 5.1 | Valgrind instrumentation steps: decompilation, instrumentation and recompilation. . . . .            | 62 |
| 5.2 | Instrumentation of floating-point operands with the float_ratio function. . . . .                    | 63 |

|      |                                                                                                                                         |     |
|------|-----------------------------------------------------------------------------------------------------------------------------------------|-----|
| 5.3  | Exploring profile results using the <i>FloatWatch Explorer</i> user interface . . . . .                                                 | 65  |
| 5.4  | Profile of errant behaviour of $f(x) = \frac{1-\cos(x)}{x^2}$ . . . . .                                                                 | 67  |
| 5.5  | The Effect of Gradual Underflow . . . . .                                                                                               | 69  |
| 5.6  | Profile results for ‘MORPHY’ . . . . .                                                                                                  | 71  |
| 5.7  | The percentage of operations able to execute without errors with a given maximum shift for SpecFP95 benchmarks. . . .                   | 72  |
| 5.8  | Value-range profile results for <i>mgrid</i> . . . . .                                                                                  | 73  |
| 5.9  | The percentage of operations able to execute without errors with a given maximum shift for FFMPEG. . . . .                              | 75  |
| 6.1  | CALC1 Source – Main Loop . . . . .                                                                                                      | 79  |
| 6.2  | CALC2 Source – Main Loop . . . . .                                                                                                      | 79  |
| 6.3  | CALC3 Source – Main Loop . . . . .                                                                                                      | 80  |
| 6.4  | Operation-level profile of CALC3 inner loop . . . . .                                                                                   | 80  |
| 6.5  | <i>FloatWatch</i> profiling results for <i>102.swim</i> . . . . .                                                                       | 81  |
| 6.6  | Percentage of operations which can be executed successfully for a given maximum alignment shift ( $\Delta e$ ) in <i>102.swim</i> . . . | 83  |
| 6.7  | CALC3 Source – Re-written Main Loop . . . . .                                                                                           | 84  |
| 6.8  | CALC3 Common Data-flow Graph . . . . .                                                                                                  | 85  |
| 6.9  | Summarised CALC3 Profiling Results - Operation Level . . . .                                                                            | 86  |
| 6.10 | Register layout for the <i>102.swim</i> accelerator slave port . . . .                                                                  | 90  |
| 6.11 | Key for <i>102.swim</i> results . . . . .                                                                                               | 91  |
| 6.12 | Optimisation Results for CALC3 - Single Data-path . . . . .                                                                             | 93  |
| 6.13 | Execution Time Model . . . . .                                                                                                          | 98  |
| 6.14 | Optimisation Results for CALC3 - Single Data-path, Overlapped100                                                                        |     |
| 6.15 | Optimisation Results for CALC3 - Single Data-path, Overlapped with Processor Hardware Floating-Point Unit . . . . .                     | 101 |
| 6.16 | Optimisation Results for CALC3 - Parallel Data-paths . . . . .                                                                          | 103 |
| 6.17 | CALC3 error rate variation for non-profiled data . . . . .                                                                              | 104 |
| 7.1  | A ‘Handed’ Floating-Point Adder . . . . .                                                                                               | 110 |
| 7.2  | Graph of $x$ at each iteration of the Newton-Raphson method                                                                             | 114 |

# 1 Introduction

We present a new technique for the optimisation of floating-point data-paths, providing a 45% decrease in the size of a floating-point accelerator for our main test case, whilst retaining compliance with the IEEE-754[IEE85] double-precision standard. We show that this hardware saving can allow us to increase the number of processing paths placed onto a field-programmable gate array (FPGA).

One core principle underlies the optimisations that we make: optimised units may generate erroneous results, but they will be simply detected and corrected by re-execution on a fully featured unit, or with software emulation. As a consequence we over-optimize, trading off the re-execution rate against hardware size. This mechanism has a secondary benefit, in that datasets not matching the original profile will still run correctly, but with a higher re-execution rate (with a consequent performance penalty) than originally planned for.

We wish to set the scene for our work here, with a simplified preview of results to come. Figure 1.1 shows the effect of our optimisations on the size of a hardware accelerator for the SpecFP95 benchmark *102.swim*. Each result represents a different combination of optimisations in double-precision (denoted by “52”). The optimisations will be properly introduced in Chapter 3, however as a summary they are: Operand Alignment Reduction (OAR), Normalisation Reduction (NR) and Exponent Reduction (ER). Each option has a suffix, either a number (denoting the level of optimisation applied, 1 being the most optimised) or the letter P, which denotes a profile-driven optimisation. This graph sets the context of our work: we can see that our optimisations allow for a decrease in hardware size, but that the errors they introduce increase execution time.

Chapter 6 shows full results for the *102.swim* benchmark.

Our technique has three main components: a run-time profiler to determine floating-point behaviour, a set of optimisable floating-point hardware

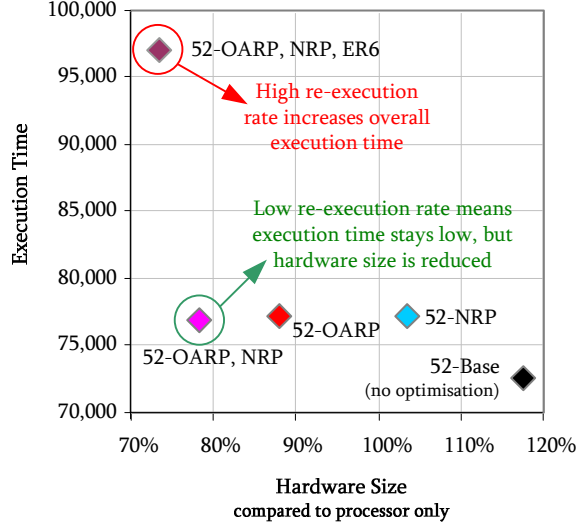


**Figure 1.1:** Simplified results for the *102.swim* benchmark, showing the effect of optimisations on error rate and hence execution time.

units and a variety of methods for integration with a host system for error correction and data communication.

We have constructed *FloatWatch*[BKL07], a floating-point profiler, to collect information on the floating-point behaviour of our target applications. We use it to evaluate a suite of selected applications, identifying opportunities for optimisation. This suite includes SpecFP95 benchmarks, computational science applications and the FFMPEG video processing libraries. One application is taken through the full optimisation process from profiling to custom hardware, using optimisations we have devised.

Chapter 2 provides the background information required to understand our method, covering other optimisation methods and the specifics of the IEEE floating-point standard.

We present the core of our technique in Chapter 3, detailing our hardware architecture and optimisations. In Chapter 4 we cover mechanisms for integrating hardware designs with a host processor, including methods for error recovery. Our *FloatWatch* profiler is presented in Chapter 5, along with profile results for a selection of applications. We also explore potential uses of the profiler outside of our work. Chapter 6 contains the full demonstration of our technique, using the *102.swim* benchmark from SpecFP95.

The benchmark is taken through the optimisation process, starting with a standard `gprof` profile and progressing to floating-point profiling and analysis. The outcome of this step is used to inform optimisation decisions, the benchmark finally being run in accelerated form on an FPGA.

Finally, Chapter 7 explores future directions for our work, and draws some conclusions from what has been achieved so far.

## 1.1 Contributions

Our work contributes new techniques to the optimisation of floating-point, targeted specifically for implementation on field-programmable gate arrays. In addition, we contribute profiling results for a number of applications, including those we do not progress to the optimisation stage.

In summary, our contributions include:

- A speculative optimisation technique offering a reduction in the hardware resources used by a hardware floating-point accelerator, showing a 45% saving in our benchmark, described in Chapter 3.
- A selection of methods for integrating our optimised floating-point hardware with a host processor, including error detection and correction mechanisms. These are described in Chapter 4.
- *FloatWatch*, a tool for profiling the characteristics of floating-point applications, described in Chapter 5.
- Floating-point profiles for a variety of floating-point applications and an interpretation of associated profile data, also in Chapter 5.
- A specific application of our technique to accelerators for the *102.swim* benchmark from SpecFP95, found in Chapter 6.

The *FloatWatch* profiler, while designed for our work here, has other applications in diagnosing errant floating-point behaviour. Section 5.2 describes these alternative uses.

Preliminary results from this research have been published in the following papers:

- Our paper “Profiling floating-point value ranges for reconfigurable implementation”, at the 2007 HiPEAC Workshop on Reconfigurable

Computing[BKL07], introduces and describes the initial version of the *FloatWatch* profiler, briefly touches on how the data is intended to be used and presents some basic profiling results for the SpecFP95 benchmarks. These results can also be found in Section 5.3.

- We described the optimisations in more detail in “Profile-directed speculative optimization of reconfigurable floating-point data paths”, at the 2008 HiPEAC Workshop on Reconfigurable Computing[BKL08], presenting further profiling results for a molecular mechanics application and an initial demonstration of the technique when applied to it. Once again, these results can be found in Section 5.3.

The reader is also invited to look at our previous work on high-level language optimisation for FPGA hardware designs, which details a C-based code transformation system for producing smaller, faster hardware. This can be found as a summary in “Generating Hardware Designs by Source Code Transformation”[BLK06], or in detail in “Optimising Transformations for Hardware Compilation”[Bro05].

## 1.2 Motivation

The IEEE-754-1985 standard[IEE85] (hereafter referred to as IEEE-754) has become almost ubiquitous in modern processing systems, including those used for scientific computation. However, for custom computing scenarios, such as field-programmable gate arrays (FPGAs)[CH02] and Application Specific Integrated Circuits (ASICs), its biggest benefit becomes its worst liability: the standard is a general-purpose solution. With this generality comes extra baggage, which may be surplus to requirements on an application-by-application basis. For this reason custom computing platforms often use their own floating-point formats, or more restrictive versions of the IEEE standard. We cover some of these options in Chapter 2.

Digital filters for video and audio processing are frequently implemented in FPGAs and ASICs, and typically use either a fixed-point[LM97] or a CORDIC[Vol59] solution in order to save on size and complexity. Graphics cards adopted a basic single-precision variety of IEEE-754, as minor errors in single frames of games were inconsequential at high frame rates. The consumer version of the Cell architecture uses fast, single-precision

floating-point “synergistic processing elements” [FAD<sup>+</sup>05] for the same reason. The more accurate double-precision format operates at a lower speed. In computational science, errors in calculations have more significance - rather than a single area of a single frame being incorrect, cumulative inaccuracies can build up and become unacceptable. Reproducibility is also a factor: ideally the same result would be achieved on multiple platforms, although this is rarely the case. Reasons why reproducibility may not occur are well-documented [Hig02], along with other pitfalls of floating-point calculations which were considered at the time the IEEE standard was being developed [Kah83]. Moving from an IEEE-754 compliant architecture to an application-specific one raises the same concerns, along with the risk of edge cases being handled incorrectly (or not at all). Demand from the scientific community led to the introduction of the IBM PowerXCell [HMZ<sup>+</sup>08], a Cell processor with faster double-precision performance, to be used in the Roadrunner supercomputer at Los Alamos National Laboratory [BDH<sup>+</sup>08]. The graphics market, too, is moving towards double-precision pipelines as GPUs increasingly see use as general-purpose application accelerators, as demonstrated by nVidia’s Tesla [LNOM08] and AMD FireStream 9170 [Adv09].

It is in this context of a drive towards improved double-precision IEEE-754 compliance that our optimisation method is presented.

### 1.3 Approach

Our approach reduces the size of key hardware elements based on profiling data captured by our *FloatWatch* profiler. These size reductions reduce the capabilities of the floating-point units undergoing optimisation, leading to a reduced range of operands producing correct results. However, unlike conventional optimisation techniques, we accept that over-reduction will cause errors to occur. The specific areas we optimise permit straightforward identification of these errors, allowing affected calculations to be re-executed with an alternative, slower method.

In the same way that branch prediction [Smi98] trades off faster super-scalar execution against a mis-prediction penalty, we can trade off parallel execution against the cost of re-executing floating-point operations. In our case a decrease in execution time comes from placing more parallel processing units on a chip, a result of hardware savings within individual units.

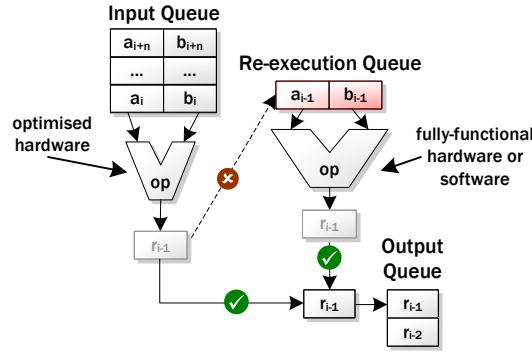


**Figure 1.2:** A simple overview of our technique – reducing hardware size enables increased parallel data-paths, but incorrect results must be re-calculated.

Alternatively, we trade pure hardware savings in a single optimised processing unit against the cost of re-execution.

Our penalty comes when these optimised units are unable to correctly calculate results for their input operands, and the result must be calculated again. This is a situation we limit by the use of *FloatWatch*, described in Chapter 5, to target optimisations to specific data-path elements. When errors do occur, the method of re-execution depends on the architecture chosen for the FPGA accelerator to be optimised. Possible options are described in Chapter 3.

Figure 1.2 shows the overall concept in diagrammatic form. We are able to almost double the number of floating-point units, but they do not all generate correct results for all operands. Results which are correct pass through unhindered, while incorrect results are re-calculated on a fully-functional floating-point implementation.

We follow a number of simple stages:

1. Profiling the target application (Chapter 5);
2. Selecting appropriate optimisations (Sections 3.2 to 3.4);
3. Selecting a system integration method (Section 4.1);
4. Making appropriate software changes.

An overall diagram of how the components interact is shown in Figure 1.3.

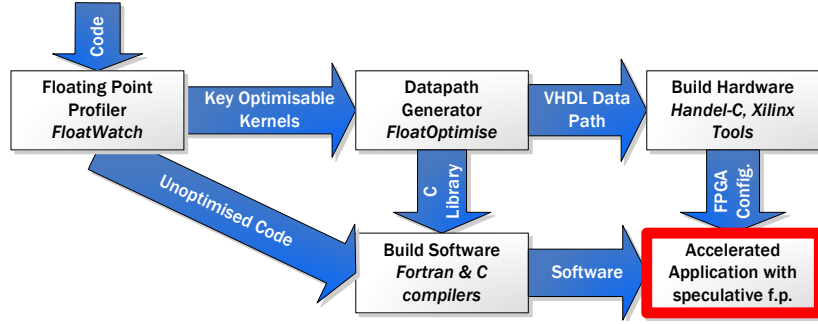


**Figure 1.3:** The optimisation tool-chain

## 1.4 Definitions

Throughout this thesis we make use of a number of standard definitions:

- $s$  – the significand of a floating-point number.
- $e$  – the exponent of a floating-point number.
- $\text{bits}(x)$  – the bit-width of  $x$  (e.g.  $\text{bits}(s)$  is the bit-width of the significand).
- $\rho$  – the error/re-execution rate encountered when using our optimisation technique.
- $n_b$  – a number,  $n$ , represented in base- $b$ . Numbers are shown in base-10 unless otherwise specified.
- $a : b$  – bit-concatenation of  $a$  and  $b$ , with  $a$  forming the most significant bits.

## 2 Background

Floating-point number representations have applications in a wide variety of computing areas, from digital signal processing in embedded systems to molecular mechanics modelling on super-computers.

The purpose of this chapter is to provide a context for our work, while also presenting the specifics required to understand it. We mainly describe variations of floating-point implementations in various devices, along with the systems which use them.

Section 2.1 focusses on the IEEE-754 standard and similar real-number representations, including its non-standard derivatives. Section 2.2 highlights other common representations in use on our target platform and comments on their suitability. Section 2.3 provides more detail about the optimisation techniques used on our target architecture and similar platforms.

Finally, Section 2.4 overviews some techniques we have drawn on for our work, including iterative compilation and speculative execution.

## 2.1 IEEE floating-point & Derivatives

Floating-point formats provide a compact representation of a broad range of numbers, akin to scientific notation. They do so at the expense of uniformly precise arithmetic. IEEE-754[IEE85, IEE08] is a well-established standard used in general purpose floating-point units, which strictly and extensively defines the behaviour of compliant systems including: formats for memory storage, rounding modes, special values such as ‘NaN’ (Not-a-Number) and  $\infty$ , and exception modes for possible error cases.

The current revision of IEEE-754 defines four types of floating-point representation, each with the same basic layout – a sign bit, significand and exponent – but different bit sizes for each. The formats are broken into two categories, ‘standard’ ones which are explicitly defined and ‘extended’ ones which are less strictly constrained. Floating-point libraries for FPGAs often allow the widths of both significand and exponent to be fully specified, producing IEEE-754-style formats not defined in the standard. The general layout for an operand complying with the standard can be seen in Figure 2.1, with Table 2.1 detailing bit-widths for the specific types it defines.



**Figure 2.1:** Generalised IEEE-754 layout.

Numbers are stored as three bit fields in common with scientific notation: sign,  $n$ , significand,  $s$  and exponent,  $e$ . The value of a number would therefore be  $n \times s \times 2^e$ , however the specifics of the implementation make this more complicated. At the hardware level, the sign is represented as a single bit, with  $1_2$  representing a negative number. The significand is normalised,

| Type                | Total Size (bits) | Significand ( $s$ ) | Exponent ( $e$ ) | Bias ( $b$ ) |
|---------------------|-------------------|---------------------|------------------|--------------|
| <b>Single</b>       | 32                | 23                  | 8                | 127          |
| <b>Extended Sgl</b> | $\geq 43$         | $\geq 31$           | $\geq 8$         | unspecified  |
| <b>Double</b>       | 64                | 52                  | 11               | 1023         |
| <b>Extended Dbl</b> | $\geq 79$         | $\geq 64$           | $\geq 11$        | unspecified  |

**Table 2.1:** Bit-widths for IEEE-754 floating-point types.

meaning the first digit is always  $1_2$ <sup>1</sup>: consequently this leading 1 is not stored, providing one additional digit of precision. Finally, the exponent is represented in binary using a biased representation, with bias,  $b$ , equal to  $(2^{\text{bits}(e)} - 1) - 1$ . The value of a number in the floating-point representation is therefore  $(-1)^n \times 1.s \times 2^{e-b}$ .

### 2.1.1 Special Cases

In addition to the normalised format, IEEE-754 has a number of special cases. Numbers where  $e = 0$  and  $e = 2^{\text{bits}(e)} - 1$  are both reserved, which explains why the bias on the single-precision floating-point format is 127 rather than 128.

The implicit leading one described previously means the value zero cannot be represented in the standard form, requiring it to be handled as one of the special cases. The value  $e = 0$  is reserved for two special uses, one of them being zero. When  $e = 0$  and  $s = 0$  the floating-point number is considered to be zero and **not**  $(-1)^n \times 1.0 \times 2^{-b}$  as would be expected.

The second special use for  $e = 0$  is denormalisation, where the significand is not normalised and the leading 1 is not implicit. Zero is in effect a specific case of denormalisation. Use of denormalisation provides an increase in the range which can be represented, filling in the gap between the smallest non-zero number and zero. In this case the number represented is  $(-1)^n \times 0.s \times 2^{1-b}$ .

A second set of special cases occurs when  $e = 2^{\text{bits}(e)} - 1$ , that is all the bits of the exponent are 1. These are reserved for infinite values and the NaN. An IEEE-754 number with all-zero significand and all-one exponent represents  $\pm\infty$ , with the sign being determined by the sign bit, as usual.

All other representations with  $e = 2^{\text{bits}(e)} - 1$  represent NaN values, which indicate an indeterminate result or error. There are two types: Quiet NaNs (QNaN's) and Signalling NaNs (SNaN's). QNaN's have a 1 in the most significant bit of the significand. They propagate through successive operations, where they can be detected by an application as a final result without needing to verify every operation. SNaN's have a 0 in the most significant bit of the significand. The presence of an SNaN as a result causes an exception to be raised for error handling.

---

<sup>1</sup>Except in special cases, defined shortly.

### 2.1.2 Basic Implementation

Hardware implementations of the IEEE-754 standard come in a variety of flavours, including open source and proprietary versions. FPGA vendors typically supply their own platform-specific versions for use as ‘black-box’ units. Hemmert presented a high-performance, open-source floating-point library [HU06] with better performance than the Xilinx proprietary cores available at the time.

The principal problem with floating-point is the size of the hardware involved, in particular large dynamic shifters, the reasons for which will become clear in the following sections.

#### Common Elements

All IEEE-754 compliant operations share common steps, required to convert from the standard memory format to an internal representation allowing calculation and back again. Denormalisation is a simple step performed on all operands, which adds a leading one to the significand if necessary - for denormal numbers the significand remains the same.

Normalisation and rounding is more complicated (and hence resource-intensive), and applied to convert an internal floating-point format into the packed IEEE-754 standard representation with implicit leading one. To achieve this the result of the fixed-point calculation in a floating-point operation must be normalised, or flagged as a denormal number requiring special handling. Four acceptable rounding modes are provided by the standard: round to nearest, round towards zero, round towards  $+\infty$ , and round towards  $-\infty$ , the first of these being the default.

Normalisation requires shifting the significand until the most significant bit becomes 1, adjusting the exponent as necessary. A special case applies where the value is too small to be represented as a normalised number, in which case the exponent becomes zero and the result is a denormal.

#### Floating-Point Adder and Subtractor

Figure 2.2 illustrates the core of a simple floating-point adder/subtractor with no support for special cases and with no normalisation hardware. Figure 2.3 shows the resource usage of the various components on an Altera Stratix II, for our floating-point components. Note that double-precision

floating-point requires an 11-bit subtractor, 53-bit adder and barrel shifter capable of shifting anything from 0 to 53 bits to the right. The latter component is the most costly of them all, providing scope for optimisations which will be discussed in Chapter 3.

A floating-point subtractor is the same, however with the sign of the second operand inverted.

### Floating-Point Multiplier and Divider

The structure of a floating-point multiplier is simple by comparison, requiring a standard integer multiplication of the significand and the sum of the operand exponents. Figure 2.4 shows the basic structure of a floating-point multiplier, without special cases.

A divider is similar, requiring an integer division and subtraction of the operand exponents. However, it is greatly complicated by the necessity to divide the significand, for which there are multiple approaches[Rob58, HP98] trading off latency with hardware area.

#### 2.1.3 Accuracy and Stability

The floating-point standard, being a finite representation of an infinite space, provides only an approximation to real numbers. A simple example of this is with the number  $1.1_{10}$ , which cannot be represented exactly under IEEE-754. As a consequence careful attention must be paid to the construction of algorithms to be run on floating-point machines: this applies equally to any attempts at further reducing or manipulating the representation. The issues to consider are:

- **Precision:** given known accuracy and error bounds, how precise are the results? If we require 5 digits of precision for an answer, are all 5 correct?
- **Stability:** Does the algorithm behave correctly when presented with small errors?
- **Reproducibility:** Does a program generate identical results on different systems?

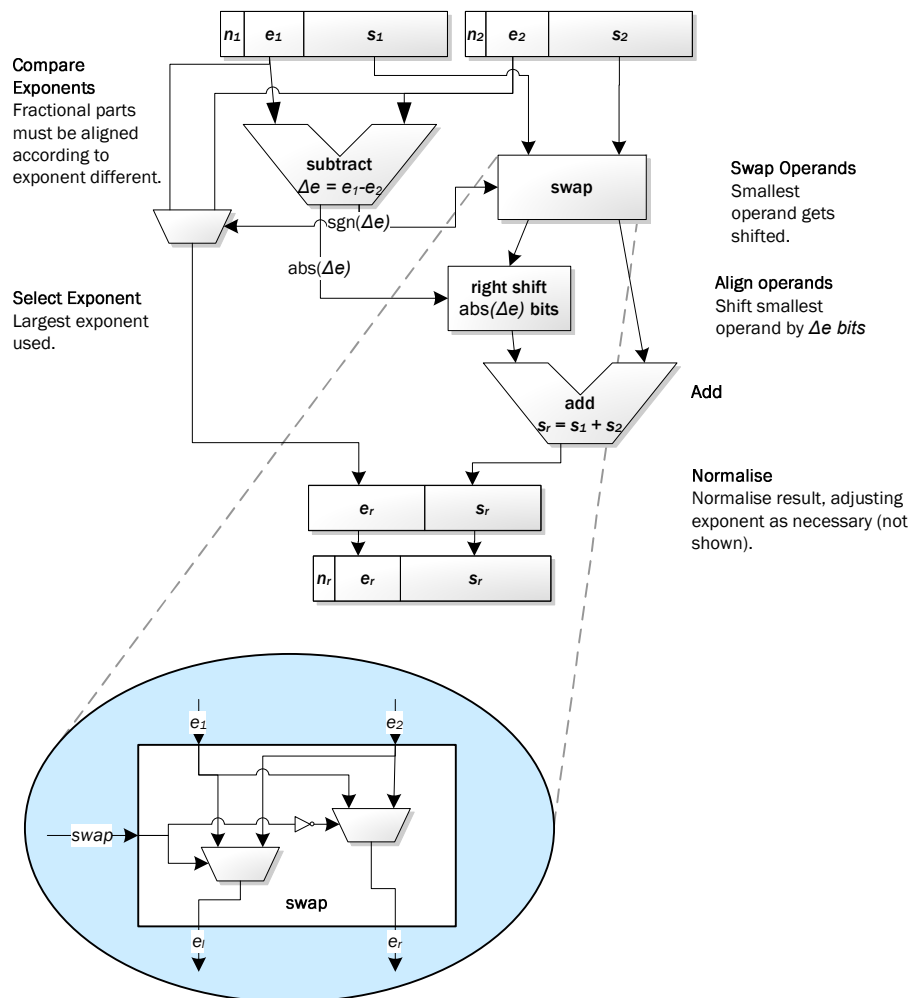


**Figure 2.2:** A simple floating-point adder, without special cases

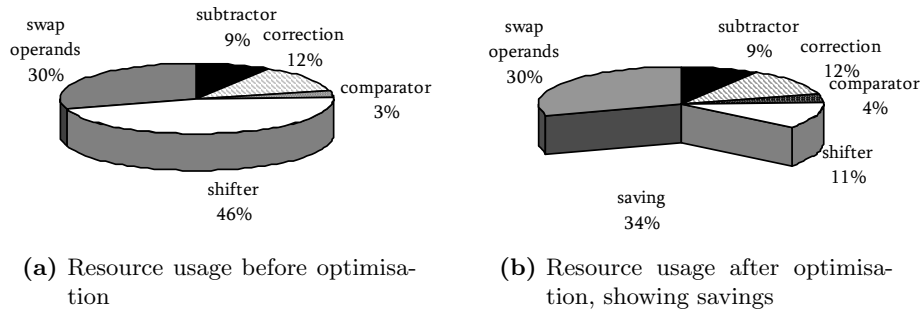


**Figure 2.3:** Charts showing resource distribution, given as Altera Stratix II ALUTs, between functional areas on a floating-point subtractor from the VFLOAT library (see Section 2.3) and the savings made by our optimisation. The components are as follows: **subtractor** - significand subtraction operation; **correction** - conversion of operand from internal format to floating-point; **comparator** - operand exponent comparison; **shifter** - the shift operation for operand alignment; **swap operands** - swap of operands so smaller is shifted.

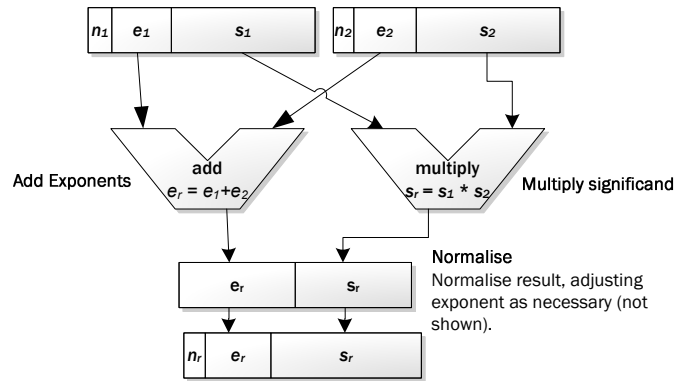


**Figure 2.4:** A simple floating-point multiplier, without special cases

Calculations in an infinite space would perform precisely as an algorithm would suggest, however discretisation introduces small errors leading to inaccuracies. We take the aforementioned  $1.1_{10}$  as an example: the fractional part,  $0.1_{10}$ , is not precisely representable as a binary number in finite precision, instead requiring an infinite recurrence. When represented as a 32-bit single-precision floating-point number, we have the following (recall that a single-precision floating-point number has a 23-bit significand, with implicit leading one):

$$1.00011001100110011001101_2 \times 2^0$$

Converting back to a real decimal number gives 1.100000023841858.

