Profile-directed speculative optimization of reconfigurable floating point data paths

Ashley W Brown[1], Paul H J Kelly[1], Wayne Luk[1]

Imperial College London

Abstract. This paper presents a methodology for generating floating-point arithmetic hardware designs which are, for suitable applications, dramatically reduced in size, while still retaining performance. We use a profiling tool for floating-point value ranges to identify arithmetic operations where the shifting required for alignment and normalisation is almost always small. We synthesise hardware with reduced-size barrel-shifters, but always detect when operands lie outside the range this optimised hardware can handle. These rare out-of-range operations are handled by a separate full floating-point implementation, either on-chip or by returning calculations to the host. Thus the system suffers no compromise in IEEE754 compliance. This paper presents results for two benchmark applications which profiling suggested would be profitable. We demonstrate the potential for this technique to yield an increase in parallel computing power of up to 43%, with a (correctable) error rate of less than 5%.

1 Introduction

The core focus of our research is the profile-driven optimisation of FPGA floating-point, targetted at scientific applications. Our approach is aggressive and speculative: we permit optimisations to cause errors, but respond to erroneous calculations by re-executing them with a more capable floating point implementation.

To this end, we have constructed a Valgrind[Neth07]-based floating-point profiler, FloatWatch, to collect value-range and operand-relationship information about our target applications. The profiler has allowed us to identify interesting trends in scientific applications and we have previously suggested a range of possible optimisations for which value-range profile data may be useful[Brow07].

In this paper we present preliminary results for one particular style of optimisation: reducing the capabilities of the floating-point unit, allowing a decrease in size and corresponding increase in parallelism. Specifically, we explore the effect of reducing operand aligment and normalisation logic on hardware size and calculation error rates. We remain IEEE-754 compliant by re-executing erroneous calculations with a more capable floating point implementation. Additionally, we briefly present our current tool-chain for profile-guided optimisation work and describe the test hardware in use.

We are evaluating our optimisations using both standard benchmarks and real-world scientific code; we have selected one of each for presentation here.

Section 2 describes related work and its applicability to the methods described here. Section 3 describes our optimisations and both the tool-chain and hardware currently being used for testing. Finally, Section 5 describes our target applications, before presenting hardware reduction and error-rate results after optimisation.

2 Background

Floating point formats provide a compact machine representation of scientific notation, with a normalised significand and exponent. IEEE-754 [IEEE] is now a well-established standard used in general purpose floating-point units. The current revision of IEEE-754 defines four types of floating-point representation, with the same basic layout – a sign bit, significand and exponent – but different bit sizes for each. The formats are broken into parts, 'standard' formats which are explicitly defined and 'extended' formats which are less strictly contrained. 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. Figure 1 shows the basic layout of an IEEE-754 floating point adder, with possibilities for optimisation highlighted.

Exploiting the power of FPGAs requires many calculations to be performed in parallel, but for floating-point this requires many floating-point units to be running in parallel. Double-precision floating-point units required for many scientific computations occupy a large amount of resources when placed on an FPGA, limiting the number of parallel operations. A variety of tools aim to reduce the required size using a static analysis of the problem at hand. The BitSize [Gaff04] tool allows this type of refinement looking at source-code alone. Cheung et al [Cheu05] developed a method for generating fixed-point versions of elementary functions, such as logarithm and square root, while using IEEE floating point for input an output. The conversion between IEEE floating point and fixed point is transparent to the user.

These possibilities for optimisation have a pitfall: altering the bit-width of the significand or making use of fixed-point may alter the approximation and change the final result.