One of the most important concepts to consider when implementing floating-point code is the stability of the algorithm chosen. An unstable algorithm is susceptible to errors in intermediate values, such as those caused by rounding: a minor change can produce a dramatically different result, as was seen with Milne's (manual) algorithm for solving differential equations[Mil26]. This was later revised[MR59] as it came to be implemented on automated computing machines, with a numerically stable version being presented. Algorithms which iterate to convergence are susceptible to this, despite their typical nature as progressively refined approximations - an unstable algorithm could produce convergence to a completely different value in the case of a small change in an intermediate calculation. Such problems were considered by those working on the IEEE standard, including Kahan who has a range of material on the problems[Kah83]. More recently, Higham has produced a useful reference for those looking to avoid some of the stability problems floating-point calculations can expose[Hig02].

In many cases reproducibility is also a desirable factor, but one that can prove elusive when moving away from sequential execution; as a result of the precision loss present in any finite representation, IEEE-754 floating-point is not associative. This is a problem faced by software compilers when re-ordering program statements and also applies to our optimisation technique. Moving from sequential to parallel execution whilst reproducing results exactly is not always possible without associativity, as re-ordering operations produces different results. These problems can be overcome, but only at the expense of extra work. We do not attempt to achieve exact results when errors are down to lack of associativity, however the work undertaken in [KD07] could be applied to our technique.

The stability issues mentioned previously arise due to representing an infinitely large space of real numbers with a finite encoding. A finite significand causes two distinct problems during calculation, in addition to the inherent discretisation problems shown above.

The first of these problems is caused when the difference of two similar numbers is calculated, with that result being used to derive a further result of a much larger or smaller magnitude. This problem is neatly shown by the example in a forthcoming textbook by Sedgewick and Wayne[SW]. Consider the equation

$$f(x) = \frac{1 - \cos(x)}{x^2},$$

which is approximately 0.5 in the range  $[-4 \times 10^{-8}, 4 \times 10^{-8}]$ .

Using floating-point to calculate data-points within this range causes an incorrect set of results to be generated, as seen in Figure 2.5. The first two plots, Figures 2.5a and 2.5b, show a Mathematica[Wol91] plot of  $f(x) = \frac{1 - \cos(x)}{x^2}$  and its equivalent  $f(x) = \frac{1}{x^2} - \frac{\cos(x)}{x^2}$ , respectively. The only difference is the order in which operations are executed, and neither is correct. Figure 2.5c shows the function plotted on a different scale, with a stationary point clearly visible about the  $y$ -axis at 0.5, which the preceding figures should both illustrate.

In the erroneous plots, the points start out close to 0.5, but then drift until  $f(x) = 0$ . The cause of this is easily identified from run-time profiling data using our *FloatWatch* profiler, as we show in Section 5.2.

#### 2.1.4 Custom Formats

Although IEEE-754 defines two main floating-point precisions, variations are possible when operand ranges can be known in advance. We first turn our attention to direct derivatives of the floating-point standards, using custom settings.

In this example we make the assumption that all operands are in the range  $[100.0000, 999999.9999]$ , and apply customisations on that basis.

Table 2.2 shows six types of floating-point representation: two complying with the IEEE-754 standard and four custom alternatives. The last column indicates the real value of 999999.9999 when represented in the given format. It is assumed that the standard floating-point rules apply, so placing all ones or all zeroes in the exponent represents special numbers such as infinity, NaN

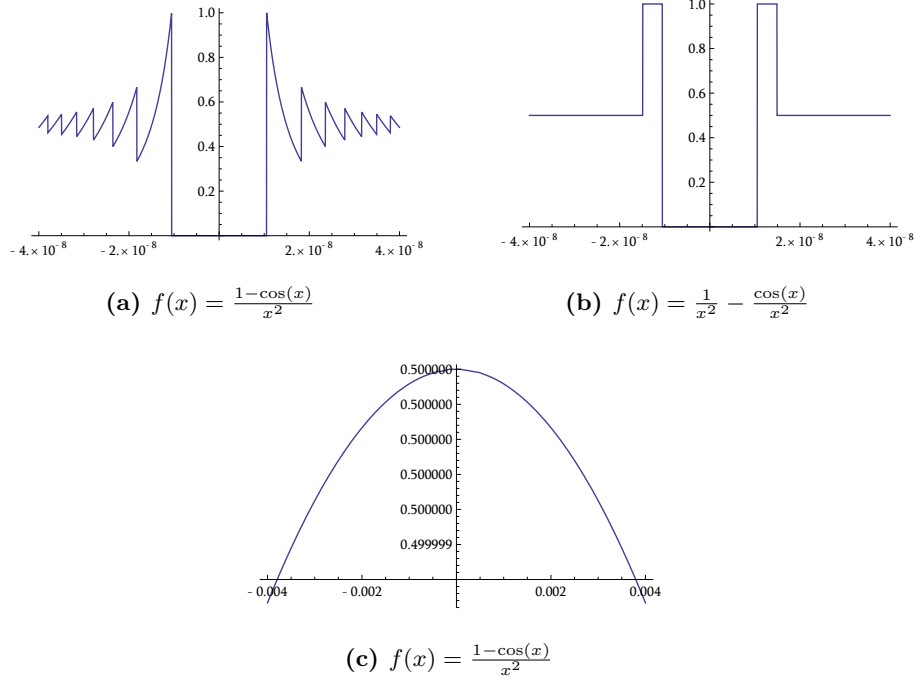


**Figure 2.5:** Errant behaviour of  $f(x) = \frac{1-\cos(x)}{x^2}$  and the equivalent  $f(x) = \frac{1}{x^2} - \frac{\cos(x)}{x^2}$ : in the range  $[-4 \times 10^{-8}, 4 \times 10^{-8}]$ , the expected value is approximately 0.5, as illustrated by the final figure.

| Type   | <i>T</i>  | <i>S</i>  | <i>E</i>  | <i>b</i>    | 999999.9999       | Smallest                | Largest               |
|--------|-----------|-----------|-----------|-------------|-------------------|-------------------------|-----------------------|
| Float  | <i>32</i> | <i>23</i> | <i>8</i>  | <i>127</i>  | 1000000.00        | $1.18 \times 10^{-38}$  | $3.4 \times 10^{38}$  |
| Double | <i>64</i> | <i>52</i> | <i>11</i> | <i>1023</i> | 999999.9999       | $2.22 \times 10^{-308}$ | $1.8 \times 10^{308}$ |
| Hybrid | 61        | 52        | 8         | 127         | 999999.9999       | $1.18 \times 10^{-38}$  | $3.4 \times 10^{38}$  |
| C57    | 57        | 50        | 6         | 31          | 999999.9999       | $9.31 \times 10^{-10}$  | 4294967296            |
| C46    | 56        | 50        | 5         | 0           | 999999.9999       | 2                       | 2147483648            |
| C55    | 55        | 50        | 4         | -5          | 999999.9999       | 64                      | 1048576               |
| C45    | 45        | 40        | 4         | -5          | 999999.9899999771 | 64                      | 1048576               |

**Table 2.2:** Floating-point Optimisation Options. Italicised items are IEEE-754 standard sizes.

and zero.

Three parameters are changeable:

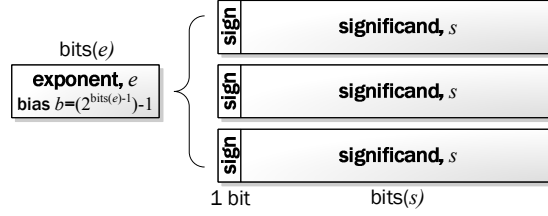
- bits(*s*), **bit size of the significand** - A larger significand provides greater accuracy, at the expense of a much larger barrel shifter.
- bits(*e*), **bit size of the the exponent** - A larger exponent provides a greater overall range of values, at the expense of larger hardware.
- *b*, **the bias** - under IEEE-754 the exponent uses a biased representation, with the bias being half of the available range (for an 8-bit exponent the bias is 127). Altering the bias allows the effective range to be moved, allowing a smaller exponent to be used.

The important thing to note from table 2.2 is that the first and last rows provide inaccurate results, as the significand has too few bits to represent the number precisely. The configuration C55 represents a 14% reduction in the number of bits for double-precision floating-point, but has no impact on the accuracy of the number being represented in this example. The representation Hybrid retains the same accuracy as double-precision floating-point, but with a reduced range.

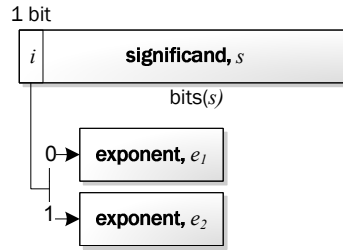
C45 and C55 utilise a custom bias to dramatically reduce the exponent, reducing the overall range available. As our numbers fall within this range, there is no problem with the calculation. It should be noted from Figure 2.2 that the bias is not explicitly represented in a floating-point adder - for normal operations it is not required, only becoming important when displaying numbers or converting them to other formats.



(a) IEEE-754-1985 binary layout



(b) Block floating-point layout



(c) Dual fixed-point layout

**Figure 2.6:** IEEE-754 with block floating-point and dual fixed-point layouts.

### 2.1.5 Related Representations

There exist other alternatives to the IEEE-754 standards which share some similarities. Here we consider block floating-point and dual fixed-point, which can be seen alongside IEEE-754 in Figure 2.6.

Block floating-point separates the exponent of a floating-point number from its sign and significand, with a block of numbers using the same exponent. Figure 2.6b illustrates this. The significands are denormalised, allowing a set of numbers to be considered together even when their exponents are not identical. Block floating-point is feasible providing all numbers in a block are able to be represented with the same exponent, which requires the exponents to be similar, or significand to have a sufficiently large bit-width. The common exponent may be adjusted during computation to reflect changes in the magnitude of the values. Should any number in the block become too large (or if the largest number in the block is small) the

shared exponent is adjusted to bring the number into range, while using the full extent of the significand available.

Dual fixed-point (DFX)[ECC04] is a hybrid of floating-point and fixed-point arithmetic. It occupies the middle ground between the flexibility of floating-point and the efficiency of fixed-point. By definition, fixed-point has the decimal point in a fixed place determined by the implementation. In DFX a single bit is reserved as a selector, allowing one of two positions for the decimal point to be selected. An alternative view of the format sees the selection bit as an exponent selector, with the ‘fixed-point’ part being a denormalised floating-point significand. The decimal point cannot ‘float’ to a range of locations, but can switch between two places. Figure 2.6c illustrates the format from this viewpoint: the selection bit acts as an index into a table of fixed exponents. The concept could thus be extended to more than one selection bit.

## 2.2 Alternative Representations

We now consider other representations for the set of real numbers,  $\mathbb{R}$ , which are used on our target architecture.

### 2.2.1 Fixed-Point

The most obvious and straightforward representation is fixed-point[VB04]. A real number is represented in the form  $m.n$ , where  $m$  is the integer part and  $n$  is the fractional part. Both  $\text{bits}(m)$  and  $\text{bits}(n)$  may be chosen to provide an appropriate range for the application being run.

A real number,  $r$ , may be converted to an integer fixed-point representation,  $f$ , via multiplication by a scaling factor and rounding:

$$f = \text{rnd}(r \times 2^{\text{bits}(n)})$$

The function `rnd` performs a standard rounding, turning the scaled real into a scaled integer. This scaled integer is stored as if it were a normal integer during calculations. It is divided by  $2^{\text{bits}(n)}$  to restore the real number, subject to any error introduced by the rounding.

For the purposes of additions and subtractions, fixed-point encoded reals may be treated as integers, as these operations have no effect on the scaling

factor. Multiplications and divisions may use standard integer operations, but require the correct scaling factor to be restored after calculation. The scaling factor is counted twice in multiplications or eliminated in divisions.

The major benefit of fixed-point is the use of standard integer operations, which are readily available on typical computer systems. Using the conventional binary representations of an integer, adjustments to the scaling factor required for multiplications and divisions are easily achieved using a shift left or right by  $\text{bits}(n)$  bits. This occurs as the scaling factor is always  $2^{\text{bits}(n)}$ .

The range of a fixed point implementation is determined by  $\text{bits}(m)$ , with a range of  $[-2^{\text{bits}(m)-1}, 2^{\text{bits}(m)-1} - 1]$  for signed numbers. The magnitude of  $\text{bits}(n)$  affects the precision of the implementation, determining how many significant digits of the fractional part are retained. Determining a specific fixed-point format is thus a trade-off between range and precision, given an upper limit on total word length (that is,  $\text{bits}(m) + \text{bits}(n)$ ).

Fixed-point operations are commonly used in custom hardware designs, as they consume far fewer resources than a floating-point system. For applications where small errors may be imperceptible, such as audio and video decoding, fixed-point allows for an efficient implementation in custom hardware or on an embedded processor. The MAD MP3 decoder[Und10] is an example of this, where MP3 decoding has been implemented entirely with fixed-point arithmetic.

On the design side, tools such as BitSize[GMLC04] are available to determine the appropriate fixed-point bit-width required to efficiently and correctly implement a design given known input and output constraints.

One of the principal drawbacks of fixed-point is the need to ‘waste’ bits in order to retain precision for a wide range of numbers. The bit-width must be set for the largest number which could be encountered, but at the bottom of the range this causes the most significant bits to be zero. Smaller numbers also have fewer significant bits due to this waste. Floating-point numbers provide a more compact and consistent format, with the same number of significant bits for any number, subject to errors introduced during calculation.

Our work has focussed on scientific applications, which use IEEE-754 standard floating-point. We chose to maintain compatibility with this standard, without compromising on precision. However, parts of our work in

this thesis could be transferred to fixed-point use, the profiler discussed in Chapter 5 being particularly useful for identifying value ranges and hence bit-widths. The general concept of flagging an error and re-executing could also be transferred, with re-execution taking place in the event of an overflow or underflow for an optimised fixed-point implementation.

### 2.2.2 CORDIC

CORDIC (COordinate Rotation DIgital Computer) is a method for implementing trigonometric and hyperbolic functions using simple additions and bit-shifts. It takes its name from the computer used in the United States' B-58 bomber[Vol00], for which the technique was originally designed. It was devised and implemented by Volder[Vol59] at a time when transistors were new and hence both slow and expensive, but has found uses even in today's systems. Walther drew together a number of algorithms based on the coordinate-rotation principle, to provide a unified algorithm[Wal71] for the calculation of elementary functions, including square-root, multiplication, division, and the logarithmic, trigonometric and hyperbolic classes. In 1998 Andraka[And98] conducted an extensive survey of CORDIC algorithms, and Kota had previously looked at the trade-off between hardware use and accuracy[KC93].

Mencer et al implemented CORDIC for FPGAs[MSMD00] using the PAMBlox[MMF98] framework. A floating-point unit for an embedded processor was developed by Zhou et al[ZDL<sup>+</sup>08, ZDL08], with double-precision floating-point operands and result, using co-ordinate rotation for the calculation step.

The principal benefit of CORDIC is that it has an efficient implementation which is low on hardware requirements, making it common in small microcontrollers and for FPGA implementations. Fixed-point elementary functions are calculated using a sequence of adds and shifts. Each iteration leads to a rotation clockwise or anti-clockwise, starting at  $45^\circ$  and reducing by half each time. The combination of these rotations allows any fixed-point rotation to be produced.

### 2.2.3 Logarithmic and Residue Representations

In addition to CORDIC, the field of digital signal processing has implemented other representations to reduce the complexity of specific operations.

In a logarithmic representation, a real number,  $R$ , is represented by its sign and the logarithm of its magnitude. Swartzlander provided a summary of the representation, which includes steps to avoid negative logarithms and other edge cases, along with a possible implementation[SCNS83]. The principal benefits are rapid multiplication and division, being possible with simple fixed-point additions and subtractions. However, additions and subtractions on the real numbers themselves are complex, so a logarithmic representation works best when these are not present.

Another alternative is to use a residue representation. A residue number system (RNS)[JL77] makes use of the Chinese Remainder Theorem and is defined by a set of moduli,  $\{m_1, m_2, \dots, m_i\}$ , which must be co-prime. A real number,  $R$ , is stored as a set of smaller numbers, each one of the form  $r_i = R \bmod m_i$ . Addition, subtraction and multiplication can all be achieved by adding, subtracting or multiplying corresponding residues, as so:  $c_i = a_i \text{ op } b_i \bmod m_i$ . Division requires more complexity, with Kaltofen and Hitz describing one possible method in [KH95]. The principal benefit of a residue representation is a reduction in the size of adders required, as each residue value is smaller than the original number. This reduces the size of carry-chains, which in turn increases the maximum possible clock frequency of an adder.

## 2.3 Hardware Optimisation Approaches

Although IEEE-754 strictly defines almost everything about its standard formats, for application-specific implementations this rigorous specification can add extra complexity and cost with little benefit. For this reason a number of specialised architectures make use of cut-down versions of the standard, for example with fewer rounding modes or ignoring special cases.

We begin with a discussion of our optimisations for our target architecture, the field-programmable gate array (FPGA), before continuing with graphics processing units (GPUs) and the Cell processor.

### 2.3.1 Reconfigurable Systems

Reconfigurable systems have become increasingly prevalent, with the field-programmable gate array (FPGA) increasing in size and utility every year. Todman produced a survey of the many different reconfigurable architectures available[TCW<sup>+</sup>05].

In the context of our work, FPGAs can use any conceivable floating-point implementation that can be made to fit on the chip. With the continuing increase in effective gate count, a few large, complicated floating-point units - or multiple smaller ones - may be placed on an FPGA. They also provide for variable-precision implementations, for example providing more bits for the fractional part of a 32-bit floating-point number than in the IEEE standard.

Double-precision IEEE floating-point was previously difficult to achieve on an FPGA due to space constraints. An increase the size of FPGAs and the addition of faster memory interfaces has improved the performance of libraries such as BLAS[UH04], making FPGAs a viable platform.

More recently, a 64-bit floating-point design was shown to achieve 15.6 GFLOPS on an FPGA[DVKG05], higher than a single Intel Itanium 2 core, one of the latest processors at the time. Consequently, assessments of using floating-point on FPGAs have become more optimistic as larger devices have become available[SWA95, LMM<sup>+</sup>98, UH04, Und04]. For several years FPGAs have included coarse-grained elements such as integer hardware multipliers[Xil07], in addition to fine-grained look-up tables. These embedded hardware blocks provide a smaller, faster implementation of commonly-used functions that would otherwise have to be implemented with look-up tables alone.

These positive developments have led to a shift, from FPGAs being solely the realm of embedded systems to numerous applications in computational science[LKM02, AAS<sup>+</sup>07, NH05, SGTP06, KP06].

Despite the increase in chip size, placing a fully-featured floating-point unit onto an FPGA still consumes large amounts of on-chip resources. Re-using the same unit repeatedly is possible, but introduces an artificial bottleneck: the performance advantage of FPGAs stems from their ability to perform many operations in parallel. The goal is thus to reduce the hardware size of a particular functional unit, increasing the number of units able to be placed in parallel. In order to ease this problem, Ho et al have

suggested a reconfigurable architecture using floating-point-specific coarse-grained units, with fine-grained elements for control logic[HYL<sup>+</sup>09].

BitValue[SG00] and BitWise[SBA00] use data-flow analyses to reduce computation bit-widths. The BitSize[GMLC04] tool is similar, but allows the user to specify target design constraints for size, clock frequency and latency. These are used to select optimum number representations from variable-precision fixed-point or floating-point options, with the aim of meeting those constraints as closely as possible. Cheung et al[CLM<sup>+</sup>05] developed a method for generating fixed-point versions of elementary functions, such as logarithm and square root, while using IEEE floating-point for input and output. The conversion between IEEE floating-point and fixed point is transparent to the user.

These methods all reduce the overall precision of the calculation, which may cause results to deviate, as has previously been shown. Gu et al[GVH05] implemented a molecular dynamics application on an FPGA and discussed this issue at length. They used 35-, 40- and 51-bit wide data-paths and checked for fluctuations in the total energy in the system being modelled. On the other hand, Zhuo et al[ZP05] implemented a sparse matrix solver in double-precision floating-point, with IEEE-754 compatible units. Monte Carlo simulation has also been successfully implemented with double-precision floating-point units[GFA<sup>+</sup>04].

### 2.3.2 GPUs and Cell

Graphics processing units and the Cell processor adopted single-precision varieties of IEEE-754 in their initial designs. Both served the same market: graphics rendering for games and video. Minor errors in single frames of games were inconsequential at high frame rates, while the reduced size of the single-precision unit allowed for both an increase in processing pipelines and decrease in memory use compared to a double-precision option. Standard GPU floating-point support is generally limited to 32-bit precision IEEE-compatible arithmetic, but without floating-point exceptions. Similarly, the consumer version of the STI Cell architecture uses fast single-precision floating-point “synergistic processing elements”[FAD<sup>+</sup>05, GHF<sup>+</sup>06] for the same reason, with the more accurate double-precision format operating at a much lower speed.

The large parallel processing ability of these graphics-oriented architectures made them appealing for use in computational science. Their lack of double-precision (or in the Cell’s case, fast double-precision) led to work on achieving double-precision results with single-precision units. Strzodka, as well as others such as Dongarra[Don06], have conducted work to use limited precision floating-point units on both GPUs[GST05] and FPGAs[SG06], relying on error-correction and iterative solution to produce accurate results in a mixed-precision environment.

As a result of such techniques, the consumer Cell processor showed promise for computational science applications[WSO<sup>+</sup>06, WSO<sup>+</sup>07, Fab07]. In addition, Stanford’s Folding@Home[LSS<sup>+</sup>09] supports the PlayStation3, allowing scientific calculations to run when a networked PS3 is not in use.

As described in Section 1.2, both of the graphics-oriented architectures have now introduced double-precision floating-point alternatives[HMZ<sup>+</sup>08, LNOM08].

## 2.4 Related Concepts

Our work draws on some familiar concepts from computer systems research, applying them to a specialised situation in the form of floating-point hardware optimisation. In this section we provide a brief overview of iterative and profile-driven compilation, and the use of speculative approaches in computer systems.

### 2.4.1 Iterative Compilation

Traditional compilers contain a series of static analysis and optimisation phases, which operate on intermediate representations of programs to determine their behaviour[ALSU06]. Static analysis phases within compilers have one fundamental, inescapable limitation - they can rely only on the structure of the program itself, with no knowledge of its run-time behaviour. Profile-driven compilation is a common technique, allowing the compiler to emit more efficient code for frequently taken branches, at the expense of less common ones.

There is an ever-growing body of research related to iterative and feedback driven compilation[FOK02, TVVA03]. This extends the profile-driven idea

to take bolder moves based on profile data, and drawing on machine learning techniques[LO04, ABC<sup>+</sup>06].

We seek to use similar techniques here, where we make bold decisions on floating-point calculations, but can fall back to slower methods where necessary.

### 2.4.2 Speculation

In their paper on Thread-Level Speculation (TLS), Steffan et al[SM98] described their technique as “allowing the compiler to view parallelization solely as a cost/benefit tradeoff, rather than something which is likely to violate program correctness”.

This neatly sums up the speculative approaches which are becoming common in computer architecture. Branch prediction[Smi98] is one of the earliest and most obvious examples of speculation, where a super-scalar processor executes instructions past a branch, without knowing which way that branch will go. A correct prediction yields faster performance, while an incorrect one causes a mis-prediction penalty as the incorrect instructions are scrapped and the machine executes the correct ones. In either case, the correct result is still achieved.

TLS[SCZM00, PO03] uses the assumption that threads are independent, freeing the programmer from traditional parallel-programming concerns such as locks. Each thread retains a speculative list of memory updates[GPL<sup>+</sup>03], which can be committed to memory at a later stage. If the process of committing speculative updates to physical memory reveals a conflict between threads, the affected threads can be reset to try again. The technique increases parallel execution when the assumption holds, but introduces a performance penalty when a conflict occurs.

Speculative code transformations[DCJNMW00, LCH<sup>+</sup>03] allow the compiler to overcome barriers to optimisation introduced by problems such as pointer-aliasing[ZRL96, HBCC99, LR04]. For example, pointers with a low probability of aliasing can be treated as genuinely separate items, which may allow the execution order to be modified to make most use of the underlying hardware. Recovery code confirms whether the pointers are separate and takes corrective action if not.

Our solution follows this same pattern: we assume that floating-point

operations can execute successfully on our optimised floating-point units. When this assumption does not hold, operations are re-executed with an associated performance penalty.

### 2.4.3 Hardware/Software Partitioning

An application to be accelerated using custom hardware must be split between software and hardware parts. Key kernels which benefit from hardware execution can be moved into the accelerator, while control logic remains in software on a host processor. Hardware/software co-design, or partitioning, has spawned a large selection of tools and techniques over the past twenty years.

Nimble allowed systems specified in C to be compiled for a combined embedded processor and FPGA, automatically partitioning applications at loop and basic-block level[LCD<sup>+</sup>00]. Streams-C[FGL01] and Marge[GKMK96] also used C-based descriptions, but targetted multi-FPGA systems. Streams-C requires the programmer to annotate C code in order to explore different hardware configurations, in contrast to Nimble which does this automatically. Luk, Wu and Page[LWP94] took a functional programming approach, implementing a partitioning system using the Haskell-based[HF92] Gofer environment. DEFACTO[SHD02] performs a limited design-space exploration to quantitatively assess possible loop nest implementations, selecting a near-optimal solution. COSYN[DLJ97] also provided a heuristic algorithm for partitioning, supporting multiple types of processing elements and communication architectures. Gupta et al demonstrated a behavioural system[GDM93] based on the HardwareC[KDM92] language. Kalavade focussed on an ASIC and embedded digital signal processor[KL94] combination. POLIS[BCG<sup>+</sup>97] used a similar target architecture, considering FPGAs for prototyping, but took input as a combination of graphics and the ESTEREL[BdS91] language to specify a finite state machine[HK95].

## 2.5 Summary

In this chapter we have provided a reference for the IEEE-754 floating-point standard, highlighting key features of the representation and implementation which are important for our work. We also described some related representations, block floating-point and dual fixed-point, placing the IEEE standard into context and showing the flexibility of floating-point systems. We included details of other number representations which are used on our target hardware and related platforms.

In the latter part of the chapter we provided an overview of optimisation approaches for our target platform. Finally, we described the concepts on which our work is based, drawing from the fields of iterative compilation and speculative hardware.

### 3 Speculative Reduction of Floating-Point

In this chapter we present the specific hardware optimisations we look to apply based on profiling data. We consider only the individual floating-point units in our system and how they may be optimised and flag erroneous results. The mechanisms for integrating them into a functioning accelerator, with error correction, are covered in Chapter 4.

Figure 3.1 shows the components of a floating-point adder, with possible areas for optimisation numbered from one to five. These areas are:

1. Reducing the bit-width of the exponent (Exponent Reduction, ER)
2. Reducing the size of the significand (Significand Reduction, SR)
3. Removing the hardware required to swap operands (Swap Elimination, SE)
4. Reducing the capabilities of the operand alignment shifter (Operand Alignment Reduction, OAR)
5. Reducing the capabilities of the normalisation shifter (Normalisation Reduction, NR)

Three of these optimisations have been the focus of our efforts so far: Exponent Reduction, Operand Alignment Reduction and Normalisation Reduction. Significand Reduction requires a compromise of the type we have sought to avoid: reducing the precision of our operands, with associated risks to accuracy and correctness. Swap Elimination will be considered in Section 7.2.