3 Proposed Optimisation

We have focused our efforts on two areas of floating point units which our profiler indicated should be amenable to our optimisation approach – two elements of a typical floating point unit contribute a considerable amount to their size, but play no role in the actual calculation itself. The first of these elements occurs only for addition and subtraction operations, where operands must be aligned to have the same exponent before the operation is executed. The second element occurs for all floating point operations, where a result must be normalised and

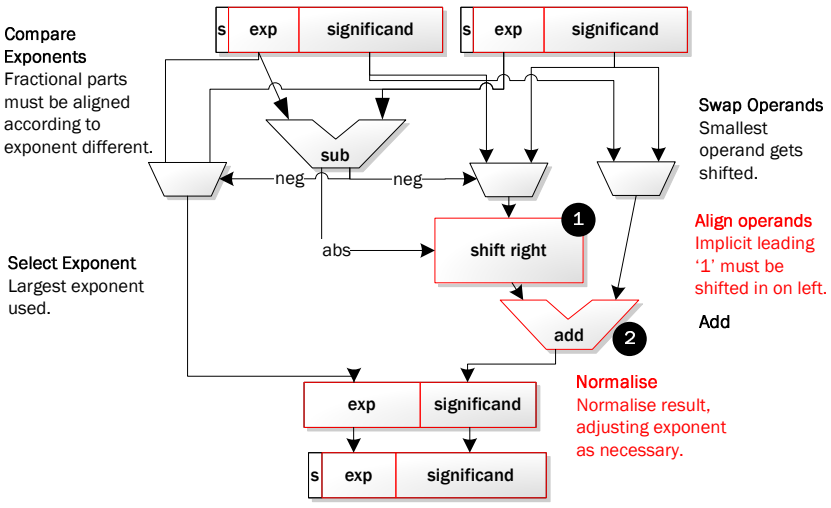


Fig. 1: Optimisation opportunities for a floating-point adder -
1) - Operand alignment stage reduced if difference between exponents is small.
2) - Adder size reduced with smaller significand size.

packed into the IEEE-754 format after the operation has completed. Both of these elements typically rely on a barrel shifter for one-cycle operation, although multi-stage high-latency options are also available to provide a higher clock-rate.

The classical way of implementing a barrel shifter is with a cascaded set of wide multiplexers, leading 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, again begin to consume routing resources to tie a number of macro-blocks together.

In this paper we explore reducing the size of shifting logic, shrinking the floating point unit without affecting precision. The maximum shift is reduced, shrinking the hardware size but imposing limitations on the input operands: operations which require large shifts for operand alignment or normalisation cannot be successfully executed with our changes in place. Errors are easily identified during execution as the maximum shift required must still be calculated, allowing erroneous results to be flagged and re-executed on the host.

Figure 2 provides an illustration of resource savings when using our optimisation technique.

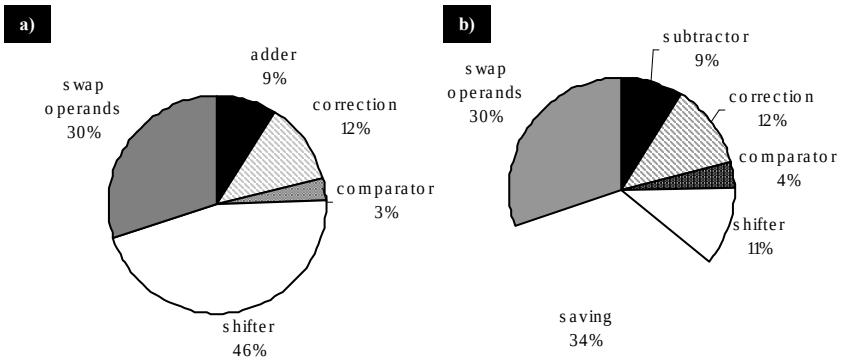


Fig. 2: Charts showing resource distribution, given as Altera Stratix II ALUTs, between functional areas on a floating point adder from the RPL library (a) and the savings made by our optimisation (b). The components are as follows:
adder - significand addition operation.
correction - conversion of operand from internal format to floating point.
comparator - operand exponent comparison.
shifter - the shift operation for operand alignment.
swap operands - swap of operands so smallest is shifted.