The optimisations OAR and NR rely on reducing the size of barrel shifters required to align operands and normalise results. Beauchamp[BHUH08,

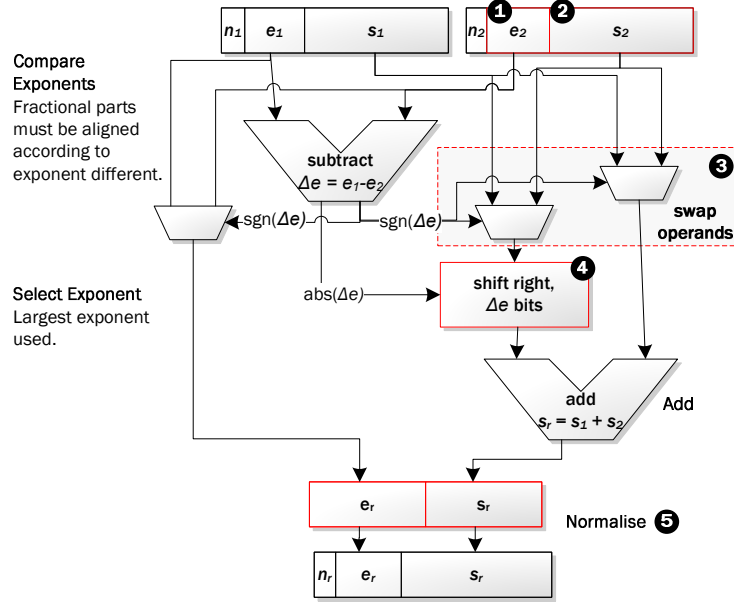


**Figure 3.1:** Optimisation opportunities for a floating-point adder

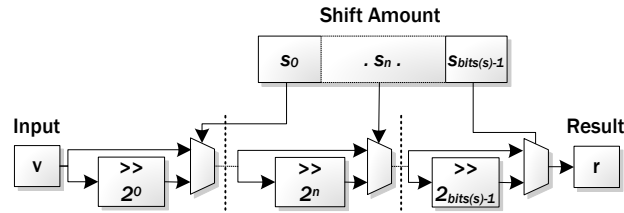


**Figure 3.2:** Staged-multiplexer design for variable shifter

BHUH06] presented alternative methods to achieve this goal, experimenting with the embedding of floating-point components into the FPGA fabric.

In our baseline architecture, both operand alignment and normalisation use a generic barrel shifter following a staged-multiplexer design, shown in Figure 3.2. Each stage shifts by a fixed power of two, with the combined stages providing a full range of possible shifts.

As the maximum required shift is reduced, stages may be removed from the shifter. We will show that it is possible to remove many of these stages in certain situations, by limiting shifts to just one or two bit positions.

Normalisation Reduction and Exponent Reduction may also be applied to floating-point multipliers, increasing the benefits they may bring.

Our optimisations allow for correct execution of operations in the common

case identified by profiling, but errors may occur for operands outside the expected ranges. Hence each form of optimisation requires a small amount of custom error-detection logic which will also be described.

The remainder of this chapter describes the optimisation options in more detail. We supply indicative values for relative hardware size using each optimisation, based on Logic Elements (LEs) for the Altera Cyclone II[Alt08a] (2C35) device. The units were placed and routed using Altera’s Quartus II software[Alt08b]. Tables of such values are presented for a single floating-point unit of a specified type, placed and routed independently of other hardware, using both single- and double-precision. Our optimisations are specified as **ABBR**- $n$ , where **ABBR** is an abbreviation as listed above, and  $n$  represents the optimisation level. For example, **NR**-2 represents Normalisation Reduction, where two stages are retained in the barrel shifter.

### 3.1 Base Hardware Architecture

We use the variable-precision VFLOAT library[BL02] as our baseline floating-point architecture, as it provides a set of cross-vendor, customisable VHDL components. FPGA vendors such as Xilinx[Xil06] and Altera[Alt09b] provide highly-optimised floating-point libraries which could be used for variable-precision floating-point operations, however they do not currently allow internal modifications of the type we are proposing here. Langhammer has produced a data-path synthesis system[Lan08] which provides some scope for customisation, however it is specific to Altera platforms. Performance in terms of area and maximum clock rate falls short of the vendor-specific libraries, but provides a suitable test-bed for evaluating the effect of optimisation on calculation error rates.

The library includes extended floating-point operations[WBL06], however we consider only addition, subtraction and multiplication at this stage. While sufficient for the scientific applications we have examined so far, the extended operations would be required for a number of important algorithms where our technique could be applied, such as the Black-Scholes[BS73] algorithm in computational finance.

A set of basic floating-point components can be assembled into full adders, subtractors and multipliers. We make our changes to these individual components, using pre-existing exception handling hardware to flag errors.

## 3.2 Operand Alignment Reduction (OAR)

In this section we explore reducing the size of the hardware required to align two floating-point operands for an add or subtract operation, as described in Section 2.1. Recall that one of the operands must be shifted by the difference of the exponents,  $\Delta e$ . Figure 3.3 provides an illustration of the process. We offer this optimisation, which we call *Operand Alignment Reduction*, as a method of shrinking the floating-point hardware without affecting the overall precision of the calculation.

We seek to reduce the maximum possible shift, reducing the hardware size but imposing limitations on the input operands: operations which require larger shifts for operand alignment cannot be successfully executed with our changes in place. Errors are easily identified during execution, as the required shift must still be calculated. If the shift is within acceptable range, the calculation proceeds as normal, otherwise an error may be flagged and the operation re-executed on a fully capable floating-point unit.

Although our base architecture uses a common method of implementing a barrel shifter, with a cascaded set of wide multiplexers, FPGA macro logic blocks provide an alternative approach. The multiplexer design can lead to high resource usage and congestion on the FPGA routing network as the size increases. However, each new generation of FPGAs brings a new generation of high-performance macro logic blocks, mostly aimed at digital signal processing. These are often configured to provide fast and efficient multiply or multiply-accumulate operations, but can also act as small barrel-shifters with the appropriate inputs. Larger barrel-shifters, such as those required for double-precision floating-point operations, also begin to consume routing resources to tie a number of macro-blocks together, as well as using valuable multiplier blocks. Consequently, we believe this optimisation to be useful in both multiplexer and macro-block designs.

We refer to this optimisation as *OAR- $n$* , where  $n$  is the number of shifter levels used in our design. *OAR-6* provides full capabilities for double-precision floating-point, while *OAR-5* does the same for single-precision<sup>1</sup>.

Error detection for Operand Alignment Reduction can be implemented with a basic check: when calculating the shift required to align operands

---

<sup>1</sup>*OAR-6* has a maximum shift of  $2^6 = 64$ , while a double-precision significand can be shifted by 53 before all significant bits are lost.

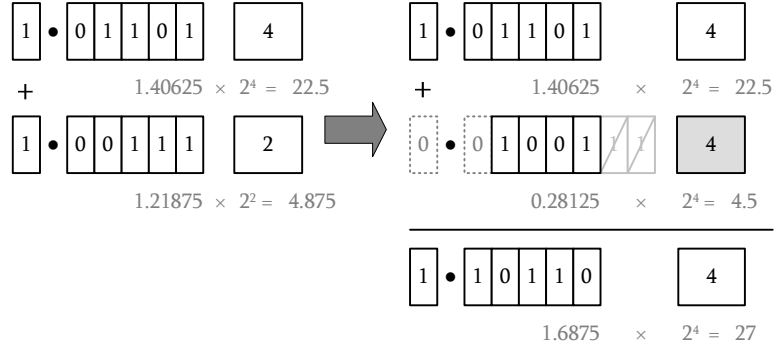


**Figure 3.3:** Illustration of operand alignment in the floating-point standard. The operand with the smaller exponent is shifted to equalise exponents, allowing a standard integer addition to take place. It is also important to note that precision is lost as numbers “drop off” the end of the operand being shifted. With a shift greater than the significand size, all significant bits can be lost in this way.

| OAR- $n$ | Single-Precision |        | Double-Precision |        |
|----------|------------------|--------|------------------|--------|
|          | Logic Elements   | % Size | Logic Cells      | % Size |
| 6        | -                | -      | 1,292            | 100%   |
| 5        | 591              | 100%   | 1,251            | 97%    |
| 4        | 571              | 97%    | 1,188            | 92%    |
| 3        | 539              | 91%    | 1,128            | 87%    |
| 2        | 515              | 87%    | 1,074            | 83%    |
| 1        | 512              | 87%    | 1,071            | 83%    |

**Table 3.1:** Hardware savings with Operand Alignment Reduction for floating-point adders and subtractors

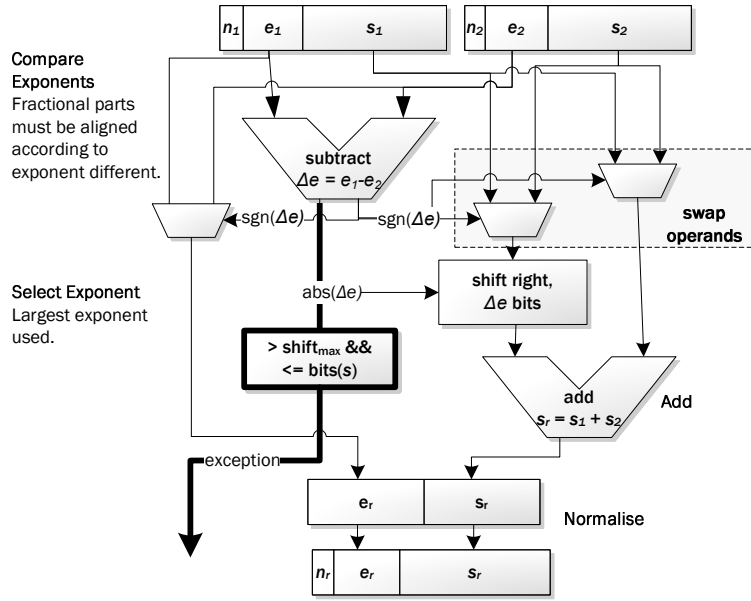


**Figure 3.4:** Functional diagram of error detection hardware for Operand Alignment Reduction

( $\Delta e$ ) we check for values which are out of range of our shifting hardware and raise an exception. Figure 3.4 shows how this error detection hardware integrates with the reduced floating-point unit.

In some circumstances an extended check can be made to reduce the error rate at the expense of a larger hardware design. This extended check determines if  $\Delta e$  exceeds the bit-width of the significand,  $\text{bits}(s)$ , supressing an error and returning zero for the fractional part. Table 3.2 shows the overhead required for error detection in Operand Alignment Reduction, for single- and double-precision implementations. As can be seen, basic error detection logic accounts for just two logic elements in both sizes (0.4% and 0.10% respectively). Additional logic to *avoid* triggering an error when the required shift is greater than the significand size occupies a further 1.6-2.1%.

Section 6.1 shows results where this extended check could produce great improvements to the error rate.

| Configuration            | Single<br>LEs | Double<br>LEs |
|--------------------------|---------------|---------------|
| Full Capability          | 591           | 1,292         |
| OAR-1 - No Detection     | 499           | 1,052         |
| + Out-of-Range Detection | 501           | 1,054         |
| + Significand Size Check | 512           | 1,071         |

**Table 3.2:** Logic element use for a floating-point adder, with Operand Alignment Reduction and error detection options

### 3.3 Normalisation Reduction

Our second proposed optimisation is *Normalisation Reduction* (NR). As with Operand Alignment Reduction, this reduces the capabilities of a shifter, this time in the normalisation logic. Unlike OAR, however, it applies to *all* floating-point operations, rather than just additions and subtractions. The normalisation step is required to pack a floating-point result into the standard binary format.

This optimisation provides the greatest gains, due to its wide applicability. Table 3.3 shows the hardware savings made when NR is applied to floating-point adders and subtractors, with Table 3.4 showing the same for multipliers. Note that the standard number of shift levels in a multiplier is one higher than for adders and subtractors, because a multiplier generates a result significand which is twice as wide as the input significands.

Detecting out-of-range results for our reduced normalisation components is as simple as for Operand Alignment Reduction, as they are both based on the same underlying component. In this case a priority encoder deter-

| NR- $n$ | Single-Precision |        | Double-Precision |        |
|---------|------------------|--------|------------------|--------|
|         | Logic Cells      | % Size | Logic Cells      | % Size |
| 6       | -                | -      | 1,292            | 100%   |
| 5       | 591              | 100%   | 1,250            | 97%    |
| 4       | 572              | 97%    | 1,195            | 92%    |
| 3       | 551              | 93%    | 1,146            | 89%    |
| 2       | 516              | 87%    | 1,091            | 84%    |
| 1       | 513              | 87%    | 1,091            | 84%    |

**Table 3.3:** Hardware savings with Normalisation Reduction for floating-point adders and subtractors

mines the magnitude of the shift required, which can be checked against the maximum possible. Unlike OAR, this check must take place after the result has been calculated, so there is no scope for short-cutting the calculation itself. Table 3.5 shows the impact of error detection logic on hardware size for a floating-point adder.

### 3.4 Exponent Reduction

Exponent Reduction is the simplest of the optimisations we consider here, and one for which many techniques exist. In this paper we restrict ourselves to a simple approach. Our profiler reveals the distribution of floating-point values within an application, which is used to restrict  $\text{bits}(e)$  to the smallest value capable of representing the overall range required. In keeping with our ‘aggressive optimisation’ approach, we continue to reduce  $\text{bits}(e)$  even if it would produce a small error. The use of a biased representation in the IEEE-754 standard presents a further option: modifying the bias in conjunction with  $\text{bits}(e)$ . This would eliminate a wasted bit for value distributions which have predominantly positive exponents.

Tables 3.6 and 3.7 show the hardware savings to be made with Exponent Reduction for adders/subtractors and multipliers, respectively. With adders and subtractors the hardware savings are initially much smaller than the other two optimisation types, however large gains can be made where the exponent can be reduced to four bits or less. For multipliers this optimisation has little effect, achieving no more than a 7% reduction in the most aggressive case.

| NR- $n$ | Single-Precision |        | Double-Precision |        |
|---------|------------------|--------|------------------|--------|
|         | Logic Cells      | % Size | Logic Cells      | % Size |
| 7       | -                | -      | 1,461            | 100%   |
| 6       | 566              | 100%   | 1,394            | 95%    |
| 5       | 566              | 100%   | 1,176            | 80%    |
| 4       | 481              | 85%    | 1,053            | 72%    |
| 3       | 425              | 75%    | 977              | 67%    |
| 2       | 394              | 70%    | 918              | 63%    |
| 1       | 396              | 70%    | 918              | 63%    |

**Table 3.4:** Hardware savings with Normalisation Reduction for floating-point multipliers

| Configuration          | Single<br>Logic Cells | Double<br>Logic Cells |
|------------------------|-----------------------|-----------------------|
| Full Capability        | 591                   | 1,292                 |
| NR-1 - No Detection    | 512                   | 1,071                 |
| Out-of-Range Detection | 513                   | 1,091                 |

**Table 3.5:** Logic element use for a floating-point adder, with Normalisation Reduction and error detection options

| ER- $n$ | Single-Precision |        | Double-Precision |        |
|---------|------------------|--------|------------------|--------|
|         | Logic Cells      | % Size | Logic Cells      | % Size |
| 11      | -                | -      | 1,292            | 100%   |
| 10      | -                | -      | 1,280            | 99%    |
| 9       | -                | -      | 1,267            | 98%    |
| 8       | 591              | 100%   | 1,255            | 97%    |
| 7       | 577              | 98%    | 1,242            | 96%    |
| 6       | 568              | 96%    | 1,217            | 94%    |
| 5       | 544              | 92%    | 1,159            | 90%    |
| 4       | 509              | 86%    | 1,085            | 84%    |
| 3       | 469              | 80%    | 1,017            | 79%    |
| 2       | 429              | 76%    | 951              | 74%    |
| 1       | 417              | 71%    | 942              | 73%    |

**Table 3.6:** Hardware savings with Exponent Reduction for floating-point adders and subtractors

| ER- $n$ | Single-Precision |        | Double-Precision |        |
|---------|------------------|--------|------------------|--------|
|         | Logic Cells      | % Size | Logic Cells      | % Size |
| 11      | -                | -      | 1,461            | 100%   |
| 10      | -                | -      | 1,455            | 100%   |
| 9       | -                | -      | 1,449            | 99%    |
| 8       | 566              | 100%   | 1,443            | 99%    |
| 7       | 559              | 99%    | 1,435            | 98%    |
| 6       | 554              | 98%    | 1,429            | 98%    |
| 5       | 548              | 97%    | 1,412            | 97%    |
| 4       | 542              | 96%    | 1,410            | 97%    |
| 3       | 532              | 94%    | 1,403            | 96%    |
| 2       | 531              | 94%    | 1,398            | 96%    |
| 1       | 524              | 93%    | 1,392            | 95%    |

**Table 3.7:** Hardware savings with Exponent Reduction for floating-point multipliers

| bits( $e_1$ ) | bits( $e_2$ ) | Logic Cells                    |                                |
|---------------|---------------|--------------------------------|--------------------------------|
|               |               | bits( $e_1$ ) to bits( $e_2$ ) | bits( $e_2$ ) to bits( $e_1$ ) |
| 11            | 10            | 12                             | 13                             |
| 11            | 9             | 11                             | 12                             |
| 11            | 8             | 11                             | 11                             |
| 11            | 7             | 10                             | 10                             |
| 11            | 6             | 9                              | 9                              |
| 11            | 5             | 9                              | 8                              |
| 11            | 4             | 8                              | 7                              |
| 11            | 3             | 7                              | 6                              |
| 11            | 2             | 7                              | 5                              |
| 11            | 1             | 6                              | 4                              |

**Table 3.8:** Hardware use for exponent adjustment logic

The method of error detection is a little different for Exponent Reduction than the other options, as the major part does not occur within the functional unit itself. Two sets of error detection logic are actually required: one set between data-path elements of differing exponent sizes (an error indicates a conversion is not possible), and another set when a calculated result overflows the exponent. The exponent overflow check is already performed, leaving just the conversion error to consider.

Increasing or decreasing the exponent size requires an adjustment to the bias, in addition to dropping or introducing extra bits. There is a consequent hardware consumption associated with this step, however since it need only occur where the exponent size changes, the cost can be spread over several units. Table 3.8 shows the hardware cost of conversion to and from an exponent of eleven bits, the double-precision standard. As this table shows, a greater size reduction produces smaller adjustment hardware, due to fewer bits being required when adjusting the exponent. It is also clear that this optimisation becomes unviable for small reductions, if used on a single functional unit; the hardware cost of converting to a smaller exponent, then back again, removes most of the advantage.

Although we do not consider it further in this work, by moving away from the IEEE-754 standard we have an additional way to reduce the exponent to make this method more effective. As described in Section 2.1, the floating-point standard uses a biased representation for the exponent, with the bias defined as  $2^{\text{bits}(e)-1}$ . This need not be the case providing suitable

| $e_{min}$ | $e_{max}$ | $e_{range}$ | $b$  | $\text{bits}(e_{range})$ | $\text{bits}(e)$ |
|-----------|-----------|-------------|------|--------------------------|------------------|
| -1023     | 1023      | 2046        | 1024 | 11                       | 11               |
| -10       | -1        | 9           | 11   | 4                        | 5                |
| 0         | 100       | 100         | 1    | 7                        | 8                |
| -10       | 23        | 33          | 11   | 6                        | 6                |
| 90        | 100       | 10          | -51  | 4                        | 8                |

**Table 3.9:** Table showing the effect of using a custom bias. The column  $\text{bits}(e_{range})$  represents the minimum bit-width of the exponent when using the custom bias. For comparison, the column  $\text{bits}(e)$  shows the required bit-width when using a standard bias.

conversions are performed to adjust the bias between inputs, outputs and operations. The rationale behind this is described in Section 2.1.4, with a demonstration of its effect shown in Table 2.2. To implement this element of the optimisation, we would need to consider the total range of exponent values<sup>2</sup>,  $e_{range} = e_{max} - e_{min}$ , selecting  $\text{bits}(e) = \lceil \log_2(e_{range} + 2) \rceil$ . The bias may then be set accordingly, as  $b = 1 - e_{min}$ . Table 3.9 shows some examples, starting with the double-precision standard. The expected exponent width with and without a custom bias is shown for comparison. As can be seen, depending on the circumstances this approach could save one or two bits on the exponent.

### 3.5 Summary

In this chapter we have described the three optimisations implemented to date: Operand Alignment Reduction, Normalisation Reduction and Exponent Reduction. Of these, Operand Alignment Reduction applies only to addition and subtraction. The hardware on which we base our optimisations was described, along with the key architectural changes we make to achieve them. Indicative hardware savings for each optimisation were supplied for single- and double-precision implementations, which range from 10% to 40%, depending on magnitude of optimisation and base representation.

These optimisations form one of the key elements of our overall technique, with the others being described in the following chapters.

---

<sup>2</sup>Remembering to include the ‘reserved’ values of all zero and all one in the exponent.

## 4 System Integration

In Chapter 3 we considered optimisations to be applied to individual floating-point units. These must be assembled and integrated with a host system in a way which enables both fast processing and error correction to take place. Methods for partitioning hardware and software were given in Section 2.4.3; it is anticipated that our technique would be compatible with many of these.

The details of how an application is partitioned between hardware and software is not directly of concern to us, as our optimisations take place at a low-level and could be deployed within any partitioned system. Inclusion of our optimisation steps as part of a larger tool-chain, such as Harmonic[LCT<sup>+</sup>09], would provide another route to system integration. However, we must be able to modify a partitioned system to include the error detection and recovery mechanisms our optimisations require.

Figure 4.1 shows a typical hardware/software stack, with our additions highlighted. It is intended that error recovery takes place within the application interface library, hiding the fact that errors occur from the application. In our concluding Chapter 7, we consider methods which may improve the effectiveness of our optimisations by tighter integration of the application and hardware.

The method in which this error detection and correction is implemented is vital: the success of our technique requires a trade-off between error rates and re-execution time.

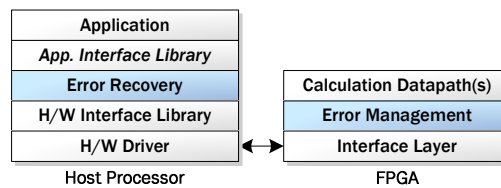


**Figure 4.1:** Software and hardware stacks with error detection/correction

## 4.1 Hardware Options

This section describes two configurations of optimised floating-point hardware and their processor integration methods, which are illustrated by Figure 4.2. We consider a small embedded processor with an optimised floating-point unit, and a streaming data-path coupled to a fully-fledged processor with hardware floating-point unit.

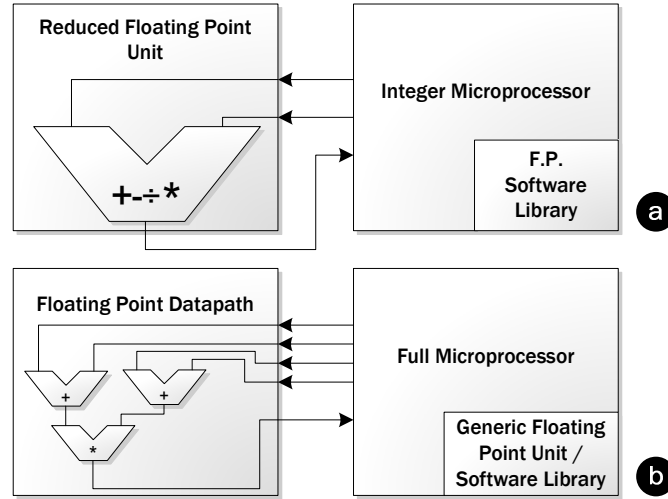


**Figure 4.2:** Integration methods with a host processor:

- a) A single optimised floating-point unit,
- b) An optimised floating-point data-path for a fixed set of operations.

### 4.1.1 Optimised Floating-Point Arithmetic Unit

The most basic integration method is an optimised floating-point arithmetic unit added to the processor, either through integration into the data path or as an external hardware device. This type of integration is more suited to embedded systems, where area and power savings are important. Rather than use a fully-functional floating-point unit a smaller one may be used to decrease overall cost. When referring to a floating-point unit, we allow that this may be a collection of several individually-optimised units combined into a single functional unit, rather than a blanket optimisation being applied to all components.

Under this design the same floating-point unit is used for all operations,

limiting the scope of optimisations: profiling results as seen in Section 5.3 show that value ranges of operations may be disjoint. As will be seen in the case study in Chapter 6, optimisation at the operation level has considerable advantages.

This style of integration may be suitable in a general-purpose case, where a specific data-path to be optimised cannot be identified, but there exists a range of calculations with similar behaviour, either within an application or across applications.

Due to its inherent limitations we do not consider this design further, but have mentioned it here for completeness.

#### 4.1.2 Optimised Data-path

Our second option is to implement a floating-point data-path consisting of a sequence of operations, for example the inner loop of a loop nest. We then stream operands in to the data-path and results out, either directly to/from a host processor, or via on-chip control logic. We have referred to a processor with a full floating-point unit on-board, which could be used to re-execute erroneous operations. This could be omitted if no general purpose FPU was required, with a software library providing a re-execution mechanism.

A streaming data-path is a common configuration in the literature, with tools such as StReAm[MHMF00] (later ASC[LT03, MPHL03, Men06]) and Streams-C[GSAK00, FGL01] providing a high-level language description. Bellas et al[BCDL06] eschewed the high-level approach for a template-based system, while highlighting the benefits of decoupled streaming data-path and data-fetch components.

Once again, we are less concerned with the precise nature of how a streaming data-path could be implemented, as long as they may be adapted to expose error detection signals. In our test cases we use an XML description and Perl script to generate a VHDL data-path. Data-fetching is implemented in Handel-C[Agi08] and compiled to VHDL for synthesis. Our XML description is derived from a single-static assignment form[ALSU06, CFR<sup>+</sup>91] of the application code to be implemented, as generated by the *Float-Watch* profiler.

The streaming approach provides a number of important advantages.

In particular, each operation may have a fully optimised implementation. Pipelining allows the units to be fully utilised despite the lack of re-use within a computation. Intermediate results also remain within the data-path, rather than being copied back and forth as in the single floating-point unit case.

This style of processor integration reduces communication requirements and generates a highly targetted solution. However, this makes it unsuitable if the hardware is required for general purpose floating-point calculations.

Figure 4.3 shows a data flow graph for part of the swim benchmark which will be shown in Chapter 6. Under this method each operation would be an optimised unit matching the profiling results for that application.

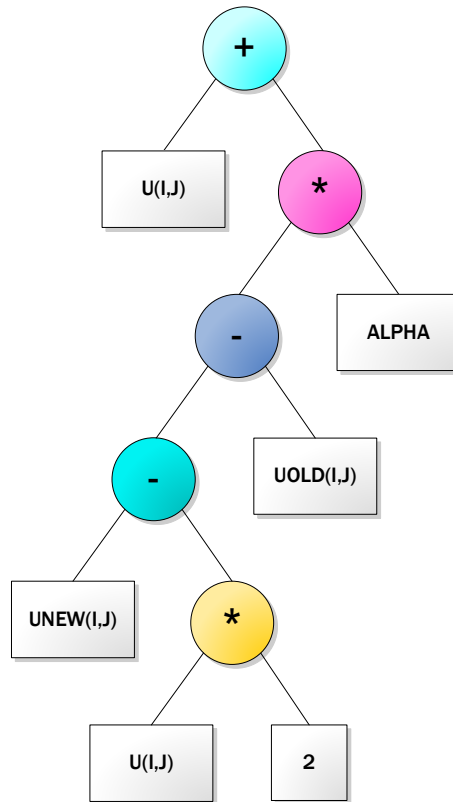


**Figure 4.3:** Flow Graph from the Swim Benchmark

Another point to note from this figure is that there are five operations but only four inputs are required, along with one output. With a single floating-point unit ten inputs and five outputs would be needed, with some results

being sent straight back to the processor. A set of floating-point registers in the floating-point unit option would resolve this wasted communication bandwidth, at the expense of more resources.

## 4.2 Error Recovery

On smaller devices there are two main options for re-execution, dependent on the architecture in use: either on a host or embedded processor or within a fully-capable FPU on the FPGA. For larger devices it is feasible to place an IEEE-754 compliant floating-point data-path alongside multiple cut-down paths, to provide an on-chip recovery option. Where the FPGA is coupled with a high-performance host processor, re-execution on the host is possible. The case study for *102.swim* in Chapter 6 uses the host processor to re-execute erroneous calculations in software.

The method of communicating with re-execution hardware provides a great deal of flexibility depending on the application being optimised. We present a variety of options, of which one - the Iteration Indexed Re-execution Buffer (IIRB) - is carried forward to implementation in Chapter 6.

A vital component of our system is the software layer on the host processor. This is responsible for configuring, then marshalling data to and from the acceleration hardware. When our system is in place it may also be required to re-order or re-execute operations as necessary.