4 Experimental Design

The first stage in the optimisation process is profiling with FloatWatch, our Valgrind-based profiler. FloatWatch is able to collect a variety of different data about the behaviour of floating point code, but for speed of profiling we only collect operand values in this case. Due to the high storage required the tool is unable to keep a record of every operand value encountered, instead aggregating the values into discrete buckets based on the values' exponents. Values are stored on an operation-by-operation basis and aggregated into source-code lines by the user interface. The "hot" regions of code which execute the highest proportion of floating point instructions are also identified by the profiler to aid optimisation.

The two applications we have chosen both exhibit similar behaviour: operand values in the "hot" code section fall into a narrow range, far smaller than that offered by the IEEE-754 standard. Where this pattern is seen floating-point operations become suitable for the particular optimisations presented here. We have previously described profiling a number of sample codes[Brow07], including those not amenable to this technique.

Following the identification of good candidates a data-flow graph fragment is fed into our optimisation tool, which generates floating-point datapaths from a set of single-operation building blocks. These are connected to represent the data-flow graph, with additional connections to raise exception flags and control data initialisation. FPGA vendors 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 they type we are proposing here. Instead we use the RPL Floating Point library[P. B02], a VHDL variable-precision library with a clean code structure. 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.

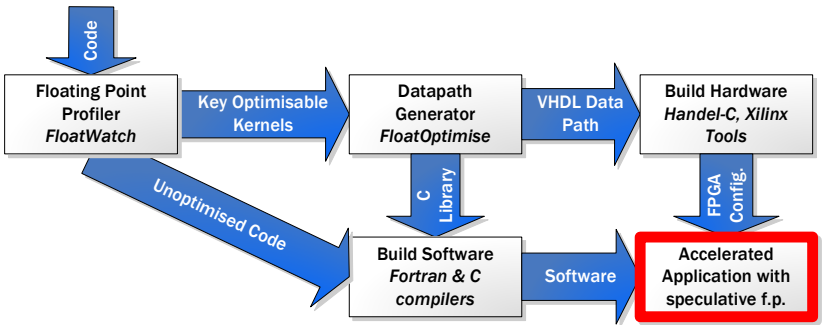


Fig. 3: The optimisation tool-chain

At the core of the RPL library is a generic barrel shifter following the cascaded-multiplexer design. Each stage shifts by a fixed power of two, providing the full range of possible shifts. Reducing the maximum allowable shift removes one stage from the cascade and provides a corresponding hardware reduction.

The data-path generator allows the user to specify the desired size of the exponent and significand; at present we use IEEE-754 standard sizes for these parameters, in keeping with the original source code. A further option specifies the maximum shift desired for both normalisation and operand alignment, given as a power of two due to the system architecture. In a standard double-precision implementation a maximum shift of 52 places would be required (the size of the significand), given as 2^6. The graphs presented in Section 5 show results from 2^6 down to 2^1, the full range available. We do not discriminate between operand alignment and normalisation, shrinking the maximum shift amount in both cases. Consequently we see a reduction in hardware size for additions, subtractions and multiplications, which would not be seen if focusing solely on operand alignment.

As seen in Figure 3, our optimisation tool generates a software library in addition to the VHDL floating-point datapath. The software library can be linked against the original application, which must currently be hand-modified to call the library instead of executing the "hot" code segment. The VHDL is passed to vendor place-and-route tools and combined with host-to-FPGA hardware to produce a complete FPGA design capable of communicating with the software.

Our test hardware currently consists of a Celoxica RC-10 development board, containing a low-end Xilinx Spartan 3 and slow USB communication link to

the host PC. Although this is unsuitable for gathering performance results, it has allowed us to test the resulting hardware and check error rates without waiting for our high performance PC+FPGA system to arrive. We also explored hardware use on a Xilinx Virtex 5 FPGA, the chip expected to feature in our high performance system. This will have a dual-socket motherboard containing a current-generation Intel processor and Virtex 5 connected via the Intel Front Side Bus standard.