### 4.2.1 Iteration Indexed Re-execution Buffer

The Iteration Indexed Re-execution Buffer requires a shared-memory system, the accelerator performing direct memory access to fetch operands and store results. It is designed for systems where loop iterations may be executed truly independently. In this configuration the acceleration hardware keeps track of the iteration number being executed, fetching operands using direct memory accesses. The host processor passes the hardware a pointer to an allocated shared memory area which is used as a re-execution buffer (RXB). When an error occurs, the iteration number is written directly to the shared memory area. The host may query the acceleration hardware to determine the number of items in this buffer and re-execute any iterations stored within it during execution of the accelerator.

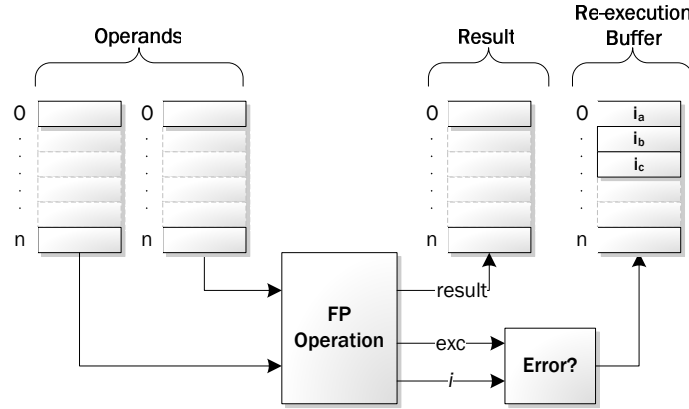


**Figure 4.4:** Functional Diagram of the Iteration Indexed Re-execution Buffer

The iteration number allows the host to select operands from memory, re-execute the calculation using its own floating-point mechanism (hardware units or software libraries), then write the result back to the correct location in memory.

The *102.swim* case study expands on this method in Section 6.2 with a concrete implementation.

#### 4.2.2 In-design Correction

On large devices the most application-friendly solution is to correct errors within the accelerator itself. This limits communication between host and accelerator and allows a simplified library at the application layer.

Initially, we consider an out-of-order system, where operations enter the accelerator in an application-defined order, but may return to the host in an arbitrary order. The application-level library dispatches operations to the accelerator with a tag, for example a loop iteration number or result destination address. Upon completion the result is returned to the host processor where it can be processed as appropriate for its tag. Under this system, shown in Figure 4.5, results are returned in-order until an error occurs, at which point a result becomes delayed until it has been re-calculated on the fully-capable floating-point unit/data-path.

Referring to Figure 4.5, we can see the following steps in this re-execution method:

1. Items from the input queue are distributed to the next available data-path;
2. Correct results are added directly to the output queue; operands for incorrect results are sent to the re-execution queue;
3. The fully-functional floating-point unit uses operands from the re-execution queue when available, or the input queue if not;
4. A correct result is generated by the fully-functional unit;
5. The output queue stores results, which may be out-of-order;

The key responsibilities of the software layer in this solution are to assign tags for operations, and place results returning from the accelerator in the correct location based on those tags. This may be as simple as re-ordering results before returning them to the application layer.

A hardware re-order buffer[Joh], as frequently used in superscalar processors, would be a logical addition to this system. Hardware would thus replace much of the software layer, with results being re-ordered before dispatch back to the host. The hardware could also assign tags, reducing the volume of host-accelerator communication required.

An accelerator based on the Tomasulo algorithm[Tom67] would be a further logical extension, allowing dependent loop operations to be issued together, waiting for erroneous results to be re-executed on-chip if necessary. This could increase the complexity of the accelerator by introducing the concept of registers, moving away from the streaming data-path scenario and towards a floating-point co-processor. However, the results output array could be treated as a set of registers, using the required array index as a tag for the purpose of the algorithm.

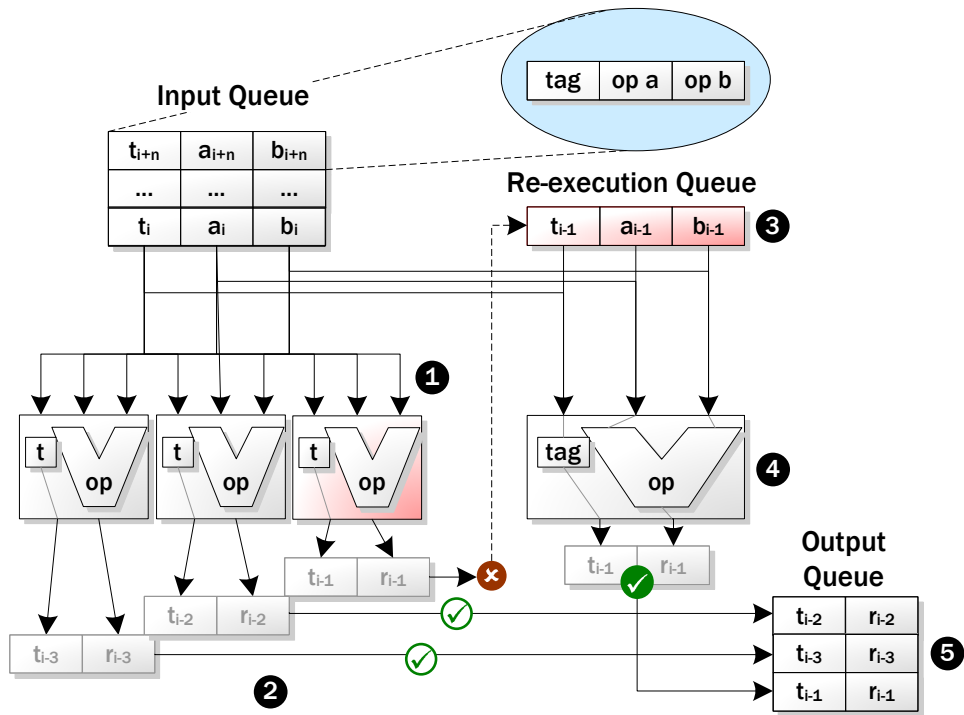


**Figure 4.5:** In-design correction, with multiple parallel data-paths.

### 4.3 Summary

In this chapter we have taken our unit-level optimisations, as shown in Chapter 3, and demonstrated ways they may be integrated into a complete system. It is these methods of system integration which permit speculative execution, with erroneous results generated by our units being re-executed by the surrounding system.

This is not an exhaustive list of options, as we anticipate many more system configurations being available depending on the underlying architecture and target application. The options described here provide a demonstration of the concepts involved, while providing a suitable base for proving the success of our optimisations.

## 5 *FloatWatch*: A Floating-Point Profiler

*FloatWatch* is the combination of a dynamic execution profiler within the Valgrind[NS03, NS07] framework, and a collection of processing and reporting tools. Aggregated data on the value ranges and numerical relationships between floating-point operands are gathered at runtime by the profiler, with post-processing producing a report file for use in a graphical user interface. This permits the exploration of profiling data on a program, source file, source line or individual-operation basis. Filtered and aggregated results can be exported for use in graphing tools.

*FloatWatch* has played a dual-role in our research: initially, as a tool for providing insight into the floating-point behaviour of scientific code and latterly to inform the optimisation choices we make. It may also be used to identify potential errant behaviour in floating-point software, a topic which we will return to shortly, and in more detail in Section 5.2.

We initially used *FloatWatch* to explore the behaviour of some floating-point codes, descriptions of which can be found in Section 5.3. From these initial explorations, our optimisation approach developed and the profiler was extended to support the optimisations described in Chapter 3.

The following data is collected for every floating-point operation:

- a value distribution measure, in the form of the exponent ( $e$ ) of the operands and result of each operation <sup>1</sup>,
- occurrences of zero and denormal values,
- an operand ratio measure, which is the difference in exponent between operands of floating-point operations ( $\Delta e$ ),
- the total shift required to normalise a floating-point result.

---

<sup>1</sup>Only the result for each operation is collected directly, however this includes floating-point loads so the magnitudes of floating-point operands are known.

The value distribution measure is used principally to inform the Exponent Reduction optimisation (Section 3.4), providing an experimental observation of the main value ranges in the application. However, our initial investigations with the profiler were to determine if alternative representations, such as fixed point, would be appropriate and the value-distribution is still useful for this purpose. The profiler tracks zero values and de-normal floating-point numbers as part of the value-collection process, which may be indicators of a sub-optimal algorithm or performance problems.

The operand ratio measure informs the Operand Alignment Reduction optimisation (Section 3.2), but can also identify a loss of precision. Within floating-point additions and subtractions the significand ( $s$ ) of the smallest operand is shifted by  $\Delta e$  bits in order to align both operands for a corresponding fixed-point operation. When the condition  $\Delta e \geq \text{bits}(s)$  is satisfied, the smaller operand loses all its significant bits and effectively becomes zero. This can be a cause of errant behaviour under certain conditions, particularly in algorithms involving iterative convergence.

Normalisation amount is an important metric to gather, as it informs the Normalisation Reduction optimisation (Section 3.3) applicable to *all* floating-point operations, and can also indicate the presence of catastrophic cancellation as described in Section 2.1.3.

Section 5.1 covers the implementation details of the tool. Section 5.2 provides more detail on the alternative uses for our profiler, expanding on the above. The results collected using *FloatWatch* have shown some interesting application characteristics, which can be found in Section 5.3. Chapter 6 presents additional results and illustrates the role they play in the optimisation process.

## 5.1 The Tool

*FloatWatch* operates on x86 binaries compiled with debugging information, under the Valgrind [NS03] dynamic instrumentation framework. Valgrind reads x86 and PowerPC binaries, converting them to an intermediate representation consisting of simple pseudo-instructions, similar to single static assignment (SSA) form. Complicated x86 arithmetic with memory operands is flattened to a sequence of loads, stores and arithmetic with temporaries. Basic blocks are processed one at a time, with each block passed to an instrumentation tool (*FloatWatch* in this case) which inserts, removes or modifies instructions as necessary. The intermediate representation is then converted back to machine instructions, cached and executed. Figure 5.1 illustrates the process.

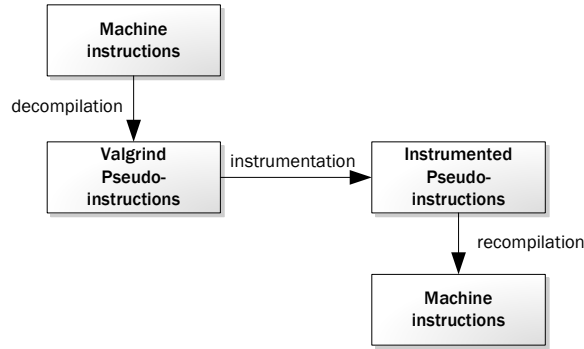


**Figure 5.1:** Valgrind instrumentation steps: decompilation, instrumentation and recompilation.

*FloatWatch* uses this framework to monitor the floating-point behaviour of a target application at runtime. The collection of profile data proceeds on every operation with a 64-bit floating-point return type. By default this will cover all such operations within an application, but may be limited to just specific function calls or source lines to speed up the process.

Figure 5.2 shows the instrumentation step. When using the *FloatWatch* tool, every floating-point operation has a call to `float_ratio` inserted after it, with the type of operation, result and operands passed as parameters. The

`float_ratio` function calculates and stores the required profile information from these parameters.

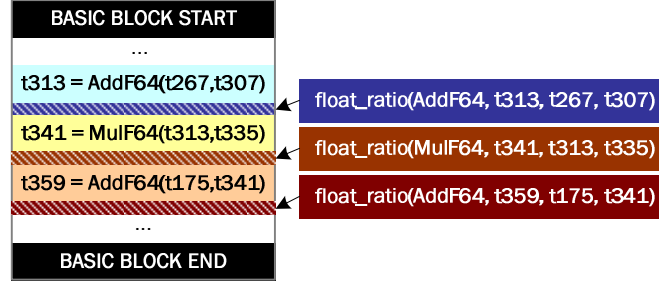


**Figure 5.2:** Instrumentation of floating-point operands with the `float_ratio` function.

It is impractical to store a detailed value history for every floating-point operation, due to the high execution count of such operations in the applications under examination. The application execution time is also increased by an order of magnitude, and would be increased further with the added overhead of writing a large dataset to disk. For this reason the profiler stores summary information only, which is all that is required for our optimisation purposes.

In the case of the value distribution measure, the profiler stores a count of every exponent seen for each floating-point operation. Measurements are performed on double-precision floating-point values, with an 11-bit exponent, giving a theoretical  $2^{11}$ -entry table for each operation should the maximum spread of values be used. In reality this does not occur (indeed, our optimisations rely on it not occurring) and the profiler dynamically allocates table entries to minimise memory use.

The operand ratio measure is calculated by taking the difference between exponents of two operands. This is the same as the shift amount required for operand alignment, and so an ideal measure for the optimisations we wish to perform. Once again this could lead to a  $2^{11}$ -entry table for each operation, however there are other restrictions which can be made. As previously mentioned, for double-precision operands an alignment shift of 53-bits or greater leads to that operand becoming zero. Consequently only shifts in the range  $[-52, 52]$  need to be tracked individually. Shifts falling outside this range are tracked and aggregated into positive and negative

buckets.

The normalisation shift measure is calculated by taking the difference between the exponent of the calculated result and that of the smaller operand. This avoids calculating the result separately, relying on the information already calculated. Results are stored in a table in the same way to the operand ratio measure.

After execution the tool creates a raw output file with the data it has collected, along with the intermediate representation of basic blocks with floating-point operations. This file contains a list of bucket/value pairs for each measure being profiled, for each operation in the program. The *FloatWatch* post-processor takes this output and combines it with the application source files, producing an XML report suitable for use in the *FloatWatch Explorer* graphical user interface. The data can be dynamically manipulated by the user to produce graphs of value ranges for particular lines of code. The values may then be exported to graphing software for use in reports.

The information can be aggregated for each line, then on a line-by-line basis by the user. Figure 5.3 shows the source display and value graph provided by the Java user interface.



**Figure 5.3:** Exploring profile results using the *FloatWatch Explorer* interface. Each highlighted line of source code can be expanded to show assembly-level code. Each line's floating-point value distribution can be selected for graphical display.

## 5.2 Alternative Uses

The design rationale of *FloatWatch* was approached with our optimisations in mind, however the information it reveals can also identify errant floating-point behaviour. Section 2.1.3 describes the nature of this behaviour in more detail, while this section will focus on how it can be identified from profiling results.

*FloatWatch* is able to identify the following problems:

1. Loss of accuracy due to large numbers being added to or subtracted from small numbers;
2. Loss of precision resulting from catastrophic cancellation;
3. The presence of denormal numbers in a calculation;
4. Excessive calculation of zero values.

### 5.2.1 Accuracy Loss

The operand ratio measure can identify loss of accuracy resulting from adding or subtracting a mixture of large and small numbers. This occurs when the difference of the exponents of two operands is greater than or equal to the bit-width of the significand (including the implicit leading one), that is  $\Delta e \geq \text{bits}(s) + 1$ . The smaller of the two is thus shifted by such a degree that all significant bits are lost.

This behaviour may be identified by examination of the operand ratio graph produced by *FloatWatch*. Operand ratios of 53 or above indicate total loss of significant bits for the double-precision format, which *FloatWatch* monitors. Ratios approaching this value also indicate a high loss of significant bits.

### 5.2.2 Precision Loss from Cancellation

The second form of erroneous behaviour is precision loss due to catastrophic cancellation of similar values. Section 2.1.3 describes when this occurs and presents an example, which we will now explain. In this example,  $\frac{1-\cos(x)}{x^2}$ , the  $\cos(x)$  term becomes close to 1, leaving the numerator,  $1 - \cos(x)$ , with values in only the least significant bits. During normalisation the exponent

is adjusted and the least significant bits are shifted to the most significant positions, the value 0 filling the remaining bits. In double-precision the result is thus left with a 52-bit significand, of which the majority of bits are 0 and carry no information about the number. This problem is then compounded with the division by  $x^2$ , which leads to a large multiplication since  $x$  is a small number less than one, and consequent escalation of the error.

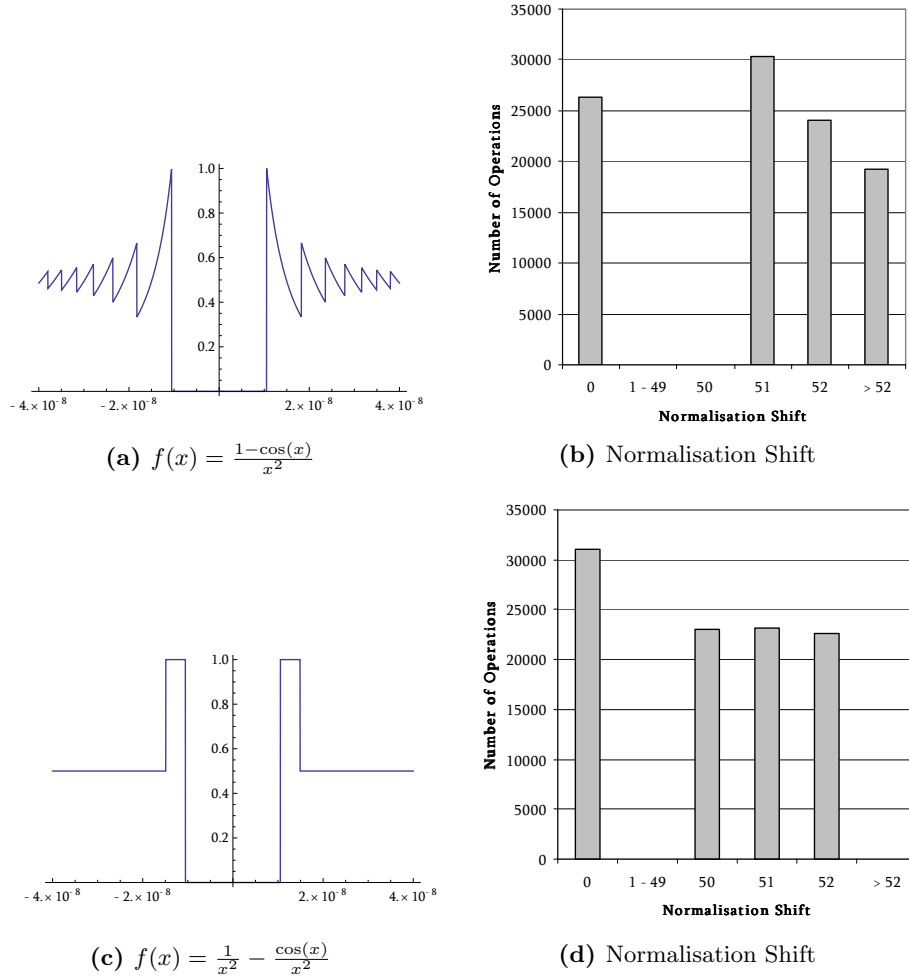


**Figure 5.4:** Normalisation shift profiles for the errant behaviour of  $f(x) = \frac{1-\cos(x)}{x^2}$  and the equivalent  $f(x) = \frac{1}{x^2} - \frac{\cos(x)}{x^2}$ .

The normalisation shift measure generated by *FloatWatch* can reveal when this behaviour is likely to occur. Once again, it is the size of the significand which is of interest: as the shift measure approaches and exceeds the value of  $\text{bits}(s) + 1$ , it indicates that cancellation is occurring, with least significant bits being promoted to significance.

Figure 5.4 shows the distribution of shifts required to normalise the result of the subtraction term in both the original configuration and its rearrangement,  $\frac{1}{x^2} - \frac{\cos(x)}{x^2}$ . As can be seen, when the result does not become zero, shifts of 52 or greater bits are required. A double-precision floating-point number has 53 bits of significand, including the implicit leading 1 which is not stored in memory. As a consequence this calculation achieves between zero and one bits of precision (when the result is not zero), a problem further exacerbated when dividing by  $x^2$  in the next step. The result falls to zero when 1.0 and  $\cos(x)$  fall within the same interval and are represented by the same 52-bit significand and 11-bit exponent.

The re-arranged version (Figures 5.4c and 5.4d) shows slightly different behaviour, with normalisation shifts between 50 and 52, retaining more precision. As a consequence it has fewer problems, with a value of 0.5 for much of the range. However, it also experiences a failure by dropping to zero.

### 5.2.3 Denormal Numbers and Zero Values

The presence of a large quantity of denormal numbers or zero values can indicate performance problems. A brief explanation of denormal numbers can be found in Section 2.1. Calculations with denormal numbers may have an adverse effect on performance, because typical processor designs implement support for them in software, whilst custom hardware designs may not implement them at all. Compilers may switch off support for denormals to increase performance at the expense of accuracy: Intel’s ICC compiler[Int08] makes use of this ability in its `-fpmath=fast` option.

Identifying the use of denormal numbers allows hardware to be produced for such numbers if desirable, or the underlying code to be modified to avoid them, where possible. Scaling operands is one method of avoiding denormals, by increasing them out of the denormal range during calculation, then reducing them back down afterwards. Multiplications and divisions

must take into account elimination or duplication of the scaling factor, but performance problems can be limited while accuracy is retained.

Figure 5.5 shows the effect of denormal numbers on the performance on three common desktop processors: Intel Pentium IV, AMD Opteron and the IBM PowerPC-based Apple G5. A convolution was performed on a dataset using a convolution kernel with progressively smaller numbers, until they entered the denormal range. The Opteron is the worst performer, with performance dropping off even before the denormal range is reached. On the Pentium IV performance drops as the denormal range is entered, producing over a 20 $\times$  slow down. The Apple G5 executes at just half the speed in the denormal range, not exhibiting the catastrophic degradation in performance seen in the x86-based processors.

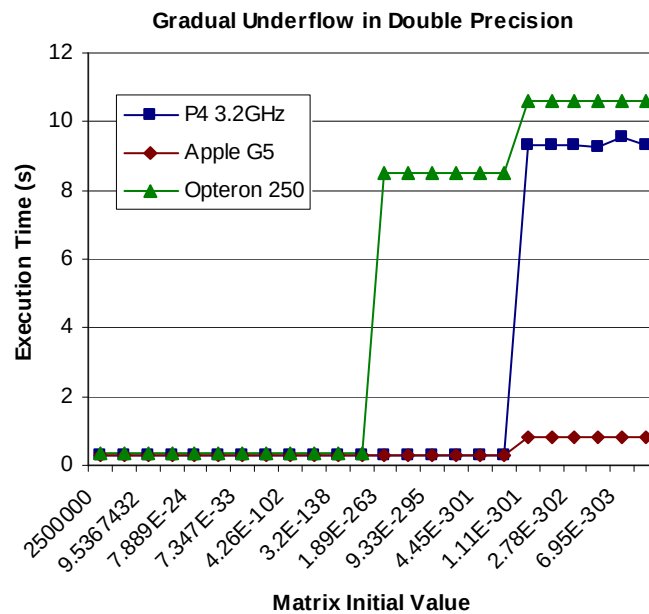


**Figure 5.5:** The Effect of Gradual Underflow. Convolution with a kernel containing progressively smaller values. Performance on Intel and AMD processors falls off dramatically as values enter the subnormal range. On the Opteron, performance begins to fall off even before this range.

Denormal numbers are tracked separately by the profiler, allowing their presence to be easily identified.

The occurrence of many zero values has a far more fundamental explana-

tion: it indicates that calculations are being performed that may be unnecessary. An alternative algorithm may be desirable, for example using sparse matrix methods instead of a full-matrix calculation. Once again, zeroes are clearly identifiable as they are tracked separately by the profiler.

## 5.3 Results

In the following section we present a selection of profiling results for applications including some SpecFP95 benchmarks and the FFMPEG video encoding and decoding library. These results assisted with the development of the profiler and our optimisation technique.

Further results are presented in Chapter 6, showing the application of profiling information to the optimisation process.

The results shown in this section are for the whole run of an application, however our profiler allows characteristics to be determined on an operation-by-operation basis, enabling extensive customisation of individual data-path stages.

### 5.3.1 MORPHY

MORPHY[Pop96] is a commercial application under development at the University of Manchester which performs an automated topological analysis of a molecular electron density. It has two modes, fully analytical and semi-automatic, with the semi-automatic method running faster but not always able to produce a result.

The application was run with data for water, peroxide and methane molecules. Figure 5.6 shows the value ranges for this application, using the semi-automatic method. The y-axis shows the fraction of values falling within the range shown – the absolute number of calculations performed varies between the datasets.

Examination of the results reveals some interesting features. Firstly, the graph has two distinct ranges, one either side of zero. These ranges are slightly asymmetric, but the overall range is very narrow. This indicates the potential for a successful fixed-point implementation. We can also see that very few zero values are generated, as the graph contains no central spike.

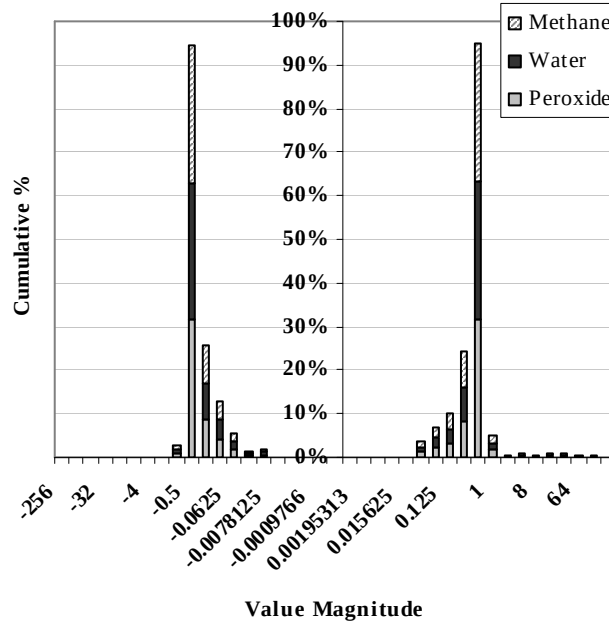


**Figure 5.6:** Profile results for ‘MORPHY’. This graph shows the core ranges of values – some values fall out of the range shown on the graph, however very sporadically.

Additionally, the ranges are similar across the three datasets tested, indicating that the same type of optimisation would be suitable across these datasets.

### 5.3.2 SpecFP95 Benchmarks

Figure 5.7 presents operand ratio profiling results for four benchmarks from the SpecFP95 suite, selected for their varied characteristics. From this figure the *tomcatv* benchmark immediately has a problem for our operand-alignment optimisation, as only 70% of operations fall within the shift limit of 8 places, which translates to a saving of three shifter stages. A shift of 24 places is required to keep 90% of operations within the executable range, meaning five stages would be required. Further analysis of the data at an operation-level is possible, which may show that isolated elements of the data-flow graph are successfully optimisable.

The *hydro2d* and *wave5* benchmarks immediately look more promising, as they require a shift of no more than 8 places for over 90% of their operations,

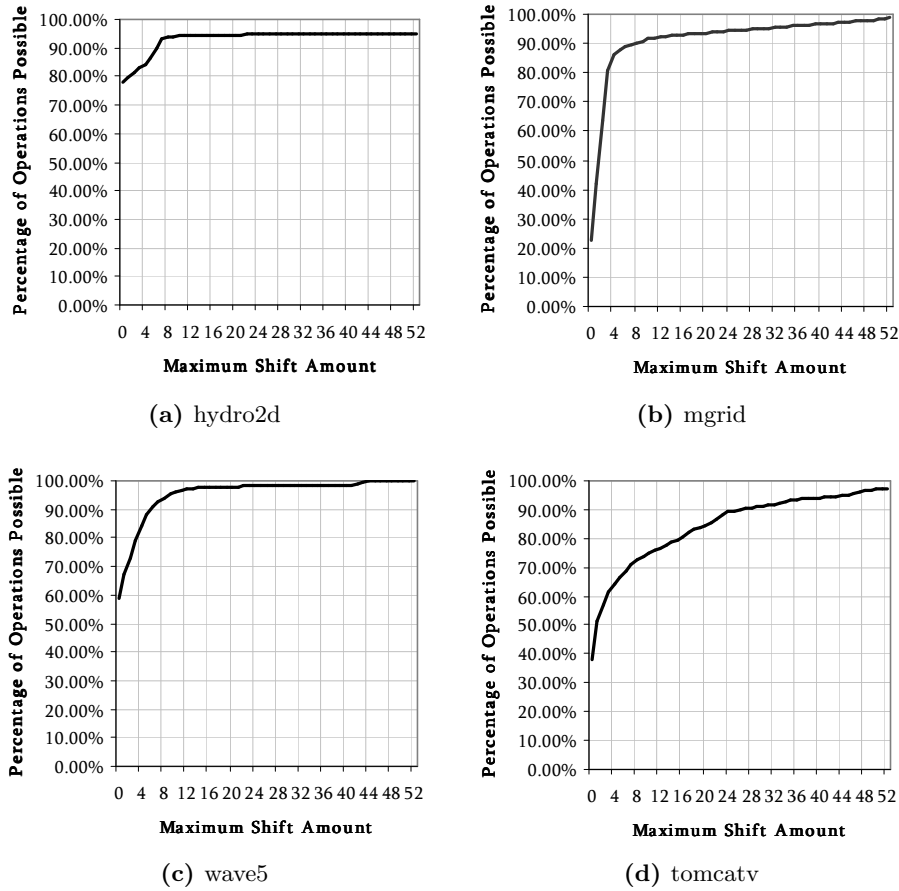


**Figure 5.7:** The percentage of operations able to execute without errors with a given maximum shift for SpecFP95 benchmarks.

while *mgrid* reaches 90% with an 8-place shift. All three would therefore be good candidates for our technique.

Given a suitable speed-up in the optimised case, *hydro2d* may be suitable for a further reduction in maximum shift amount, with 77.9% of operations requiring no shift at all. A secondary consequence of requiring no shift is the potential to remove the “swap operands” stage in floating-point adders and subtractors, which Figure 2.3 indicated made up 30% of these units. With operand alignment and swapping missing, what is left is effectively a fixed-point circuit with post-execution normalisation and rounding.

We have a further result for *mgrid*. Figure 5.8 shows the full value-range profile. The pronounced spike in the centre indicates zero numbers are

being generated, pointing to a possible performance problem as described in Section 5.2.

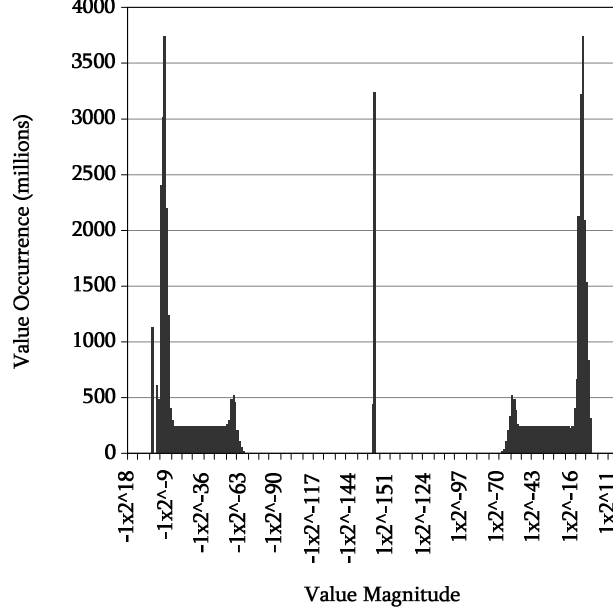


**Figure 5.8:** Value-range profile results for *mgrid*

### 5.3.3 FFMPEG and Runtime Reconfiguration

In addition to our focus on scientific applications and benchmarks, we have profiled the open-source video transcoder FFMPEG[Bel]. The video codec library at the heart of this application, libavcodec, is used in a variety of other applications[FdLB<sup>+</sup>03, Ger03] and utilises floating-point calculations for video processing, rather than fixed-point alternatives found in some embedded systems. The tests presented here involved transcoding from a source video to MPEG-2 and Flash video formats.

The three test videos and their characteristics are detailed in Table 5.1.

Figure 5.9 shows three operand ratio graphs, one for each of the three videos. Each graph shows the maximum operand shift required for transcoding to both target formats.

This set of graphs illustrates where a static hardware configuration is limiting, with reconfiguration allowing greater flexibility. The VPT and Jez datasets show similar behaviour to each other, and both have the same input

| Video         | Bitrate   | Video               | Audio                      | Content                                                              |
|---------------|-----------|---------------------|----------------------------|----------------------------------------------------------------------|
| <b>VPT</b>    | 3,112kb/s | WMV3,<br>720 × 576  | WMA2,<br>48kHz,<br>128kb/s | Single camera with fixed camera position, limited in-frame movement. |
| <b>Jez</b>    | 2,489kb/s | WMV3,<br>720 × 576  | WMA2,<br>48kHz,<br>128kb/s | Fixed backgrounds with text, panning, camera changes and movement.   |
| <b>LMS.02</b> | 298kb/s   | MPEG4,<br>336 × 240 | MP2,<br>44.1kHz,<br>64kb/s | Multiple cameras, panning and in-frame movement.                     |

**Table 5.1:** Characteristics of three test videos

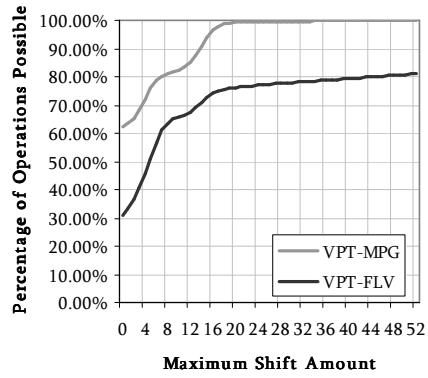
video format. When converting to MPEG-2 format, a maximum shift of 8 places allows over 80% of operations to be executed for both of these inputs, with a shift of 16 allowing over 95%. Contrasting this to the LMS02 input, which is of lower resolution and bit rate, we see that a shift of 20 places is required to cater for 80% of operations, while exceeding 95% requires 32.

A similar difference is seen when encoding to the Sorenson Spark format, used in Adobe’s Flash player. As can be seen from Figure 5.9, only 80% of operations for the VPT and Jez inputs and 70% of operations for the LMS02 input are possible with a maximum shift of 52 places. Double-precision floating-point has a 52-bit significand, so for Flash encoding 20-30% of operations result in one operand “disappearing”, as all significant bits are shifted out.

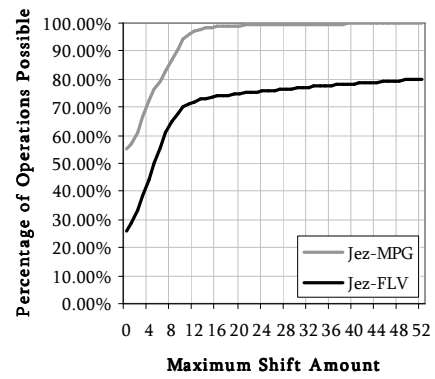
These variations in behaviour are dependent on both input and source formats, so prevent the use of a static optimisation. A speculatively reduced floating-point unit suitable for one input source (for example VPT converted to MPEG) could lead to frequent re-executions when used with a different source (for example, LMS02 converted to Flash). In this very severe case a unit with a maximum shift of 16 places would go from an error rate of less than 5% to one of more than 40%.

For this situation we propose that a variety of optimised units are available, ranging from aggressive optimisation suitable for the VPT/Jez case to standard floating-point implementations for the Flash video case. In the FFMPEG example presented here it is possible to select a suitable optimisation based initially on the input and output formats, with the possibility of run-time reconfiguration should the error rate become too high to achieve

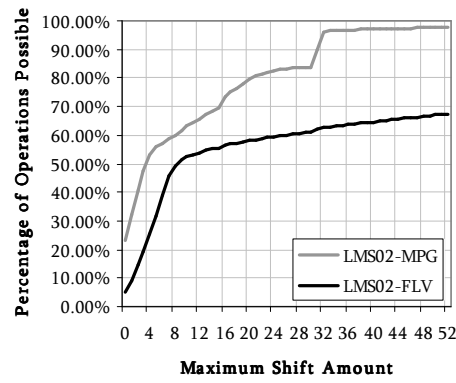
a net speed increase.



(a) VPT



(b) Jez



(c) LMS02

**Figure 5.9:** The percentage of operations able to execute without errors with a given maximum shift for FFMPEG.

## 5.4 Summary

In this chapter we covered the final key part of our optimisation technique: the *FloatWatch* profiler. We described the underlying Valgrind-based architecture, supporting tools for data analysis and how the data collected can inform our optimisation choices. Results for a selection of applications were provided, with commentary on how they could be interpreted.

A number of alternative uses for *FloatWatch* were also presented, demonstrating its wider application outside of our optimisations. In particular, it may be used to identify problematic behaviour resulting from the inherent limitations of the floating-point format, such as catastrophic cancellation.

## 6 Case Study: SPEC FP95

### *102.swim*

We have used a variety of applications for the initial analysis of our optimisation technique, a small selection of which have already been discussed in Section 5.3. We have chosen one promising example for a more thorough demonstration, the *102.swim* benchmark from SpecFP95. This benchmark, suitable for supercomputers fifteen years ago, can be run with small modifications on an Altera Nios II processor[Alt09c], a ‘soft-core’ processor for Altera FPGAs. Our target system used this processor on the Altera DE2 board, which has an Altera Cyclone II EP2C35 FPGA and 8-MBytes of SDRAM. We describe this system in more detail in Section 6.2.

The benchmark is a weather predictor based on a paper by Sadourny[Sad75]. In terms of structure, it consists of a series of  $M \times N$  arrays, with three primary calculation sub-routines, named `CALC1`, `CALC2` and `CALC3`. Each of these sub-routines contains a double loop iterating over elements of the arrays. The benchmark uses  $512 \times 512$  arrays by default, while the original code on which it is based used  $128 \times 128$  arrays. In our study we reduced the size to  $128 \times 128$  for the profiling stage, upon which we based our optimisation decisions. The same hardware was then tested with  $64 \times 64$ ,  $256 \times 256$  and  $320 \times 320$  arrays. There is insufficient memory on our target architecture to use the  $512 \times 512$  configuration.

Section 6.1 provides an analysis of application behaviour, showing the results of profiling with *FloatWatch*, along with code inspection and conventional profiling output. Section 6.2 describes the system architecture of our streaming accelerator, including the re-execution method used. Section 6.3 presents the results of the case study, including possible hardware savings and observed re-execution rates for non-profiled datasets.

## 6.1 Analysis

Analysis of the code behaviour was performed using GNU `gprof` and our *FloatWatch* profiling tool on an Intel Core 2 Duo. The binary was compiled under `gfortran` v4.2.3, using options `-pg -O0`. The original FORTRAN-77 source was used as input to the compiler.

Each array is declared in FORTRAN to be of type `REAL`, which uses the single-precision floating-point format for storage. A standard compilation under `gfortran` produces code for execution with x87 floating-point instructions, which operate on double-precision<sup>1</sup> operands only, executing them in extended precision (80-bit) on the x86 architecture[Int09]. To achieve a full profile this is the desired behaviour, as limiting calculations to 32-bit single-precision prevents the identification of potential loss of accuracy as discussed in Section 2.1.3.

| Subroutine | Percentage of Execution Time |
|------------|------------------------------|
| CALC1      | 42.15                        |
| CALC2      | 33.88                        |
| CALC3      | 23.97                        |
| MAIN       | 0.00                         |
| CALC3Z     | 0.00                         |
| INITAL     | 0.00                         |

**Table 6.1:** Execution Time Profile

Table 6.1 shows the percentage of program run-time spent in each subroutine, as reported by `gprof`. The subroutine `CALC3Z` provides a variant of `CALC3` specific to the first iteration of the program and `INITAL` performs initialization of the arrays. From the table, `CALC1` uses the most execution time, followed by `CALC2` and then `CALC3`. While `CALC1` would therefore be a logical starting point for optimisation, examination of the source code reveals that `CALC3` has a number of features which make it more amenable to optimisation using our approach. Figures 6.1–6.3 show the main loops for each of the subroutines. It is clear from the code that `CALC3` has a very simple structure - lines 5, 6 and 7 differ only by the variable names, meaning a single hardware path would be directly applicable to all three. The remaining lines of the loop body (8, 9 and 10) are simple copies which

---

<sup>1</sup>DOUBLEREAL or REAL\*8 in FORTRAN

---

```

1  SUBROUTINE CALC1
2  ...
3    DO 100 J=1,N
4    DO 100 I=1,M
5      CU(I+1,J) = .5*(P(I+1,J)+P(I,J))*U(I+1,J)
6      CV(I,J+1) = .5*(P(I,J+1)+P(I,J))*V(I,J+1)
7      Z(I+1,J+1) = (FSDX*(V(I+1,J+1)-V(I,J+1))-FSDY*(U(I+1,J+1)
8      1      -U(I+1,J)))/(P(I,J)+P(I+1,J)+P(I+1,J+1)+P(I,J+1))
9      H(I,J) = P(I,J)+.25*(U(I+1,J)*U(I+1,J)+U(I,J)*U(I,J)
10     1      +V(I,J+1)*V(I,J+1)+V(I,J)*V(I,J))
11    100 CONTINUE
12  ...
13  END

```

---

**Figure 6.1:** CALC1 Source – Main Loop

---

```

1  SUBROUTINE CALC2
2  ...
3    DO 200 J=1,N
4    DO 200 I=1,M
5      UNEW(I+1,J) = UOLD(I+1,J)+
6      1      TDTSS*(Z(I+1,J+1)+Z(I+1,J))*(CV(I+1,J+1)+CV(I,J+1)+CV(I,J)
7      2      +CV(I+1,J))-TDTSDX*(H(I+1,J)-H(I,J))
8      VNEW(I,J+1) = VOLD(I,J+1)-TDTSS*(Z(I+1,J+1)+Z(I,J+1))
9      1      *(CU(I+1,J+1)+CU(I,J+1)+CU(I,J)+CU(I+1,J))
10     2      -TDTSDY*(H(I,J+1)-H(I,J))
11      PNEW(I,J) = POLD(I,J)-TDTSDX*(CU(I+1,J)-CU(I,J))
12     1      -TDTSDY*(CV(I,J+1)-CV(I,J))
13    200 CONTINUE
14  ...
15  END

```

---

**Figure 6.2:** CALC2 Source – Main Loop

require no floating-point operations.

In contrast to **CALC3**, the other subroutines would require unique hardware paths for most lines of code, with **CALC1** also including a division which would require numerous hardware resources.

Figure 6.4 shows a histogram of value occurrence for one of the lines of code within **CALC3**, broken down into individual operations. Without this breakdown the graph is misleading: it appears from the statement-level graph that operand values are widely spread. Viewing the data operation-by-operation reveals that each one has a smaller range of operand values<sup>2</sup> – multiplications within the statement move this range around, resulting in

---

<sup>2</sup>Each operation is a Valgrind pseudo-instruction, as discussed in Chapter 5.

---

```

1  SUBROUTINE CALC3
2  ...
3  DO 300 J=1,N
4  DO 300 I=1,M
5      UOLD(I,J) = U(I,J)+ALPHA*(UNEW(I,J)-2.*U(I,J)+UOLD(I,J))
6      VOLD(I,J) = V(I,J)+ALPHA*(VNEW(I,J)-2.*V(I,J)+VOLD(I,J))
7      POLD(I,J) = P(I,J)+ALPHA*(PNEW(I,J)-2.*P(I,J)+POLD(I,J))
8      U(I,J) = UNEW(I,J)
9      V(I,J) = VNEW(I,J)
10     P(I,J) = PNEW(I,J)
11 300 CONTINUE
12 ...
13 END

```

---

**Figure 6.3:** CALC3 Source – Main Loop

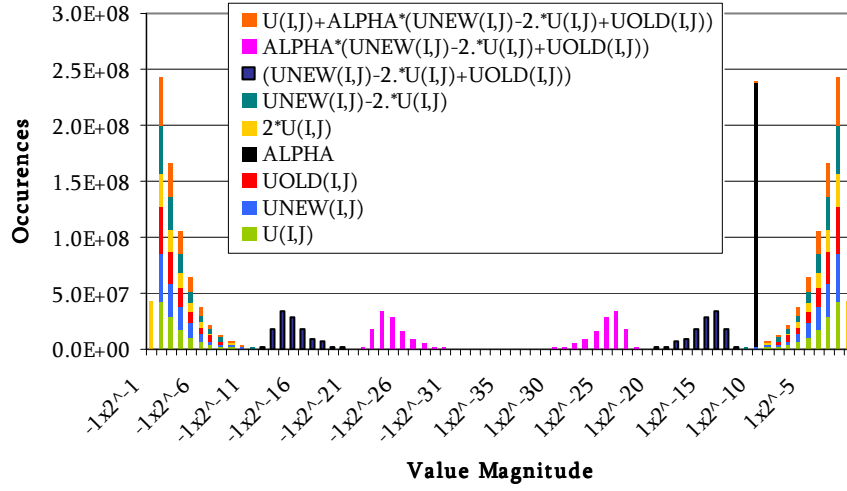
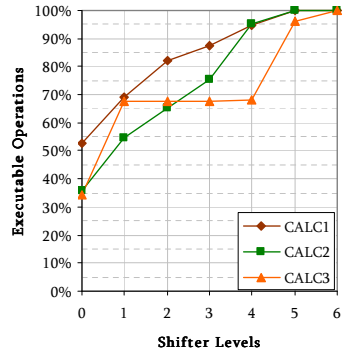


**Figure 6.4:** Operation-level profile of CALC3 inner loop

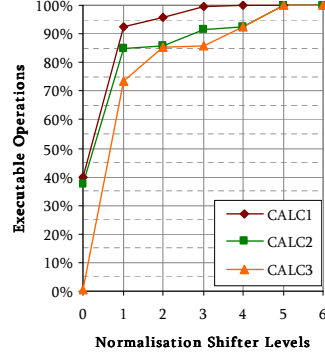
multiple peaks on the graph. Examining the graph further reveals all the values for ALPHA have the same magnitude, causing a large spike to the right of the graph. In the source, ALPHA is a constant.

Profiling results in this section are presented in a summarised form which directly relates to our optimisations. Figures 6.5a and 6.5b show the cumulative percentage of operations which could be executed for a given number of shift levels, for operand alignment and normalisation, respectively. Figure 6.5c shows the number of operations which could be correctly executed with a given exponent bit-width.

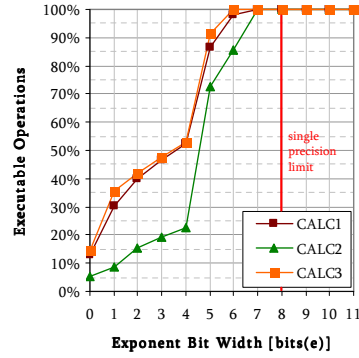
These figures expose a number of characteristics that inform the optimisation process. Firstly, Figure 6.5c shows that no floating-point result has



(a) Executable operations for each alignment shift level



(b) Executable operations for each normalisation shift level



(c) Executable operations for each exponent bit-width

**Figure 6.5:** *FloatWatch* profiling results for *102.swim* - each chart shows the percentage of operations which can be successfully executed given a specified hardware limitation.

a value which exceeds the maximum representable when  $\text{bits}(e) = 7$ . Furthermore, in **CALC3** only 0.06% of values exceed the maximum representable with  $\text{bits}(e) = 6$ . Hardware floating-point units could therefore be reduced to just seven bits for the exponent with no errors, or six bits with few errors. From these results **CALC3** appears to be the best target for optimisation: it makes up nearly one-quarter of the execution time of the benchmark, can be reduced to a 6-bit exponent with negligible re-execution rate and has a simple code structure.

Examination of the normalisation and operand alignment results is less conclusive. When the results for each operation in a FORTRAN sub-routine are combined, they suggest that with any form of optimisation, error rates would quickly become high. Figure 6.5a, as presented, indicates that a uniform operand alignment optimisation on all the adders and subtractors required to implement **CALC3** would be unsuccessful - while five levels of shifter would produce only a small re-execution rate, dropping to four would see this exceed thirty percent. The same applies to Figure 6.5b, but for Normalisation Reduction rather than Operand Alignment Reduction. However, uniformity across the floating-point units is not required in a streaming datapath, since each one can be customized specifically for the operation it will perform.

We will return to operation-level profiling, after a minor detour at the sub-routine level. Figure 6.6 shows the full results for *FloatWatch*'s operand alignment shift measure, with the single-precision limit highlighted. This shows that in **CALC3**, only 74.3% operations in the loop body fall to the left of the single-precision limit. To the right of this limit, the smallest operand in an addition or subtraction would lose all of its significant bits and become zero, as discussed in Section 2.1.3. In order to retain at least one significant bit, a 35-bit significand is required. The arrays being operated on are defined in the FORTRAN source code as the single-precision type **REAL**, however calculations with just a 23-bit significand, as defined for this type, would produce a less accurate result due to the loss of all significant bits.

Having concluded that the **CALC3** subroutine is a suitable candidate for acceleration using an optimistically-reduced floating-point implementation, the next step is to identify the precise nature of the hardware to be generated. We can see that the code fragment in Figure 6.3 may be trivially

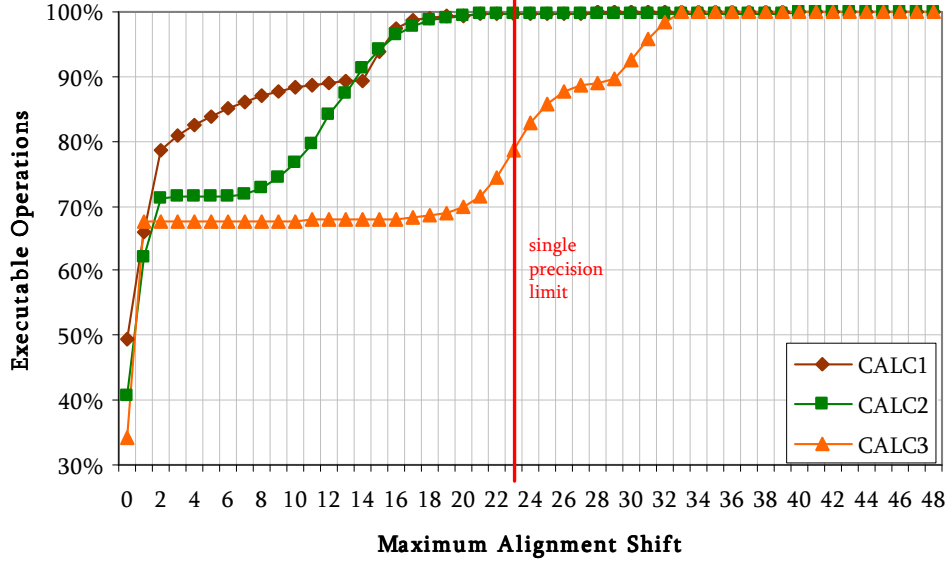


**Figure 6.6:** Percentage of operations which can be executed successfully for a given maximum alignment shift ( $\Delta e$ ) in *102.swim*

rewritten to that in Figure 6.7, in order to separate the floating-point calculation (lines 5, 6 and 7) from the rest of the loop body.

We now have three statements in the loop body which differ only by the names of the variables at each position. In addition, there are no dependencies between statements within the loop body, nor any loop-carried dependencies. Consequently we can focus on generating hardware for a statement of the form

$$r = a + \alpha(b - 2 \times a + c), \quad (6.1)$$

where:

$$\{r, c\} = \{\text{UOLD}, \text{VOLD}, \text{POLD}\}(I, J),$$

$$a = \{\text{U}, \text{V}, \text{P}\}(I, J),$$

$$b = \{\text{UNEW}, \text{VNEW}, \text{PNEW}\}(I, J)$$

and

$$\alpha = 0.01.$$

The value of the variable **ALPHA** remains constant throughout the execu-

---

```

1  SUBROUTINE CALC3
2  ...
3    DO 300 J=1,N
4    DO 300 I=1,M
5        UOLD(I,J) = U(I,J)+ALPHA*(UNEW(I,J)-2.*U(I,J)+UOLD(I,J))
6        VOLD(I,J) = V(I,J)+ALPHA*(VNEW(I,J)-2.*V(I,J)+VOLD(I,J))
7        POLD(I,J) = P(I,J)+ALPHA*(PNEW(I,J)-2.*P(I,J)+POLD(I,J))
8    300 CONTINUE
9
10 c Separate memory copy
11 DO 310 J=1,N
12 DO 310 I=1,M
13     U(I,J) = UNEW(I,J)
14     V(I,J) = VNEW(I,J)
15     P(I,J) = PNEW(I,J)
16 310 CONTINUE
17 ...
18 END

```

---

**Figure 6.7:** CALC3 Source – Re-written Main Loop

tion of the benchmark, having been initialised from an input file. The input files use the same value of 0.01, hence the value of  $\alpha$  in Equation 6.1 is considered constant and can be hard-coded in the final hardware. Alternatively, it could be initialised before execution begins.

Figure 6.8 shows a data-flow graph for Equation 6.1, with each operation named for later reference. The independent nature of each loop iteration lends itself well to a streaming data-path implementation as described in Section 4.1. In such a configuration, each operation in the data-path translates directly into a floating-point unit in hardware, allowing optimisation at a per-operation level.

It is at this stage that we re-introduce the profiling results, examining the floating-point behaviour for each of the data-path elements. Figure 6.9a shows the distribution of  $\log_2(\Delta e)$  values for each addition and subtraction in the data-path. The value of  $\log_2(\Delta e)$  is the number of shifter levels required to perform a shift of  $\Delta e$  using the VFLOAT libraries. As can be seen, the operations **sub1** and **add1** require only one level of shifter to achieve correct results, while **add2** requires a full-range shifter. In a similar way, Figure 6.9b shows the distribution of required shift levels for normalisation. The data-path element **mul1** is not shown, as it is a fixed multiplication by two and hence optimised away.

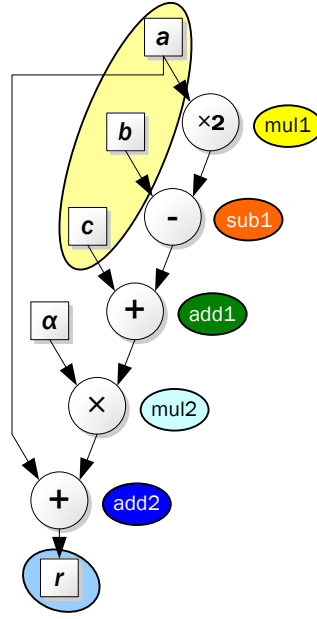
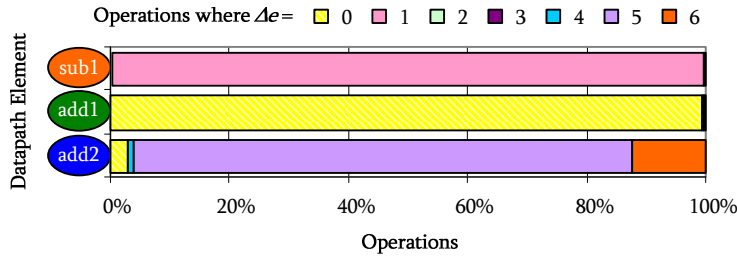


**Figure 6.8:** Common Data-flow Graph for CALC3 Loop Body (inputs and outputs highlighted yellow and light blue, respectively)

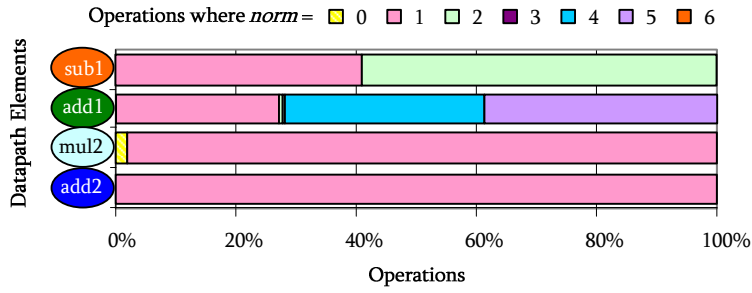
We consider the following optimisation options for this data-path:

- **OAR- $n$**  - Operand Alignment Reduction, with  $n$  shift levels for all addition and subtraction units.
- **OAR-PROFILED** - Operand Alignment Reduction based on unit-level profiling information, as described above. **add2** uses a fully capable shifter, **sub1** and **add1** use a reduced-capability single-level shifter.
- **NR-PROFILED** - Normalisation logic reduction, based on unit-level profiling information.
- **ER- $n$**  - Exponent Reduction, every floating-point unit has an exponent bit width of  $n$ .