Due to the constraints of our current hardware we use fragments of code from within hot loops, sending data to the FPGA on every cycle. This is sufficient to test error behaviour however for performance our final design requires multiple parallel datapaths behaving as a complex vector instruction.

5 Results

For this paper we have selected the applications amenable to this style of optimisation. One is a benchmark, "swim" from SpecFP95, while the other is part of a library used for computational chemistry.

The "swim" benchmark is a weather predictor based on shallow-water equations, using finite difference approximations. The benchmark is declared in C to use single-precision floats as storage, however current x86 processors execute all operations in extended precision (80-bit) unless SIMD instructions are used. Our optimised version of this application uses a 32-bit single-precision floating point data-path rather than a 64-bit or 80-bit execution as used in the x86 floating point unit.

The code we refer to as "ydl_pij" is an iterative solver for computational chemistry, using the Molecular Mechanics - Valence Bond method[BM03]. The code has a wide variety of uses, including the modelling of electromagnetic properties. The code is double-precision throughout and this is mirrored by a 64-bit datapath in our optimised version.

The remainder of this section describes the results of the optimisation process seen in Section 4, as applied to our two sample applications. Some of the profiling results for our applications have been presented previously, however they are worth re-visiting.

5.1 "ydl_pij" (Molecular Mechanics – Valence Bond Method)

Figure 4a shows a histogram of operand values encountered during profiled execution of "ydl_pij" with five datasets. It demonstrates an almost ideal graph for this optimisation: operands fall into narrow ranges, approximately symmetrical about zero. This suggests that operands should have similar exponents and require limited alignment for additions and substractions. There are low levels of extra values outside the ranges shown on the graph which makes the situation less ideal: these out-of-range values will require large shifts for operand alignment and have the potential to increase the error rate if they occur too frequently.

The profiler reveals that a relatively simple section of code in an inner-loop is responsible for most of the floating-point operations:

H(U)=H(U)+FINT∗VALUE∗VR(V)
H(V)=H(V)+FINT∗VALUE∗VR(U)

This was converted into a six-input, two-output dataflow graph and passed to the datapath generator, using the IEEE-754 double-precision standard. Figure 4b shows the effect on error rate of varying the maximum shift amount possible, from 2^6 down to 2^1. Three datasets were used, including one not profiled previously. For two of the datasets, 12_2 and the un-profiled 13_c2v, error rates remain under 6% even with the maximum shift reduced to its lowest level. The 13_d3h dataset sees error rates rise to 10%. The most striking part of this graph is the sharp drop between 2^1 and 2^2 – the error rate falls by around two-thirds by adding just one extra stage to the shift. Figure 4b also shows the effect on hardware size of adding this stage back: on the Spartan 3 it adds around 3% to the relative size, but in the Virtex 5 architecture it has no effect at all.

Figure 4c presents the previous results in a different form, comparing the increase in parallelism and error rate arising from the variation of maximum shift amount. We measure parallelism as the number of datapaths which could be placed onto the FPGA for a given configuration, currently ignoring overheads. We achieve a theoretical improvement of 43% after shrinking the datapath, with an error rate of just 5%.

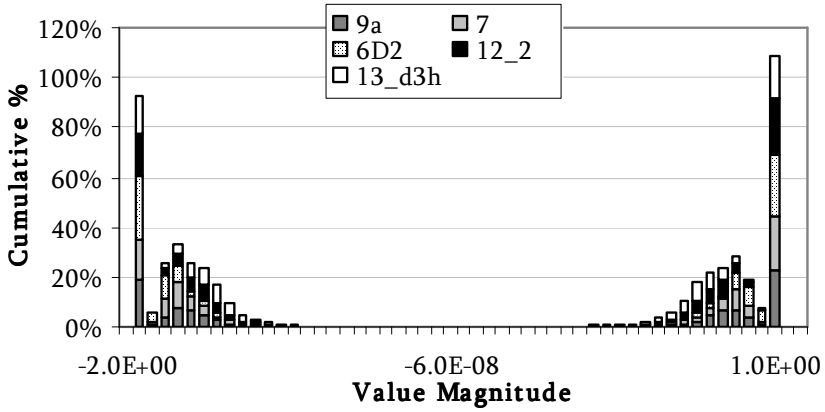
5.2 SpecFP95"swim"