We must consider one small variation on Operand Alignment Reduction at this point. In Section 3.2 we described how a given  $\Delta e$  may be outside the shift range of a reduced floating-point unit, but this is only an error when  $\Delta e < \text{bits}(s)$ , that is, the total amount to shift is less than the number of



(a) Distribution of  $\log_2(\Delta e)$  values for each addition/subtraction in the **CALC3** data-flow graph



(b) Distribution of normalisation shift levels for each unit in the **CALC3** data-flow graph

**Figure 6.9:** Summarised **CALC3** Profiling Results - Operation Level

| Operation | Significand Check |        |
|-----------|-------------------|--------|
|           | No                | Yes    |
| sub1      | 0.23%             | 0.23%  |
| add1      | 0.20%             | 0.20%  |
| add2      | 97.1%             | 35.42% |

**Table 6.2:** Expected Error Rates from Profiling Data for optimisation option OAR-1, with and without a check on the mantissa size.

bits in the significand. When this is not the case the significand becomes zero.

Table 6.2 gives the expected error rate for optimisation option OAR-1, for each addition and subtraction in the data-path. Options with and without a check against bits(*s*) are shown. The operations `sub1` and `add1` show no difference in expected error rate, so could potentially be implemented without the check in place. On the other hand, `add2` shows a difference of over 60% so would benefit greatly from the check.

## 6.2 System Architecture

In this section we describe the architecture of our accelerator and its interaction with the host system. The host processor is an Altera Nios II/f[Alt09c] with no embedded floating-point unit. The processor has instruction and data caches, branch prediction, an integer hardware multiplier and divider and hardware barrel shifter. This is connected via Altera’s Avalon[Alt09a] switch fabric to 8MB of SDRAM adjacent to the Cyclone II FPGA, an EP2C35F672C8 device. A phase-locked loop provides clock scaling from a native 50MHz oscillator to the 70MHz operating frequency of the system. This host system is used without an accelerator to provide a software-only base case for hardware size comparisons.

Our accelerator is configured as a component within Altera’s SOPC Builder System and attaches to the Avalon switch fabric as a master peripheral, sharing access to the SDRAM with the host processor. There is no pipelining of the accelerator, which operates at the system speed of 70MHz. The base components are capable of being pipelined, however the system is bound by memory bandwidth so the additional pipelining resources are not necessary. An architecture with greater memory bandwidth could exploit the pipelining capabilities. On-chip multiplier units are used to implement fixed-point multiplication hardware within the floating-point cores. The total hardware size for this system, without an accelerator, was 4,669 Cyclone II logic cells

For this case study we have chosen to re-execute erroneous operations using a software floating-point library on the host microprocessor. We use the Iteration-Indexed Re-execution Buffer (IIRB) method to communicate erroneous results to the host, as described in Section 4.2.1. A re-execution buffer (RXB) is allocated by the accelerator’s C interface libraries on the host processor, its base address being sent to the accelerator with the input operands. When our optimisation hardware detects that an erroneous result will be generated, it raises an exception flag to indicate that the result is incorrect and should be disregarded. The accelerator’s write back stage then writes the current iteration number to the RXB. Erroneous results are not written to the results array, allowing input and results arrays to point to the same location if desired.

A shared memory architecture allows the accelerator direct access to the operand arrays and RXB allocated by the host processor. Once initiated,

the accelerator reads operands from the supplied arrays and writes a result back to the correct array location for each successful operation. The host processor flushes its cache before this begins to ensure the accelerator reads the most recent array values. The software libraries do not read from the result array before the accelerator has completed, ensuring that no stale values enter the cache in our single-threaded system.

Configuration of the accelerator is via a slave port connected to the Avalon switch fabric, which the host processor can access using memory-mapped I/O functions. The slave port register layout is shown in Figure 6.10, with the registers as follows:

- **UAddress** - The base address of the array holding values of the operand **U**. Write-only.
- **UNewAddress** - The base address of the array holding values of the operand **UNewAddress**. Write-only.
- **UOldAddress** - The base address of the array holding values of the operand **UOldAddress**. Write-only.
- **Count** - The number of operands to be used (the size of the operand array). During execution this stores the number of operands remaining. Read/write.
- **Start** - Set to '1' to commence execution, reset to '0' when the accelerator is finished. Read/write.
- **ResultAddress** - The base address of the array in which to store the calculated results. Write-only.
- **RexAddress** - The base address of the re-execution buffer. Write-only.
- **RexCount** - The current number of entries in the re-execution buffer. Read-only.

A set of C library functions perform configuration of the accelerator, setting the slave port registers to the correct settings. After initiating execution of the accelerator by setting the **Start** register to '1', the host processor begins polling the status of the slave port registers, reading from the **Start**

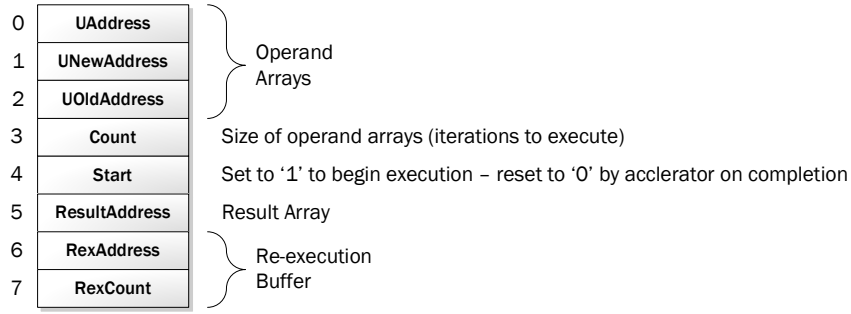


**Figure 6.10:** Register layout for the *102.swim* accelerator slave port

and **RexCount** registers to determine if the accelerator has finished and if any operations need re-executing.

Our single-threaded embedded system polls the current status of the accelerator, however a multi-threaded system would be able to use interrupts to avoid occupying processor time unnecessarily. The Avalon switch fabric provides a dedicated connection between processor and accelerator slave port, so polling the port causes no contention issues with the memory accesses required by the accelerator.

Re-execution in our system is performed using the Nios II floating-point software libraries, and is overlapped with the accelerator execution itself. Erroneous operations are read from the re-execution buffer while the accelerator is running, allowing the processor to re-execute early operations while the accelerator is executing later ones. Overlapping re-execution in this way reduces the overall cost of re-execution.

We have also collected data on re-execution speed for the Nios II hardware floating-point unit (FPU), and present estimated results for hardware FPU re-execution in the next section.

## 6.3 Results

We have collected results for a variety of optimised configurations. In the following tables, graphs and analysis each configuration is identified by a name of the form: **nn-OPTLIST**, where **nn** is a number representing the base hardware configuration and **OPTLIST** is a comma-separated list of optimisations.

The base accelerator configurations are as follows:

- A full double-precision unit, with 52-bit significand and 11-bit exponent, denoted by **52**;
- A custom unit with 35-bit significand and 8-bit exponent, denoted by **35**;
- A single-precision unit, with 23-bit significand and 8-bit exponent, denoted by **23**.

For ease of reference, optimised units derived from these base configurations are identified according to the key in Figure 6.11.

The list of optimisations uses the abbreviations described in Chapter 3 (OAR, ER, NR), with a suffix of **P** indicating that each operation was customised according to profiling data, and a numerical suffix indicating the level of optimisation. We include only one non-profiled configuration, **52-OAR1**, as an example of how our technique benefits from the profiling step, or rather suffers without it.

Unless otherwise specified our results are given relative to the measures shown in Table 6.3, which represent the results from a software-only configuration. No hardware resources are devoted to an accelerator in this configuration, which is shown as **Software Only** in the results.

We have two distinct sets of results to present: empirical results from executing *102.swim* on our system as described, and derived results for alternative error-recovery methods and systems with multiple data-paths.

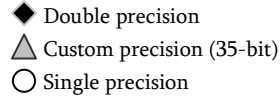


**Figure 6.11:** Key for *102.swim* results

### 6.3.1 Single Data-path

Our first set of results is for the system as described, with a single data-path processing loop iterations. This is the limit of our hardware platform, as multiple data-paths leave the computation waiting for memory accesses. In this scenario, generating an accelerator with a reduced hardware size is the desired goal, rather than increased parallelism from placing an additional data-path in the free space. This configuration also allows us to look at the effect of each optimisation in isolation.

Figure 6.12 shows a plot of hardware size increase against measured execution time, relative to a software-only approach. A section of the results is expanded to provide easier comparison of our optimisation options. Table 6.5 shows the total hardware size for each option, in Altera Cyclone II logic cells.

The accelerated versions are much faster; with the exception of the non-profiled **52-0AR1** configuration, they all reduce the execution time below 3% of the software-only version. As expected, the base configurations (denoted by **nn-Base**) execute the fastest, as they do not re-execute any operations. Table 6.4 shows observed re-execution rates for various optimisation options, which are independent of the base configuration.

Of key interest is how much size we have added to the design by introducing the accelerator. The base cases add the most, with the standard double-precision accelerator, **52-Base**, producing a 118% increase. At the other end of the range, **23-0ARP,NRP,ER6** increases the size by 57%. The trade-off between size and speed also becomes apparent. The smallest double-precision configuration is **52-0ARP,NRP,ER6** with a 73% increase, but this takes 26% longer to execute than the alternative **52-0ARP,NRP** which comes in at 78%. Table 6.5 shows the size of each configuration in Altera Cyclone II logic cells.

| Base Metric                 | Value      |
|-----------------------------|------------|
| Hardware Size (Logic Cells) | 4,669      |
| Execution Time (ms)         | 3,890,000  |
| Total Iterations            | 44,830,854 |

**Table 6.3:** Base metrics for software-only execution, Altera EP2C35F672C8 at 70MHz. Relative comparisons are made to these results.

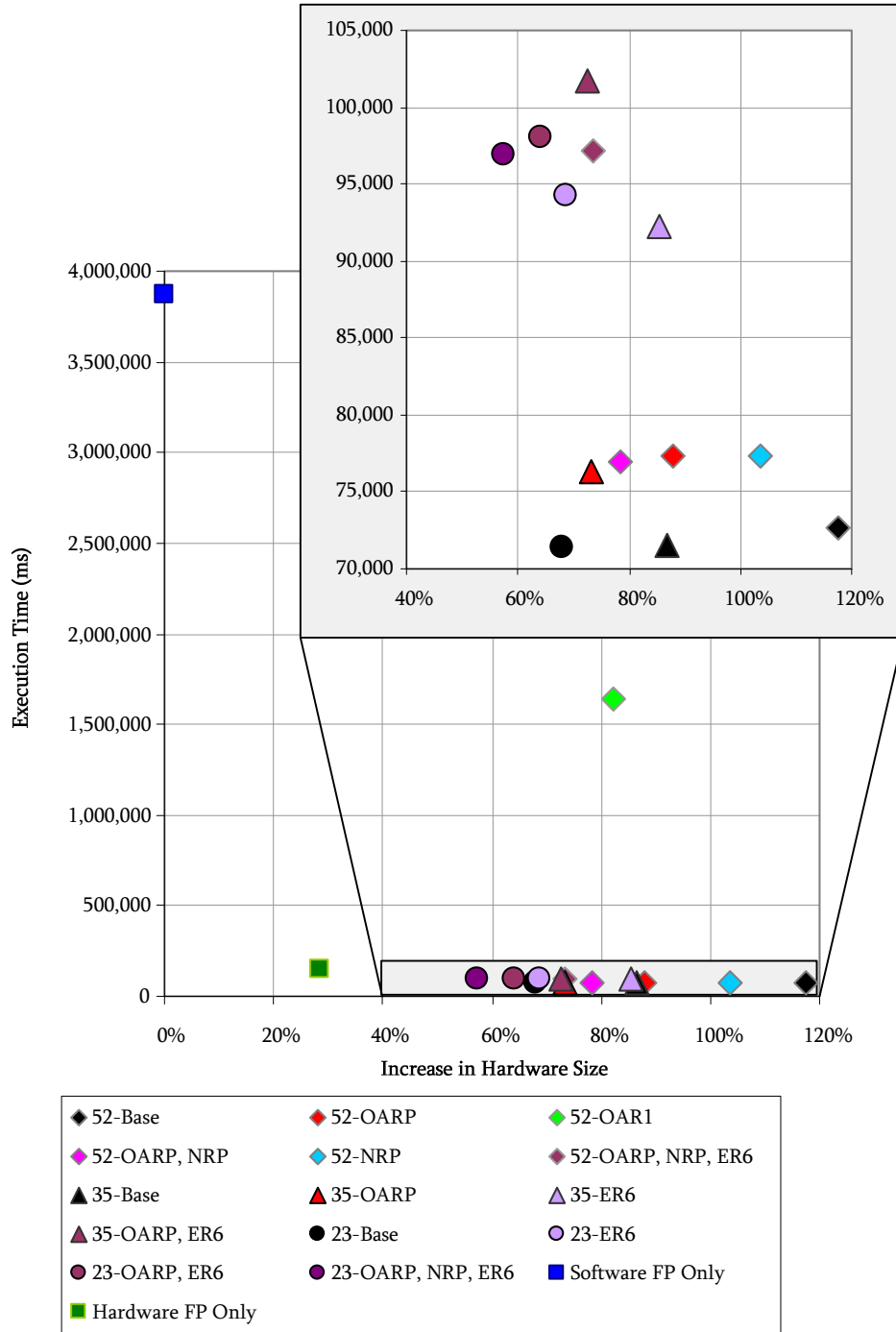


**Figure 6.12:** Optimisation results for CALC3, using a single data-path on an Altera EP2C35F672C8 at 70MHz. This approach seeks to reduce the size of the hardware design, at the expense of increased execution time.

| Optimisation        | RX Rate     | Notes                            |
|---------------------|-------------|----------------------------------|
| <b>OAR6</b>         | 0.0%        | <i>Double-precision default</i>  |
| <b>OAR5</b>         | 0.0%        | <i>Single-precision default</i>  |
| <b>OAR4</b>         | 38.0%       |                                  |
| <b>OAR1,2,3</b>     | 39.1%       |                                  |
| <b>OARP</b>         | <b>0.4%</b> | <i>Per-unit optimisation</i>     |
| <b>ER11</b>         | 0.0%        | <i>Double-precision default</i>  |
| <b>ER8</b>          | 0.0%        | <i>Single-precision default</i>  |
| <b>ER6</b>          | 0.7%        |                                  |
| <b>OARP,ER6</b>     | 1.1%        |                                  |
| <b>NR6</b>          | 0.0%        | <i>Double -precision default</i> |
| <b>NR5</b>          | 0.0%        | <i>Single-precision default</i>  |
| <b>NRP</b>          | 0.03%       |                                  |
| <b>OARP,NRP,ER6</b> | 1.1%        |                                  |

**Table 6.4:** Observed re-execution (RX) rates for optimisation combinations

| Configuration                    | Cyclone II Logic Cells |
|----------------------------------|------------------------|
| Software FP Only                 | 4,669                  |
| Hardware FP Only (Processor FPU) | 6,007                  |
| 52-Base                          | 10,161                 |
| 52-OARP                          | 8,656                  |
| 52-OAR1                          | 8,458                  |
| 52-OARP, NRP                     | 8,226                  |
| 52-NRP                           | 9,336                  |
| 52-OARP, NRP, ER6                | 7,957                  |
| 35-Base                          | 8,635                  |
| 35-OARP                          | 7,950                  |
| 35-ER6                           | 8,535                  |
| 35-OARP, ER6                     | 7,860                  |
| 23-Base                          | 7,742                  |
| 23-ER6                           | 7,692                  |
| 23-OARP, ER6                     | 7,541                  |
| 23-OARP, NRP, ER6                | 7,248                  |

**Table 6.5:** Cyclone II logic cell usage for base and optimised configurations, on an Altera EP2C35F672C8 at 70MHz.

We now explore the effect of different optimised configurations, *relative to the double-precision base-case, 52-Base*. Normalisation Reduction performs well in this context providing a 15.6% decrease in hardware size, with a negligible increase in execution time due to re-executions.

Configuration 52-OAR1 shows the effect of naively using OAR-1 on all data-path elements. Although it results in a 35.3% decrease in data-path size from the double-precision base case, it produces a re-execution rate of 39.1%, leading it to be over 2,000% slower than 52-Base. In contrast, applying targetted optimisations for 52-OARP gives a hardware reduction of 30.3% but with a re-execution rate of just 0.4%. From these results it is clear that profile-directed optimisation of individual data-path elements has a distinct advantage over a blanket optimisation policy, reducing the re-execution rate by two orders of magnitude with a relatively small increase in hardware.

The advantages to be gained from our optimisations when using double-precision hardware are significant, with a 44.8% reduction in hardware size over 52-Base for a manageable increase in relative execution time (from 1.9% of the software only approach to 2.5%).

Configurations 23-Base and 23-ER6 demonstrate the negligible effect of Exponent Reduction when using a small significand. A two-bit reduction in exponent produces less than a 2% reduction in hardware size, but with a 0.7% increase in error rate and corresponding jump in execution time. Contrasting 52-OARP,NRP with 52-OARP,NRP,ER6 shows a more significant 5% hardware reduction with five fewer bits in the exponent.

We can also see that our optimisations offer a greater benefit for double-precision units than single-precision. Comparing 52-Base and 23-Base with 52-OARP,NRP,ER6 and 52-OARP,NRP,ER6 shows that the optimisations close the gap in hardware size between the two types (from 49.7% to 15.4%).

### 6.3.2 Accelerator/Processor Modelling

The previous empirical results used a serial-execution system, where data are sent to the accelerator by a processor which then waits for results or re-executions. We now derive a model for overlapped execution, where the processor reserves data for itself. Both accelerator and processor work on a dataset, yielding a faster execution time overall. We know the average

execution time per operation (including overheads) for both processor ( $t_p$ ) and accelerator ( $t_a$ ). Thus we can calculate the total time ( $T_p$ ) the processor spends on its allocation of work ( $n_p$ ):

$$T_p = n_p t_p \quad (6.2)$$

And also for the accelerator:

$$T_a = n_a t_a \quad (6.3)$$

We desire both to complete their workload in the same time, so neither sits idle. Consequently we get:

$$n_p t_p = n_a t_a \quad (6.4)$$

And between them the processor and accelerator must execute the total number of operations required,  $N$ :

$$N = n_a + n_p \quad (6.5)$$

From these equations we can derive  $n_a$  and  $n_p$  as follows:

$$n_a = N \frac{t_p}{t_a + t_p} \quad (6.6)$$

$$n_p = N \frac{t_a}{t_a + t_p} \quad (6.7)$$

Which it is clear provides the desired effect of allocating work in inverse proportion to the execution time for a single operation in each system.

However, this solution is unsuitable for our optimisations, as we do not have a conventional heterogeneous system where both processor and accelerator generate correct results. The accelerator generates errors at a rate  $\rho$ , determined by the distribution of the operands, which must be re-executed by the processor. Hence while the execution time for the accelerator remains as 6.3, the execution time for the processor must include operations it is re-executing:

$$T_p = n_p t_p + n_a \rho t_p \quad (6.8)$$

We still desire neither execution option to remain idle, so must derive  $n_a$  and  $n_p$  from  $T_p = T_a$ :

$$n_a t_a = n_p t_p + n_a \rho t_p \quad (6.9)$$

Substituting  $n_p$  for a re-arrangement of 6.5 and solving for  $n_a$  gives:

$$n_a = \frac{N}{t_a/t_p + 1 - \rho} \quad (6.10)$$

The processor is allocated the remaining  $N - n_a$  operations. In our idealised system the total execution time for processor and accelerator combined is thus:

$$T = \frac{N t_a}{t_a/t_p + 1 - \rho} \quad (6.11)$$

We use this as our model in subsequent sections. Where the error rate and per-operation execution speed difference is large, the model will allocate over 100% of the work to the accelerator. This indicates that the accelerator will become idle, as the processor clears a back-log of re-executed operations. Our implementation handles this degenerate case by limiting the total assignment of work to the accelerator to 100%.

Figure 6.13 illustrates the behaviour of the model. The execution time changes as the error rate and re-execution speed vary. 100% relative execution time represents one of our floating-point data-paths operating with no errors. Consequently, when the re-execution speed is equal to the accelerator speed, overlapped execution leads to a 50% reduction in execution time as the workload is spread equally between the accelerator and processor. Lines above 100% execution time show that the system is saturated, so further increases in error rate lead to non-overlapped executions: the accelerator sits idle while the processor finishes re-execution of operations. Consequently the model produces a straight line increasing linearly with error rate.

From this we can see the maximum error rate we can tolerate for a given ratio of execution to re-execution speed, assuming the re-execution method can be used for normal calculations when idle.

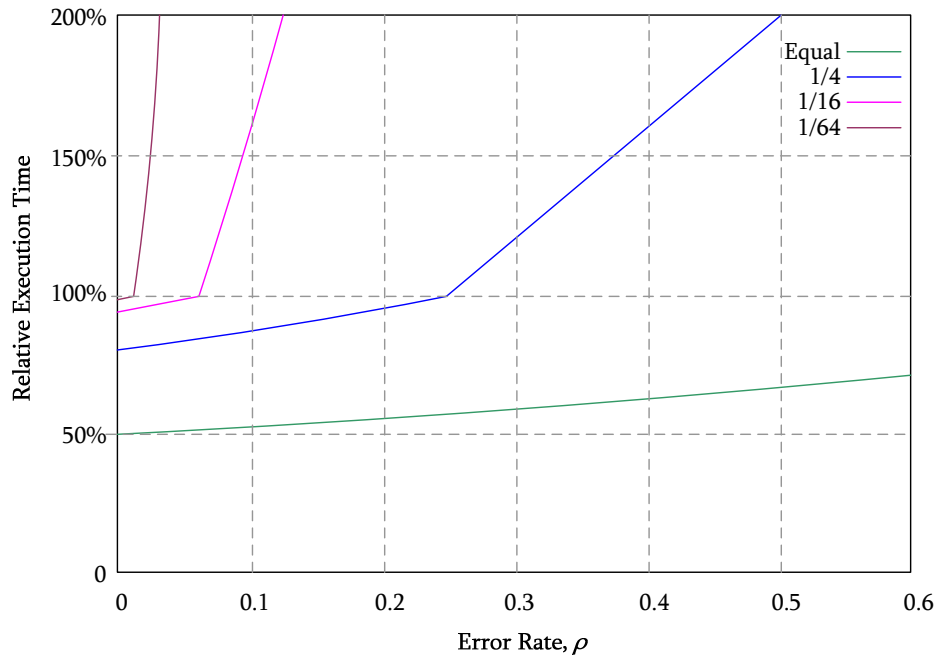


**Figure 6.13:** Behaviour of overlapped execution model with varying error rate and re-execution speed. The relative execution time is based on an accelerator executing by itself with no errors. Each line represents a different ratio of accelerator speed to re-execution/processor speed, from equal speeds to the re-execution occurring at  $\frac{1}{64}$  of the speed.

### 6.3.3 Overlapped Execution

Using the model in Equation 6.11, we now present estimated results for overlapped execution, where the processor and accelerator are working on the dataset at the same time. We include two sets of results - one for re-execution in software on the processor, and one for re-execution on a hardware floating-point unit.

#### Software Re-execution

Figure 6.14 shows the effect of overlapped execution on our single data-path system, with re-execution in software.

We have expanded the same section of the graph as in Figure 6.12. The variation in execution time between the optimised versions is greatly reduced, with the most heavily-optimised versions such as `52-OARP,NRP,ER6` suffering a penalty of under two seconds, rather than twenty-five. The allocation of some operations directly to the processor reduces the total number of re-executions required. However, the substantial re-execution penalty for `52-OAR1` is not overcome by overlapped execution.

Comparing `52-Base` in Figures 6.12 and 6.14 reveals that overlapped execution does not yield a large decrease in overall execution time, due to the size of the speed differential between accelerator and processor.

#### Hardware Re-execution

In Figure 6.15 we present estimated results for our system with re-execution on a hardware floating-point unit within the processor. In this example our basis for comparison is the system with a hardware floating-point unit present, so the relative increase in hardware size introduced by the accelerator is lower.

In this scenario, where the accelerator and processor are closer in performance, we can see that the execution time improvement is considerable, with around a 35% reduction over the equivalent software-based re-execution system. A double-precision accelerator, `52-OARP,NRP,ERP`, can also be included for a 35% size penalty over the non-accelerated hardware FPU system.

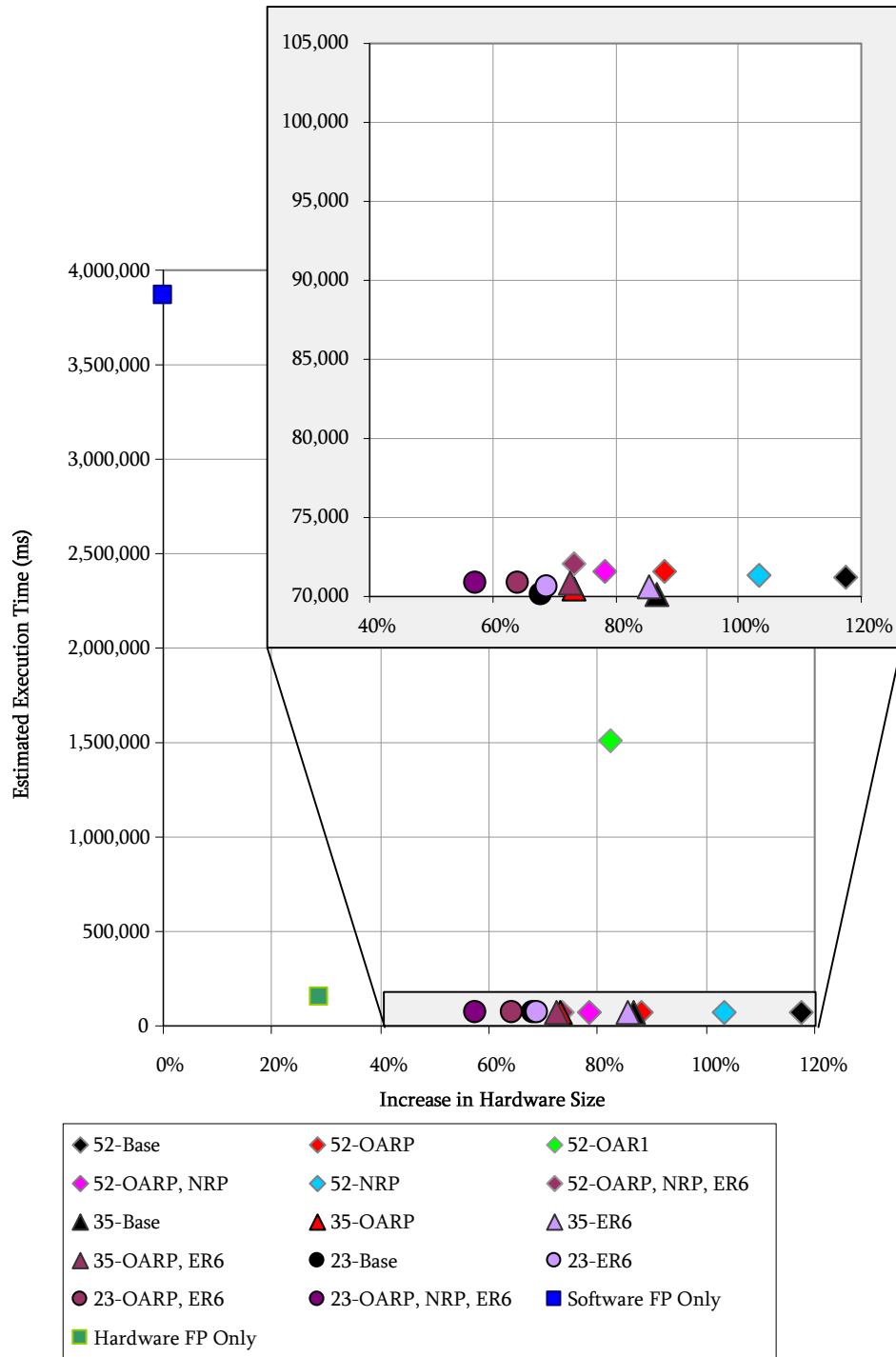


**Figure 6.14:** Modelling results for CALC3, using a single data-path and overlapped execution. Re-execution of floating-point operations takes place in software libraries on the Nios II processor. Model source data based on an Altera EP2C35F672C8 device at 70MHz.

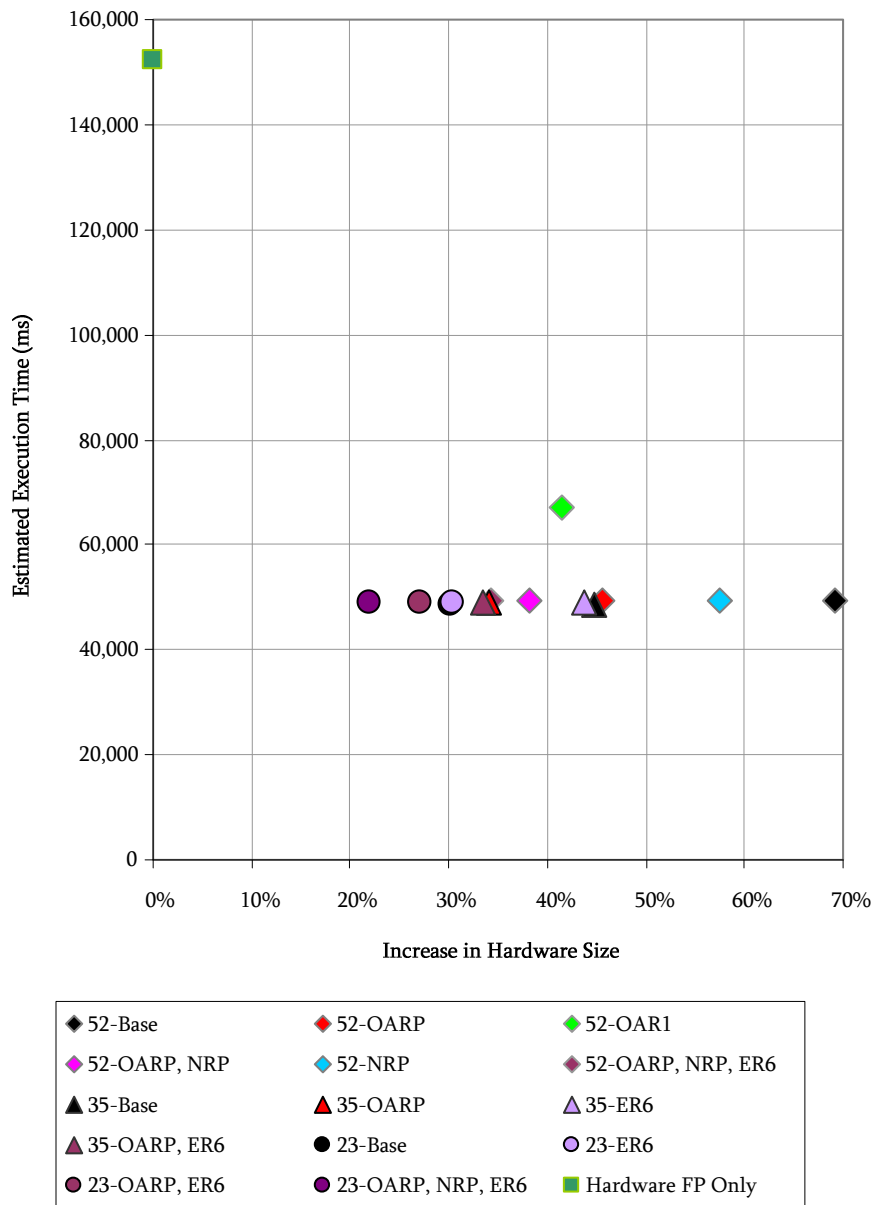


**Figure 6.15:** Modelling results for CALC3, using a single data-path and overlapped execution. Re-execution of floating-point operations takes place using a hardware floating-point unit on the Nios II processor. Model source data based on an Altera EP2C35F672C8 device at 70MHz.

### 6.3.4 Parallel Data-paths

So far our results have examined the application of our technique to reducing the size of a hardware accelerator. We now present a set of results which considers the use of multiple parallel data-paths, filling the space saved by optimisation. In this scenario a reduction in execution time is more important than a reduction in hardware size.

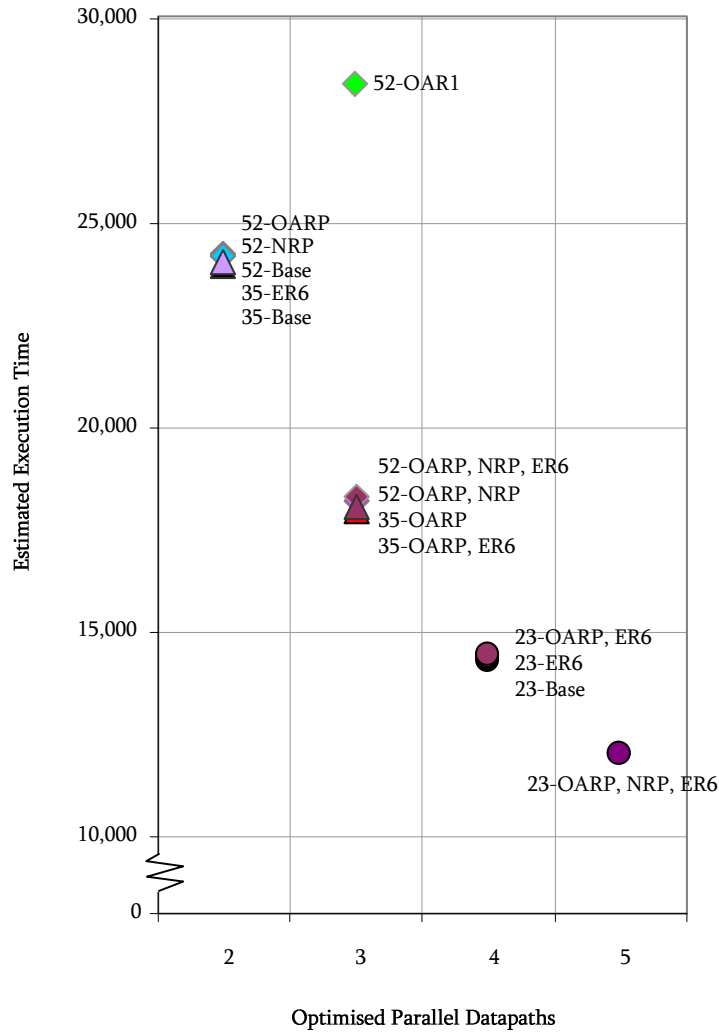
Our presentation of results is slightly different in this case, as we show the number of parallel data-paths against estimated execution time. Our base case has one fully-capable double-precision floating-point data-path (i.e. one instance of **52-Base**), with two further data-paths subjected to our optimisation technique. Hardware savings in these two optimised data-paths are used to introduce additional data-paths where possible. Figure 6.16 shows these results. Due to the tight clustering we have included labels on the data points.

We can see that five optimisations have no benefit in this scenario, as they do not reduce hardware size sufficiently to add an additional parallel data-path, but marginally increase execution time. The non-profiled optimisation, **52-OAR1**, allows an additional data-path to be added, but this does not compensate for the extra cost of re-execution.

The configurations **52-OARP,NRP** and **52-OARP,NRP,ER6** save enough resources to add an additional data-path, providing a consequential reduction in execution time. Reducing to single-precision mode allows four data-paths to be placed in the same hardware footprint, while the optimised single-precision configuration **23-OARP,NRP,ER6** allows five parallel data-paths, bringing the estimated execution time to under half the double-precision base case.

### 6.3.5 Non-profiled Data

Our final set of results demonstrate how the error rate varies with array size. As described at the start of this chapter, we used an array size of  $128 \times 128$  for profiling runs. Figure 6.17 shows the error rate of four non-profiled variations, compared to the original  $128 \times 128$  version. We can see that the error rates remain low, with our profiled version actually having the second-highest rate.



|                |                     |                     |
|----------------|---------------------|---------------------|
| ◆ 52-Base      | ◆ 52-OARP           | ◆ 52-OAR1           |
| ◆ 52-OARP, NRP | ◆ 52-NRP            | ◆ 52-OARP, NRP, ER6 |
| ▲ 35-Base      | ▲ 35-OARP           | ▲ 35-ER6            |
| ▲ 35-OARP, ER6 | ● 23-Base           | ● 23-ER6            |
| ● 23-OARP, ER6 | ● 23-OARP, NRP, ER6 |                     |

**Figure 6.16:** Modelling results for CALC3, using a single unoptimised double-precision data-path and two further optimisable double-precision data-paths. Re-execution of floating-point operations takes place on the unoptimised data-path. Model source data based on an Altera EP2C35F672C8 device at 70MHz.

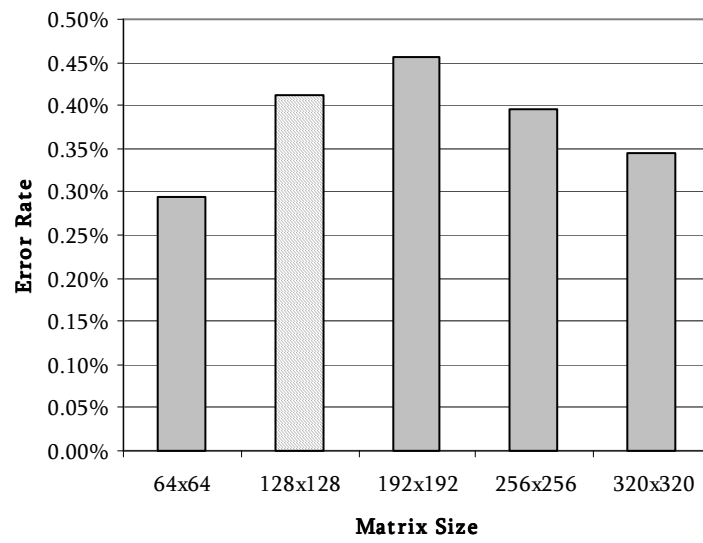


**Figure 6.17:** *102.swim* Array Size Variation - Observed re-execution rate for non-profiled array sizes using configuration 52-OARP,NRP.

## 6.4 Summary

This case study has shown the varying success of our optimisations when applied to the SPEC FP95 benchmark *102.swim*.

The single data-path approach is a simple trade-off between hardware savings and increased execution time. From our results we can summarise the following about our optimisations, as applied to this scenario:

- Normalisation Reduction with profiling performs well, achieving a good hardware saving with negligible error rate;
- Operand Alignment Reduction with profiling achieves even greater hardware reductions, but with increased re-execution cost;
- Exponent Reduction achieves minimal hardware gains, with a high performance penalty;
- Our optimisations offer a greater benefit for double-precision units;
- Operation-level profiling plays a vital role in selecting optimisations.

We have also shown that operating optimised units with a non-optimised fall-back option, such as a processor, allows the overall error-rate and hence re-execution penalty to be reduced.

In a parallel data-path approach the aim is to reduce execution time, rather than simply save hardware resources. In this scenario some optimisations become non-viable, because they do not generate a sufficient saving to increase the number of parallel data-paths. With a high error rate, as with the non-profiled 52-OAR1, the time required for re-execution outweighs the time saved by introducing an extra data-path.

## 7 Conclusion

We now conclude our work with a brief summary, by reviewing our contributions and looking into the future directions we could take.

We have presented a profile-driven method for reducing the size of floating-point hardware, whilst adhering to the IEEE-754 standard. Our optimisations generate calculation errors that are both easily detectable and correctable by re-execution, in software or hardware. Our technique speculates that the re-execution rate is low enough to allow re-execution of erroneous results without a high overall penalty. Our floating-point value profiler allows us to identify applications and code segments where this is so, then select appropriate reduced configurations for each operation in the accelerator data-path.

We have described our floating-point value profiler and presented a small number of profiling results from the MORPHY and FFMPEG programs, with extended results for the SPEC 95 benchmark *102.swim*.

In a case study we have demonstrated error rates as low as 1%, with reductions in accelerator hardware size of up to 45%. Furthermore, we have illustrated that targeted profile-driven optimisation can lead to a dramatic improvement in error rate while retaining hardware size reductions.

Our optimisations take effect at a basic hardware level, but require high-level support from both hardware and software to undertake the re-execution of erroneous results. We have presented a number of options to this end, but envisage many more being available. How the ideal solution appears is dependent on the target hardware architecture and application.

## 7.1 Review of Contributions

In Section 1.1, we claimed a number of contributions. We now review those contributions in light of the work presented so far.

- **A speculative optimisation technique offering a reduction in the hardware resources used by a hardware floating-point accelerator, showing up to a 45% saving in our benchmark.**

Our optimisation technique is centred around the low-level modification of IEEE-754 floating-point units, to remove or reduce operations that are expensive in reconfigurable logic. We presented details of these core optimisations in Chapter 3. Our three main optimisations are: Operand Alignment Reduction (OAR, described in Section 3.2), Normalisation Reduction (NR, described in Section 3.3) and Exponent Reduction (ER, described in Section 3.4). In double-precision these optimisations can achieve up to 17%, 37% and 27% reductions, respectively. In Chapter 6 our case study demonstrated that, for the optimal combination of optimisations, we could achieve a 44% hardware saving in double-precision.

- **A selection of methods for integrating our optimised floating-point hardware with a host processor, including error detection and correction mechanisms.**

We divided integration methods into two distinct parts. Firstly, in Section 4.1 we considered two possible configurations for turning our individual floating-point units into useful accelerators. We concluded that operation-level optimisation of floating-point units is the best way to achieve good results with our technique, so opted for a streaming data-path structure. We also reviewed the hardware/software co-design problem, showing that there are many solutions which could make use of our optimised units.

Secondly, in Section 4.2, we examined what happens when our floating-point units generate erroneous results. We presented some solutions to the problem, but do not rule out a wide variety of other options, depending on the hardware/software co-design solution and application context.

- ***Float Watch*, a tool for profiling the characteristics of floating-point applications.**

In Chapter 5 we presented our Valgrind-based profiler, *Float Watch*. We described the types of data it collects and explained how this relates to our low-level optimisations. Although designed primarily to assist with our optimisation process, we also showed a number of alternative uses, where the profiler is able to uncover poor floating-point behaviour.

- **Floating-point profiles for a variety of floating-point applications and interpretation of associated profile data.**

In addition to a description of our profiler, in Section 5.3 we presented some results we had gathered during the development of *Float Watch*, along with a commentary on their meaning. We explored the open-source FFMPEG audio and video conversion tool, benchmarks from the SpecFP95 suite and the MORPHY electron density analysis tool.

- **A specific application of our technique to accelerators for the *102.swim* benchmark from SpecFP95.**

The final contribution draws our work together. In Chapter 6, the *102.swim* benchmark from SpecFP95 was taken from pure FORTRAN running on a CPU to an FPGA-accelerated solution. We performed several analyses of the code, from a simple execution-time profile to an operation-level floating-point profile analysis. A region of code suitable for optimisation was selected and an accelerator generated, using the Iteration Indexed Re-execution Buffer method, described in Section 4.2.1.

We demonstrated our technique with optimisations in a variety of configurations. These included a naive implementation which achieved a 35% decrease in double-precision hardware size, but with a 40% re-execution rate. More advanced optimisations, using operation-level profiling data, allowed a hardware reduction of 40% with a 0.4% error rate, or a reduction of 45% with a 1.1% error rate.

We will revisit how far these contributions met our original aims in our concluding remarks.

## 7.2 Future Enhancements

Our work has required the development of a number of different components, including optimisable floating-point units, our *FloatWatch* floating-point profiler, accelerator architectures supporting error-recovery and specific hardware implementations for our case-study. Each of these components has been developed to the level required for our work to date, however they could all be developed further.

In this section we will explore the future directions this work could take.

### 7.2.1 Additional Optimisations

We begin with avenues for additional optimisations. The three types presented in Chapter 3 were devised based on the proportion of hardware used by their respective components on the FPGA architectures we were interested in. However, these optimisations are of less utility on ASIC platforms, where shifters attract a lower cost. Devising optimisations suitable for an ASIC, if possible, along with changes to the profiler, would be a constructive avenue of development. The same applies for other floating-point unit designs, including those from the FPGA vendors, as well as the trigonometric and logarithmic functions.

We had considered a further optimisation for the FPGA architecture, which when combined with software modifications could - in some circumstances - yield a large hardware reduction for limited error rate.

The optimisation we propose, Swap Elimination, is related to Operand Alignment Reduction, forming part of the hardware required to prepare normalised floating-point operands for a fixed-point add or subtract. Our proposal is to eliminate the hardware required to swap operands where they are presented the ‘wrong way round’, i.e. with the smallest operand where the hardware expects the largest to be.

This optimisation allows two different ‘handed’ designs to be generated, as shown in Figure 7.1. Operations which require operands to be swapped (or not, in the case of the pre-swapped unit), would generate an error and lead to re-execution, as is the case with other errors. Section 5.3 showed examples of where the profiling data indicated a good chance of success with this optimisation; operands were presented with the greater exponent mostly to one side.

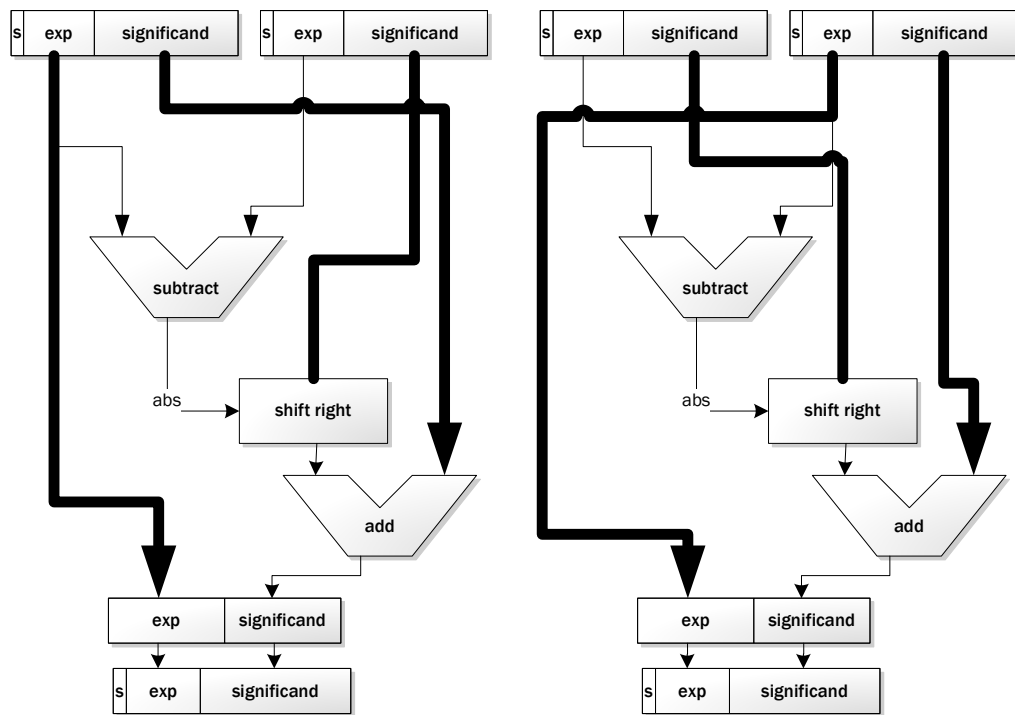


**Figure 7.1:** A 'Handed' Floating-Point Adder

This optimisation also raises another interesting avenue of exploration: floating-point units with different optimisations, to which operands are routed appropriately. For example, a design could contain equal numbers of left- and right-handed units, with a pre-routing comparison sending each operation to the correct type. This relies on operations being equally balanced between left- and right-handed versions to avoid units sitting idle. Having one unit sitting idle may be preferable if power is a consideration, as the smaller ‘handed’ units would use less power while running, and could be turned off when not required.

This idea could also apply to on-chip correction (Section 4.2), where out-of-range operations would be routed to the fully-functional unit without first being executed on the optimised units.

Finally, although floating-point optimisations have been the focus of this work, we need not focus on that area in the future; run-time profiling and speculative execution could be applied to hardware optimisation for other numerical representations.

### 7.2.2 Power Optimisation

Our main results in Chapter 6 focussed on the hardware-reduction aspect of our optimisations, however there is another factor: power.

Reducing power requirements is an important consideration, both in energy-limited embedded systems and in high-performance computing, where the cost of cooling systems has led to a drive for more efficient designs[NR 02, SHF06]. The architecture of FPGAs has led to a number of interesting opportunities for power optimisation, including inverting look-up-table inputs[AN06] and power-constrained synthesis of hardware from high-level descriptions[SJ02].

Gayasen et al[GTV<sup>+</sup>04] proposed a system where the FPGA would be divided into regions, each of which could be placed into a ‘sleep’ mode where the clock was disabled, reducing dynamic power. This could be applied to a variation of our on-chip re-execution technique, whereby a fully-functional floating-point unit is contained in a ‘sleeping’ region, to be activated only as needed for re-execution.

Clearly re-execution brings with it energy requirements, so the aim must be to minimise the total energy used over the course of an application,

saving energy by using optimised units only to expend it if re-execution is required.

Exploring dedicated power-optimisations which could make use of profiling information is another interesting area for future research.

### 7.2.3 Algorithmic Modifications

We must also consider the effect that software modifications could have on our optimised data-paths. In particular, we consider techniques used to improve the accuracy and error behaviour of floating-point calculations. Higham[Hig93] and Wilkinson[Wil94] have collected together several methods, however we will focus on the process of floating-point summation.

As previously described, as the  $\Delta e$  of operands to be added or subtracted increases, so too does the loss of accuracy in the representation. When  $\Delta e$  approaches the size of the significand,  $\text{bits}(s)$ , almost all significant bits are lost from the smaller operand. Higham performed error analyses on some methods to remove this behaviour[Hig93], the idea being to add only similarly-sized operands as far as possible, reducing  $\Delta e$ . These techniques should also improve the performance of Operand Alignment Reduction, which relies on a small  $\Delta e$  to reduce hardware size.

Solutions which rely on absolute ordering of values would also benefit our proposed Swap Elimination optimisation; operands are pre-ordered to have the largest first (or last, depending on ordering). In either case Swap Elimination could provide a substantial reduction in hardware size for a 0% error rate.

### 7.2.4 Dynamic Reconfiguration

Our target architectures have been FPGAs of various flavours, however we have so far proposed only static configuration of the devices. In this mode an application programs the device as an accelerator at the start, but does not reconfigure while it is in use.

With dynamically-reconfigurable fabrics at our disposal, we are able to change the representation for different phases of an application if we are able to identify both the phases and appropriate representations, as demonstrated by Styles[SL05].

CORDS[DJ98] targetted embedded processors, dividing software elements

into tasks which could be placed on the device separately, as required. This straightforward approach would be equally applicable to our work: in particular, our case study in Chapter 6 could make use of a different accelerator for each main sub-routine in the program.

ReCoBus Builder[KBT08] provides a more sophisticated approach which could make more extensive use of our method. It defines a set of dynamically reconfigurable, isolated blocks which can be linked together and communicate. They are assembled at run-time as required.

In our system, we envisage a dynamically reconfigurable set of floating-point units or data-paths, along the same lines. The system would be self-tuning, starting with a moderately optimised set of floating-point units, providing as much parallelism as possible within the confines of the device. During execution the error-correction logic (be it in hardware or software) would monitor the error rate. Should the rate be higher than acceptable, the data-paths would be reconfigured with a set of less-optimised units, with the aim of bringing the error rate down at the expense of decreased parallelism. If the error rate were to fall below a threshold, the reverse could occur with data-paths adopting more heavily optimised units, increasing parallelism. There is no requirement that all data-paths in a system should have the same level of optimisation applied.

The potential benefits of this approach can be demonstrated with a simple example, using the Newton-Raphson method[Rap90]. The same applies to any algorithm which iterates smoothly to convergence.

In this example the aim is to find a solution to  $a \cos(x) - bx^3 + c = 0$ , where  $a$ ,  $b$  and  $c$  are user-defined parameters. The Newton-Raphson method allows this to be re-written as an iterative refinement, using the following relationship:

$$x_{n+1} = x_n - \frac{f(x)}{f'(x)}$$

Using this method  $x$  will converge towards a solution, with the relationship for this example being:

$$x_{n+1} = x_n - \frac{a \cos(x) - bx^3 + c}{-a \sin(x) - 3bx^2}$$

The starting value,  $x_0$ , is chosen as an estimate of the expected final value if available. Figure 7.2 shows a graph of  $x$  on each iteration for starting values between 0.1 and 1.2, with  $a = 1$ ,  $b = 1$  and  $c = 0$ . The characteristics of this particular problem result in  $x$  increasing dramatically away from the actual value when the estimate is low, however it then decreases towards

the correct answer. The same pattern occurs with other values for the coefficients  $a$ ,  $b$  and  $c$ .

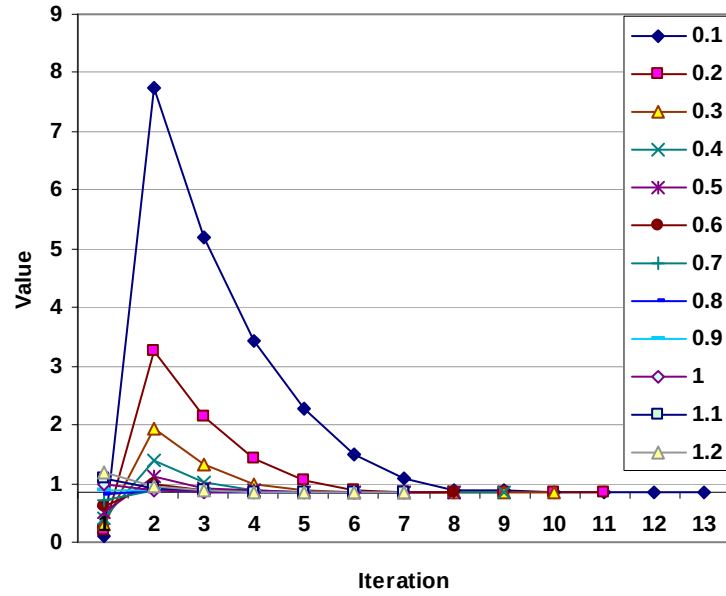


**Figure 7.2:** Graph of  $x$  at each iteration of the Newton-Raphson method; each line represents a different starting estimate for  $x$

With knowledge of this characteristic it is possible to dynamically reconfigure the floating-point data-path available, relying on the reducing range of intermediate and final values to use a more optimised data-path, increasing parallel processing ability. In this example it may mean placing a single trigonometric unit into hardware initially – performing the cosine and sine operations sequentially – before reconfiguring to two units executing in parallel to provide a decrease in execution time. While we have no specific optimisations for these units at present, Exponent Reduction is applicable.

With so few iterations in this example there would be no payoff, however for long-running applications there is a potential benefit from increased parallelism.

### 7.2.5 Modelling and Design Space Exploration

The trade-off between hardware size and re-execution time is central to our technique. The most appropriate solution depends on three factors: the

hardware savings made by optimisation, the cost of re-execution on the chosen architecture and the distribution of operand values. With a profiled selection of example data sets, these factors can all be approximated from known values at the operation level. Consequently it would be possible to produce a model for design-space exploration, to identify appropriate trade-offs without hardware generation and testing of all designs. In Section 6.3 we presented a system-level model, however a more detailed, operation-level model would be required to achieve the best results. Trade-offs could then be modelled on an operation-by-operation basis, combining to produce a good solution for the whole accelerator. So far optimisation levels have been selected using a manual analysis.

Given the effort required to generate multiple hardware options, place and route them, program them onto the hardware and then run test datasets, a model-driven design-space exploration would improve the total design time considerably. Such a solution could also use profile-derived optimisations (as shown in the case study in Chapter 6) to reduce the design space and save even more time. While hardware estimations would not be as accurate as a true hardware compilation step for a target device, a reasonable approximation can be derived from the results already collected.

### 7.2.6 *FloatWatch*

The *FloatWatch* profiler and supporting software has a great deal of scope for future improvements. There is a need for more automation in the system, particularly in the analysis of optimisation levels and errant floating-point behaviour. At present this step is conducted outside the *FloatWatch Explorer*, which is used to generate overview graphs and select interesting lines of code.