Figure 6 shows a histogram of value occurence for one of the most frequently-executed lines of code in the "swim" benchmark, 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 and hence inconvenient for locality-based optimisations. Drilling down to an operation level reveals that each operation has a much smaller range of operand values – multiplications within the statement move this range around, resulting in 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.

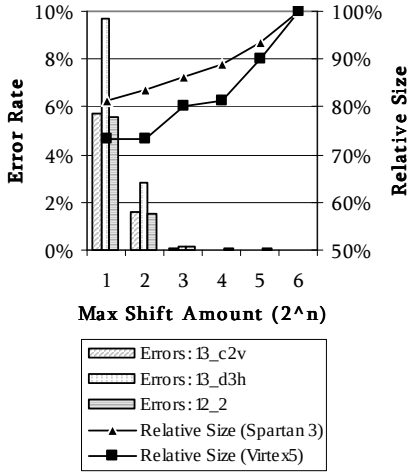
This statement can be converted into a three-input, one-output datapath, with "2" and "ALPHA" encoded in hardware as constants. Figure 5 presents the variation in error rate and Spartan 3 hardware size together. Over a 30% decrease in hardware resources is seen with an error rate of just 0.05%, however this may say more about the reliability of benchmarks than the success of the approach in this case.

6 In Context

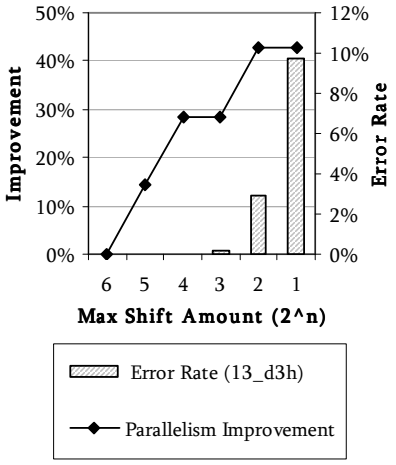
In this paper we have presented just one possible option for reducing the capabilities of a floating point unit in order to reduce its size. Our wider work considers a number of other techniques, with the goal of retaining IEEE-754 compliance in



(a) Histogram showing value distribution for five datasets



(b) Graph showing variation of error rate with maximum shift amount, for three datasets, overlaid with relative hardware size on two test systems



(c) Increase in potential parallelism and error rate with variation in maximum shift amount

Fig. 4: Profile and optimisation results for the "ydl_pij" application.

the final result, to the same precision as native execution. The FloatWatch profiler is central to these techniques, informing decisions about which optimisations may be applicable.

The optimisations presented here rely on re-execution of erroneous results, however if the error rate climbs too high any gains from parallelism are lost. The reconfigurable nature of FPGAs permits an adaptive optimisation strategy, reconfiguring the chip with an alternative implementation in the case of a high

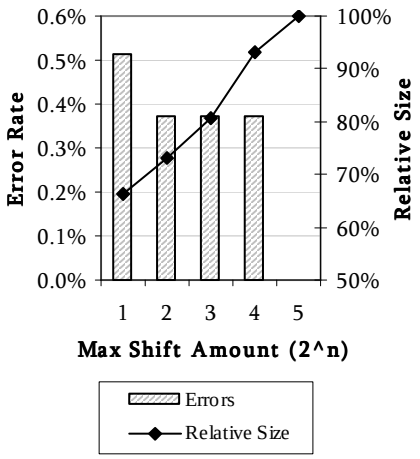


Fig. 5: Graph showing the variation of error rate and hardware size with maximum shift amount for the "swim" benchmark.