The profiler output files contain sufficient data to flag excessive shifts for alignment or normalisation, which indicate errant behaviour. One of the most useful features would be to supply the tool with a target error-rate and allow it to determine which optimisations to apply, based on the profiling result. Ideally this should be based on profiling runs with more than one dataset.

The profiler itself has one major limitation: speed. The Valgrind framework produces a dramatic slowdown which makes it unsuitable for profiling

very long-running applications. Any analysis of larger applications would benefit from a different approach to profiling, most likely compile-time generation of instrumentation code. The existing *FloatWatch Explorer* interface would still be suitable for viewing the results if the report format was retained. Other frameworks, such as Intel’s Pin[LCM<sup>+</sup>05], ATOM[ES95] or Dyninst[BH00] may provide a better balance of performance and flexibility. Another alternative is to use sampled profiling[AR01], where collection of all profiling data is replaced with data sampling to increase performance.

Support for floating-point instructions on 64-bit operating systems is limited, as x86-64 architectures use SIMD floating-point instructions, which the type system views a 128-bit vector rather than a floating-point number. In light of this and Valgrind’s previously mentioned performance limitations, an alternative profiling method is again an option to consider.

## 7.3 Limitations

We have explored an FPGA-based profile-driven technique, but it is one which has a number of limitations. We have already suggested some future work to overcome these limitations, but in this section we comment on them in more detail.

### 7.3.1 Application-Specific Integrated Circuits

Our choice of target technology provides the first, and probably the most serious, limitation. Our optimisations are focussed on an FPGA target platform, and do not transfer easily to ASICs. ASIC-targeted optimisations have been suggested as potential future work, but without those the speed and area advantage of an ASIC will always affect the viability of our technique to deliver competitive performance results on an FPGA. It also remains to be seen whether profile-driven ASIC-targeted optimisations are a viable solution. Kuon and Rose[KR07] compared FPGA and ASIC performance, concluding that a comparable FPGA design was 3.2 times slower and 40 times larger than on an ASIC. Although the FPGA area became more competitive when using FPGA macro blocks, the speed deficit remained. Thus our FPGA solution is far behind before we begin optimising.

In spite of this, our technique still has utility for applications where recon-

figurable devices are being used for reasons other than pure performance, for example where hardware must be updated whilst in the field. In these scenarios our technique could provide a reduction in the hardware area that would otherwise be used. Solutions using floating-point hardware blocks on FPGAs[HYL<sup>+</sup>09], proposed and developed while this work has been in progress, may stop our technique being viable even on these platforms.

### 7.3.2 Quality of Datasets

Our technique is dependent on the quality of the training data, so there is a problem if profiled datasets do not reflect real performance. In our examples we have used test datasets provided by the people using the applications in question. In the case of poor training datasets performance would be degraded as the percentage of re-executed operations would rise. The worst case is a re-execution of every operation, but the system would still generate correct results. The concept of dynamic reconfiguration was explained in Section 7.2. Although this could apply here, the system would remain sub-optimal when working with incorrect datasets but may be able to limit the damage this would cause to execution time.

## 7.4 Concluding Remarks

The aim of our research was to determine if an IEEE-754 floating-point unit could be reduced in size, based on prior knowledge of an application. Static analysis and user constraints have been used for many years to produce custom formats, but we were looking for a more aggressive approach.

We have succeeded in reducing the size of the floating-point units under investigation, by accepting errors will occur and correcting them. The *Float-Watch* profiler guides this decision, revealing where error rates should be low. Our results from Chapter 6 are very promising, with a large reduction in hardware size for a low error rate. The success of our method at reducing execution time through increased parallelism relies on a combination of low error rate and rapid re-execution, which we have shown is possible.

The elements of our work show potential, both as a whole and in their own right, and we look forward to seeing future developments.

# Bibliography

- [AAS<sup>+</sup>07] Sadaf R. Alam, Pratul K. Agarwal, Melissa C. Smith, Jeffrey S. Vetter, and David Caliga. Using FPGA devices to accelerate biomolecular simulations. *Computer*, 40(3):66–73, March 2007.
- [ABC<sup>+</sup>06] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael F.P. O’Boyle, John Thomson, Marc Toussaint, and Christopher K.I. Williams. Using machine learning to focus iterative optimization. In *CGO ’06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.
- [Adv09] Advanced Micro Devices, Inc. *ATI Stream Computing*. [www.amd.com](http://www.amd.com), 2009.
- [Agi08] Agility Design Systems. *Handel-C Language Reference Manual*. [www.agilityds.com](http://www.agilityds.com), 2008.
- [ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.
- [Alt08a] Altera Corporation. *Cyclone II Device Handbook*. [www.altera.com](http://www.altera.com), February 2008.
- [Alt08b] Altera Corporation. *Quartus II Handbook v8.1*. [www.altera.com](http://www.altera.com), November 2008.
- [Alt09a] Altera Corporation. *Avalon Interface Specifications v1.2*. [www.altera.com](http://www.altera.com), April 2009.

- [Alt09b] Altera Corporation. *Floating-Point Megafunctions v1.0*. [www.altera.com](http://www.altera.com), March 2009.
- [Alt09c] Altera Corporation. *Nios II Processor Reference Handbook v9.0*. [www.altera.com](http://www.altera.com), March 2009.
- [AN06] Jason H. Anderson and Farid N. Najm. Active leakage power optimization for FPGAs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(3):423–437, March 2006.
- [And98] Ray Andraka. A survey of CORDIC algorithms for FPGA based computers. In *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 191–200, New York, NY, USA, 1998. ACM.
- [AR01] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. *SIGPLAN Not.*, 36(5):168–179, 2001.
- [BCDL06] Nikolaos Bellas, Sek M. Chai, Malcolm Dwyer, and Dan Linzmeier. FPGA implementation of a license plate recognition SoC using automatically generated streaming accelerators. In *IPDPS 2006: 20th International Parallel and Distributed Processing Symposium, 2006.*, April 2006.
- [BCG<sup>+</sup>97] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-software co-design of embedded systems: the POLIS approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.
- [BDH<sup>+</sup>08] Kevin J. Barker, Kei Davis, Adolfo Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: the architecture and performance of Roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

- [BdS91] Frédéric. Boussinot and Robert de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, Sep 1991.
- [Bel] Fabrice Bellard. FFMPEG multimedia system.
- [BH00] Bryan Buck and Jeffrey K. Hollingsworth. An api for run-time code patching. *The International Journal of High Performance Computing Applications*, 14:317–329, 2000.
- [BHUH06] Michael J. Beauchamp, Scott Hauck, Keith D. Underwood, and K. Scott Hemmert. Embedded floating-point units in FPGAs. In *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 12–20, New York, NY, USA, 2006. ACM.
- [BHUH08] Michael J. Beauchamp, Scott Hauck, Keith D. Underwood, and K. Scott Hemmert. Architectural modifications to enhance the floating-point performance of fpgas. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(2):177–187, Feb. 2008.
- [BKL07] Ashley W. Brown, Paul H. J. Kelly, and Wayne Luk. Profiling floating point value ranges for reconfigurable implementation. In *informal proceedings of the Workshop on Reconfigurable Computing, HiPEAC 2007*, 2007.
- [BKL08] Ashley W. Brown, Paul H. J. Kelly, and Wayne Luk. Profile-directed speculative optimization of reconfigurable floating point data paths. In *informal proceedings of the Workshop on Reconfigurable Computing, HiPEAC 2008*, 2008.
- [BL02] Pavle Belanović and Miriam Leeser. A library of parameterized floating point modules and their use. In *12th International Conference on Field Programmable Logic and Application, FPL 2002*, pages 657–666, Montpellier, France, September 2002.

- [BLK06] Ashley W. Brown, Wayne Luk, and Paul H. J. Kelly. Generating hardware designs by source code transformation. In *position paper for STS'06, Software Transformation Systems Workshop at GPCE06/OOPSLA06*, 2006.
- [Bro05] Ashley W. Brown. Optimising transformations for hardware compilation. Master's thesis, Department of Computing, Imperial College London, 2005.
- [BS73] Fischer Black and Myron S Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–54, May-June 1973.
- [CFR<sup>+</sup>91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
- [CH02] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.
- [CLM<sup>+</sup>05] Ray C. C. Cheung, Dong-U Lee, Oskar Mencer, Wayne Luk, and Peter Y. K. Cheung. Automating custom-precision function evaluation for embedded processors. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 22–31, New York, NY, USA, 2005. ACM.
- [DCJNMW00] Roy Dz-Ching Ju, Kevin Nomura, Uma Mahadevan, and Le-Chun Wu. A unified compiler framework for control and data speculation. In *Parallel Architectures and Compilation Techniques, 2000. Proceedings. International Conference on*, pages 157–168, 2000.
- [DJ98] Robert P. Dick and Niraj K. Jha. Cords: Hardware-software co-synthesis of reconfigurable real-time distributed embedded systems. In *IEEE/ACM International Conference on Computer Aided Design*, pages 62–68, 1998.

- [DLJ97] Bharat P. Dave, Ganesh Lakshminarayana, and Niraj K. Jha. Cosyn: hardware-software co-synthesis of embedded systems. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 703–708, New York, NY, USA, 1997. ACM.
- [Don06] Jack Dongarra. The impact of multicore on math software and exploiting single precision computing to obtain double precision results. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, page 2, Washington, DC, USA, 2006. IEEE Computer Society.
- [DVKG05] Yong Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, pages 86–95, New York, NY, USA, 2005. ACM Press.
- [ECC04] Chun Te Ewe, Peter Y.K. Cheung, and George .A. Constantinides. Dual fixed-point: An efficient alternative to floating-point computation. In *Proceedings of International Conference on Field Programmable Logic 2004*, pages 200–208. Springer-Verlag, 2004.
- [ES95] Alan Eustace and Amitabh Srivastava. Atom: a flexible interface for building high performance program analysis tools. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings*, pages 25–25, Berkeley, CA, USA, 1995. USENIX Association.
- [Fab07] Gianni De Fabritiis. Performance of the Cell processor for biomolecular simulations. *Computer Physics Communications*, 176(11-12):660 – 664, 2007.
- [FAD<sup>+</sup>05] Brian K. Flachs, Shigehiro Asano, Sang H. Dhong, H. Peter Hofstee, Gilles Gervais, Roy Kim, Tien Le, Peichun Liu, Jens Leenstra, John S. Liberty, Brad W. Michael, Hwa-Joon Oh, Silvia M. Mueller, Osamu Takahashi, Akiyuki Hatakeyama,

- Yukio Watanabe, and Naoka Yano. A streaming processing unit for a Cell processor. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 134–135 Vol. 1, 2005.
- [FdLB<sup>+</sup>03] Henri Fallon, Alexis de Lattre, Johan Bilien, Anil Daoud, Mathieu Gautier, and Clment Stenac. *VLC User Guide*, 2003.
- [FGL01] Jan Frigo, Maya Gokhale, and Dominique Lavenier. Evaluation of the Streams-C C-to-FPGA compiler: an applications perspective. In *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 134–140, New York, NY, USA, 2001. ACM.
- [FOK02] Grigori Fursin, Michael F.P. O’Boyle, and Peter M.W. Knijnenburg. Evaluating iterative compilation. In *LCPC 2002: Proceedings of the Workshop on Languages and Compilers for Parallel Computing, 2002*, pages 305–315, July 2002.
- [GDM93] Rajesh K. Gupta and Giovanni De Micheli. Hardware-software cosynthesis for digital systems. *Design & Test of Computers, IEEE*, 10(3):29–41, Sep 1993.
- [Ger03] Arpad Gereoffy. *MPlayer - the movie player for Linux*, 2003.
- [GFA<sup>+</sup>04] Maya Gokhale, Janette Frigo, Christine Ahrens, Justin L. Tripp, and Ronald Minnich. Monte carlo radiative heat transfer simulation on a reconfigurable computer. In Jürgen Becker, Marco Platzner, and Serge Vernalde, editors, *FPL*, volume 3203 of *Lecture Notes in Computer Science*, pages 95–104. Springer, 2004.
- [GHF<sup>+</sup>06] Michael Gschwind, H. Peter Hofstee, Brian K. Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in Cell’s multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

- [GKMK96] Maya Gokhale, James Kaba, Aaron Marks, and Jang Kim. A malleable architecture generator for FPGA computing. In *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic, Proc. SPIE 2914*, pages 208–217, 1996.
- [GMLC04] Altaf Abdul Gaffar, Oskar Mencer, Wayne Luk, and Peter Y. K. Cheung. Unifying bit-width optimisation for fixed-point and floating-point designs. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 79–88, Washington, DC, USA, 2004. IEEE Computer Society.
- [GPL<sup>+</sup>03] Maria Jesús Garzarán, Milos Prvulovic, José M. Llaberia, Victor Viñals, Lawrence Rauchwerger, and Josep Torrellas. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 191–202, Feb. 2003.
- [GSAK00] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 49, Washington, DC, USA, 2000. IEEE Computer Society.
- [GST05] Dominik Göddeke, Robert Strzodka, and Stefan Turek. Accelerating double precision FEM simulations with GPUs. In *Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique*, Frontiers in Simulation, pages 139–144. SCS Publishing House e.V., September 2005.
- [GTV<sup>+</sup>04] Aman Gayasen, Yuh Tsai, Narayanan Vijaykrishnan, Mahmut Kandemir, M. Jane Irwin, and Tim Tuan. Reducing leakage energy in FPGAs using region-constrained placement. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 51–58, New York, NY, USA, 2004. ACM.

- [GVH05] Yongfeng Gu, Tom VanCourt, and Martin C. Herbordt. Accelerating molecular dynamics simulations with configurable circuits. In *FPL '05: Proceedings of the International Conference on Field Programmable Logic and Applications, 2005*, pages 475–480, Aug. 2005.
- [HBCC99] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, 1999.
- [HF92] Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *SIGPLAN Not.*, 27(5):1–52, 1992.
- [Hig93] Nicholas J. Higham. The accuracy of floating point summation. *SIAM J. Sci. Comput.*, 14:783–799, 1993.
- [Hig02] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.
- [HK95] Reiner W. Hartenstein and Rainer Kress. A datapath synthesis system for the reconfigurable datapath architecture. In *Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International Conference on Hardware Description Languages; IFIP International Conference on Very Large Scale Integration., Asian and South Pacific*, pages 479–484, Aug-1 Sep 1995.
- [HMZ<sup>+</sup>08] Paul M. Harvey, Rohan Mandrekar, Yaping Zhou, J. Zheng, J.J. Maloney, Steven Cain, Kazushige Kawasaki, Gary Lafontant, Hirokazu Noma, Katie Imming, T. Plachy, and David Questad. Packaging the Cell Broadband Engine microprocessor for supercomputer applications. In *Electronic Components and Technology Conference, 2008. ECTC 2008. 58th*, pages 1368–1371, May 2008.
- [HP98] John L. Hennessy and David A. Patterson. *Computer organization and design (2nd ed.): the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

- [HU06] K. Scott Hemmert and Keith D. Underwood. Open source high performance floating-point modules. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 349–350, Washington, DC, USA, 2006. IEEE Computer Society.
- [HYL<sup>+</sup>09] Chun Hok Ho, Chi Wai Yu, Philip Leong, Wayne Luk, and Steven J. E. Wilton. Floating-point FPGA: Architecture and modeling. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(12):1709–1718, Dec. 2009.
- [IEE85] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, 1985.
- [IEE08] IEEE Computer Society. *IEEE 754-2008, Standard for Floating-Point Arithmetic*, 2008.
- [Int08] Intel Corporation. *Intel C/C++ Compiler for Linux*, 2008.
- [Int09] Intel Corporation. *Intel 64 and IA-32 Architectures: Software Developers Manual*. Intel Corporation, [www.intel.com](http://www.intel.com), March 2009.
- [JL77] W. Kenneth Jenkins and Benjamin J. Leon. The use of residue number systems in the design of finite impulse response digital filters. *Circuits and Systems, IEEE Transactions on*, 24(4):191–201, Apr 1977.
- [Joh] Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, N.J.
- [Kah83] William Kahan. Mathematics written in sand. In *Proc. Joint Statistical Mtg. of the American Statistical Association*, pages 12–26, 1983.
- [KBT08] Dirk Koch, Christian Beckhoff, and Jürgen Teich. Recobuilder – a novel tool and technique to build statically and dynamically reconfigurable systems for FPGAs. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 119–124, Sept. 2008.

- [KC93] Kishore Kota and Joseph R. Cavallaro. Numerical accuracy and hardware tradeoffs for CORDIC arithmetic for special-purpose processors. *Computers, IEEE Transactions on*, 42(7):769–779, Jul 1993.
- [KD07] Nachiket Kapre and Andre DeHon. Optimistic parallelization of floating-point accumulation. In *ARITH '07: Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 205–216, Washington, DC, USA, 2007. IEEE Computer Society.
- [KDM92] David C. Ku and Giovanni De Micheli. *High level synthesis of ASICs under timing and synchronization constraints*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.
- [KH95] Erich Kaltofen and Markus A. Hitz. Integer division in residue number systems. *IEEE Trans. Comput.*, 44(8):983–989, 1995.
- [KL94] Asawaree Kalavade and Edward A. Lee. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *CODES '94: Proceedings of the 3rd international workshop on Hardware/software co-design*, pages 42–48, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [KP06] Volodymyr Kindratenko and David Pointer. A case study in porting a production scientific supercomputing application to a reconfigurable computer. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 13–22, Washington, DC, USA, 2006. IEEE Computer Society.
- [KR07] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, February 2007.
- [Lan08] Martin Langhammer. Floating point datapath synthesis for FPGAs. In *FPL '08: International Conference on Field Pro-*

*programmable Logic and Applications, 2008.*, pages 355–360, Sept. 2008.

- [LCD<sup>+</sup>00] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Jon Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 507–512, New York, NY, USA, 2000. ACM.
- [LCH<sup>+</sup>03] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. A compiler framework for speculative analysis and optimizations. *SIGPLAN Not.*, 38(5):289–299, 2003.
- [LCM<sup>+</sup>05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.
- [LCT<sup>+</sup>09] Wayne Luk, Jose Gabriel Coutinho, Timothy J. Todman, Yuet Ming Lam, William Osborne, Kong Woei Susanto, Qiang Liu, and W.S. Wong. A high-level compilation toolchain for heterogeneous systems. In *Proceedings of the IEEE System on a Chip Conference*, September 2009.
- [LKM02] Gerhard Lienhart, Andreas Kugel, and Reinhard Männer. Using floating-point arithmetic on FPGAs to accelerate scientific N-Body simulations. In *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*, pages 182–191, 2002.
- [LM97] N.K. Laurance and Donald M. Monroe. Embedded DCT coding with significance masking. In *Acoustics, Speech, and Signal Processing, 1997. ICASSP-97., 1997 IEEE International Conference on*, volume 4, pages 2717–2720 vol.4, Apr 1997.

- [LMM<sup>+</sup>98] Walter B. Ligon III, Scott McMillan, Greg Monn, Kevin Schoonover, Fred Stivers, and Keith D. Underwood. A re-evaluation of the practicality of floating point operations on FPGAs. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 206–215, Los Alamitos, CA, 1998. IEEE Computer Society Press.
- [LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.
- [LO04] Shun Long and Michael O’Boyle. Adaptive Java optimisation using instance-based learning. In *ICS ’04: Proceedings of the 18th annual international conference on Supercomputing*, pages 237–246, New York, NY, USA, 2004. ACM Press.
- [LR04] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Not.*, 39(4):473–489, 2004.
- [LSS<sup>+</sup>09] Stefan M. Larson, Christopher D. Snow, Michael Shirts, Vijay S. P, and Vijay S. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. *ArXiv e-prints*, January 2009.
- [LT03] Jian Liang and Russell Tessier. Floating point unit generation and evaluation for FPGAs. In *FCCM ’03: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, 2003*, pages 185–194. Society Press, 2003.
- [LWP94] Wayne Luk, Teddy Wu, and Ian Page. Hardware-software codesign of multidimensional programs. In *Proceedings of FCCM94, D. Buell and K.L. Pocek (eds.), IEEE Computer*, pages 82–90. Society Press, 1994.
- [Men06] Oskar Mencer. ASC: a stream compiler for computing with FPGAs. *Computer-Aided Design of Integrated Circuits*

- and Systems, *IEEE Transactions on*, 25(9):1603–1617, Sept. 2006.
- [MHMF00] Oskar Mencer, Heiko Hbert, Martin Morf, and Michael J. Flynn. Stream: Object-oriented programming of stream architectures using PAM-Blox. In *Field-Programmable Logic and Applications, LNCS 1896*, pages 595–604. Springer, 2000.
- [Mil26] William E. Milne. Numerical integration of ordinary differential equations. *The American Mathematical Monthly*, 33(9):455–460, Nov 1926.
- [MMF98] Oskar Mencer, Martin Morf, and Michael J. Flynn. PAM-Blox: high performance FPGA design for adaptive computing. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 167–174, Apr 1998.
- [MPHL03] Oskar Mencer, David J. Pearce, Lee W. Howes, and Wayne Luk. Design space exploration with A Stream Compiler. In *Proc. IEEE Intl Conf. Field-Programmable Technology*, pages 270–277, 2003.
- [MR59] William E. Milne and R. R. Reynolds. Stability of a numerical solution of differential equations. *J. ACM*, 6(2):196–203, 1959.
- [MSMD00] Oskar Mencer, Luc Séméria, Martin Morf, and Jean-Marc Delosme. Application of reconfigurable CORDIC architectures. *J. VLSI Signal Process. Syst.*, 24(2/3):211–221, 2000.
- [NH05] Naohito Nakasato and Tsuyoshi Hamada. Astrophysical hydrodynamics simulations on a reconfigurable system. In *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 279–280, April 2005.
- [NR 02] NR Adiga et al. An overview of the BlueGene/L Supercomputer. In *Supercomputing '02: Proceedings of the 2002*

- ACM/IEEE conference on Supercomputing*, pages 1–22, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.
- [NS03] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.
- [NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM.
- [PO03] Manohar K. Prabhu and Kunle Olukotun. Using thread-level speculation to simplify manual parallelization. *SIGPLAN Not.*, 38(10):1–12, 2003.
- [Pop96] Paul L. A. Popelier. MORPHY, a program for an automated “atoms in molecules” analysis. *Computer Physics Communications*, 93:212–240, February 1996.
- [Rap90] Joseph Raphson. Analysis æquationum universalis. seu ad equationes algebraicas resoh’endas methodus generalis, et expedita, ex nova Infinitarum serierum doctrina, deducta ac demonstrata. 1690. Original in the British Library, London.
- [Rob58] James E. Robertson. A new class of digital division methods. *IEEE Trans. Comput.*, pages 218–220, 1958.
- [Sad75] Robert Sadourny. The Dynamics of Finite-Difference Models of the Shallow-Water Equations. *Journal of Atmospheric Sciences*, 32:680–689, April 1975.
- [SBA00] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 108–120, New York, NY, USA, 2000. ACM.

- [SCNS83] E. Earl Swartzlander, Jr., D. V. Satish Chandra, H. Troy Nagle, Jr., and Scott A. Starks. Sign/logarithm arithmetic for FFT implementation. *IEEE Trans. Comput.*, 32(6):526–534, 1983.
- [SCZM00] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *In Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12, 2000.
- [SG00] Mihai B. Seth and Seth C. Goldstein. Bitvalue inference: Detecting and exploiting narrow bitwidth computations. In *Proceedings of the EuroPar 2000 European Conference on Parallel Computing*, pages 969–979. Springer Verlag, 2000.
- [SG06] Robert Strzodka and Dominik Göldeke. Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components. In *FCCM '06: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, 2006*, pages 259–268, April 2006.
- [SGTP06] Ronald Scrofano, Maya Gokhale, Frans Trouw, and Viktor K. Prasanna Prasanna. Hardware/software approach to molecular dynamics on reconfigurable computers. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 23–34, April 2006.
- [SHD02] Byoungro So, Mary W. Hall, and Pedro C. Diniz. A compiler approach to fast hardware design space exploration in FPGA-based systems. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 165–176, New York, NY, USA, 2002. ACM.
- [SHF06] Sushant Sharma, Chung-Hsing Hsu, and Wu-chun Feng. Making a case for a Green500 list. In *IEEE International*

*Parallel and Distributed Processing Symposium (IPDPS 2006) / Workshop on High Performance - Power Aware Computing*, 2006.

- [SJ02] Li Shang and Niraj K. Jha. Hardware-software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable FPGAs. In *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, page 345, Washington, DC, USA, 2002. IEEE Computer Society.
- [SL05] Henry Styles and Wayne Luk. Compilation and management of phase-optimized reconfigurable systems. In *FPL '05: Proceedings of the International Conference on Field Programmable Logic and Applications, 2005.*, pages 311–316, Aug. 2005.
- [SM98] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. pages 2–13, 1998.
- [Smi98] James E. Smith. A study of branch prediction strategies. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 202–215, New York, NY, USA, 1998. ACM.
- [SW] Robert Sedgewick and Kevin Wayne. *Introduction to Computer Science in Java*.
- [SWA95] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machines. In *FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on*, pages 155–162, Apr 1995.
- [TCW<sup>+</sup>05] Timothy J. Todman, George A. Constantinides, Steven J. E. Wilton, Oskar Mencer, Wayne Luk, and Peter Y. K. Cheung. Reconfigurable computing: architectures and design methods. *Computers and Digital Techniques, IEE Proceedings*-, 152(2):193–207, 2005.

- [Tom67] Robert M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.
- [TVVA03] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 204–215, Washington, DC, USA, 2003. IEEE Computer Society.
- [UH04] Keith D. Underwood and K. Scott Hemmert. Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 219–228, Washington, DC, USA, 2004. IEEE Computer Society.
- [Und04] Keith D. Underwood. FPGAs vs. CPUs: trends in peak floating-point performance. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 171–180, New York, NY, USA, 2004. ACM Press.
- [Und10] Underbit Technologies. *MAD MP3 Decoder Reference Code*. <http://www.underbit.com/products/mad/>, February 2010.
- [VB04] James Van Verth and Lars Bishop. *Essential Mathematics for Games and Interactive Applications: A Programmer's Guide*, pages 162–172. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.
- [Vol59] Jack E. Volder. The CORDIC computing technique. *Proceedings of the Western Joint Computer Conference*, pages 257–261, 1959.
- [Vol00] Jack E. Volder. The birth of CORDIC. *J. VLSI Signal Process. Syst.*, 25(2):101–105, 2000.
- [Wal71] John S. Walther. A unified algorithm for elementary functions. In *AFIPS '71 (Spring): Proceedings of the May 18-20*,

- 1971, *spring joint computer conference*, pages 379–385, New York, NY, USA, 1971. ACM.
- [WBL06] Xiaojun Wang, Sherman Braganza, and Miriam Leeser. Advanced components in the variable precision floating-point library. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society.
- [Wil94] James H. Wilkinson. *Rounding Errors in Algebraic Processes*. Dover Publications, Incorporated, 1994.
- [Wol91] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer, Second Edition*. Addison-Wesley, 1991.
- [WSO<sup>+</sup>06] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the Cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM Press.
- [WSO<sup>+</sup>07] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. Scientific computing kernels on the Cell processor. *Int. J. Parallel Program.*, 35(3):263–298, 2007.
- [Xil06] Xilinx. *Floating-Point Operator v3.0*, September 2006.
- [Xil07] Xilinx. *Multiplier v10.0*, April 2007.
- [ZDL<sup>+</sup>08] Jie Zhou, Yong Dou, Yuanwu Lei, Jinbo Xu, and Yazhuo Dong. Double precision hybrid-mode floating-point FPGA CORDIC co-processor. In *HPCC '08: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 182–189, Washington, DC, USA, 2008. IEEE Computer Society.
- [ZDLD08] Jie Zhou, Yong Dou, Yuanwu Lei, and Yazhuo Dong. Hybrid-Mode Floating-Point FPGA CORDIC Co-processor.

In *ARC '08: Proceedings of the 4th international workshop on Reconfigurable Computing*, pages 256–261, Berlin, Heidelberg, 2008. Springer-Verlag.

[ZP05] Ling Zhuo and Viktor K. Prasanna. Sparse matrix-vector multiplication on FPGAs. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, pages 63–74, New York, NY, USA, 2005. ACM Press.

[ZRL96] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: a step toward practical analyses. *SIGSOFT Softw. Eng. Notes*, 21(6):81–92, 1996.