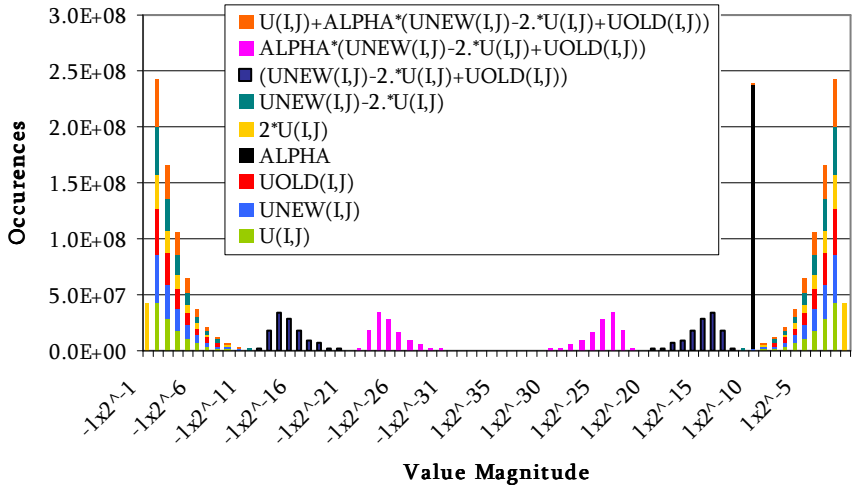


Fig. 6: Profile of hot code fragment from the "swim" benchmark, broken down by individual operation

error rate. The software would be able to dynamically trade-off parallelism and error rate to achieve the highest performance.

Static optimisation strategies typically look to reduced the size of the floating-point representation, using single-precision floating-point, fixed-point or bespoke formats. Both communication overhead and execution hardware can be reduced in this manner. We plan to evaluate a similar solution, reducing the size of the

floating-point exponent while retaining full precision. Reducing the exponent reduces the representable range, however our profile results indicate the full range is rarely needed. Adjusting the bias of the exponent allows the narrow range to be moved, providing support for value distributions as seen in Figure 6.

7 Conclusion

We have presented a method for reducing the size of floating-point hardware whilst adhering to the IEEE-754 standard. While our optimisations generate calculation errors these are both easily detectable and correctable. Our technique gambles on the error rate being low enough to allow re-execution of erroneous answers without a high overall penalty. Our floating-point profiler allows us to identify applications and code segments amenable to our optimisations.

In our evaluation of two code segments we have demonstrated error rates of between 0.05% and 10%, with improvements in parallelism of up to 43% when the full extent of our optimisation is applied. Furthermore, we have illustrated that slightly reducing the extent of the optimisation can lead to a dramatic improvement in error rate.

References

BM03. B. Bearpark MJ. Excited states of conjugated hydrocarbon radicals using the molecular mechanics - valence bond (MMVB) method. *THEORETICAL CHEMISTRY ACCOUNTS*, pages 105–114, 2003.

Brow07. A. Brown, P. Kelly, and W. Luk. Profiling floating point value ranges for reconfigurable implementation. In *informal proceedings of the Workshop on Reconfigurable Computing, HiPEAC 2007*, 2007.

Cheu05. C. Cheung, D. Lee, O. Mencer, P. Cheung, and W. Luk. Automating Custom-Precision Function Evaluation for Embedded Processors. 2005.

Gaff04. A. Gaffar, O. Mencer, W. Luk, and P. 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.

IEEE. IEEE. IEEE Standard for Binary Floating-Point Arithmetic.

Neth07. N. Nethercote and J. 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.

P. B02. P. Belanović and M. 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.