Profiling floating point value ranges for reconfigurable implementation

Ashley W Brown, Paul H J Kelly,
Wayne Luk

* *Department of Computing, Imperial College London, United Kingdom*

ABSTRACT

Reconfigurable architectures offer potential for performance enhancement by specializing the implementation of floating-point arithmetic. This paper presents FloatWatch, a dynamic execution profiling tool designed to identify where an application can benefit from reduced precision or reduced range in floating-point computations. FloatWatch operates on x86 binaries, and generates a profile output file recording, for each instruction and line of source code, the overall range of floating-point values, the bucketised sub-ranges of values, and the maximum difference between 64-bit and 32-bit executions.

We present results from the tool on a suite of four benchmark codes. Our tool indicates potential performance loss due to denormal values, and helps to identify opportunities for using dual fixed-point arithmetic representation which has proved effective for reconfigurable designs. Our results show that applications often have highly modal value distributions, offering promise for aggressive floating-point arithmetic optimisations.

1 Introduction

Scientific applications often require high accuracy, high data throughput and high speed calculations. Most commodity hardware sacrifices accuracy for speed, potentially limiting its usefulness in the scientific arena. Single-instruction multiple-data (SIMD) architectures and graphics cards provide high speed calculations on IEEE single precision floating point only. Other architectures, such as IBM's Cell [Hofs05] have also prioritised single precision floating point to target the games market.

In the reconfigurable computing space, placing a full-featured floating point unit onto an FPGA consumes vast amounts of on-chip resources for even single-precision floating point. Re-using the same unit repeatedly is possible, but introduces an artificial bottleneck. Knowledge of the likely value ranges which will reach a floating point unit allows us to refine both the data representation and the functional units to reduce resources whilst still providing performance. The BitSize [Gaff04] tool allows this type of refinement looking at source-code alone and may be a useful companion to FloatWatch. Cheung et al [pub05] 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.

Conversion of scientific software to single-precision floating point would permit better use of this commodity hardware, however with an associated loss of accuracy. With reconfigurable fabrics at our disposal custom floating point representations are possible. Moreover, 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 and Luk [Styl05].

2 Tool Structure

The FloatWatch tool operates on x86 binaries compiled with debugging information, under the Valgrind [Neth03] dynamic instrumentation framework.

Output consists of a raw data file containing profiling information and a dynamic HTML user interface to manipulate and explore the data. Alternatively the data may be exported for plotting in GNUPlot or Excel.

FloatWatch provides the following information for each assembly instruction in the program:

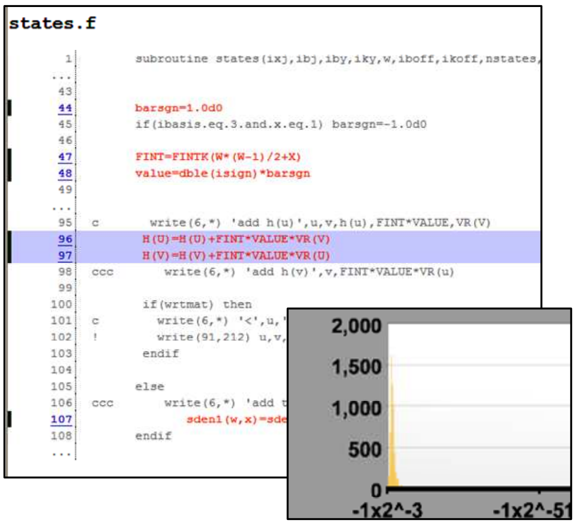
- Overall range of values
- Bucketised sub-ranges of values
- Maximum difference between 64-bit and 32-bit floating point executions

The information can be aggregated for each line, then on a line-by-line basis by the user. Figure 1 shows the source display and value graph provided by the HTML user interface.

FloatWatch was conceived to provide an insight into the behaviour of scientific code, which is often "dusty desk" software where the authors have long since left the organisation concerned. During attempts to accelerate some scientific applications it was realised that we had very little insight into the overall behaviour of the code, or characteristics of particular frequently-executed sections. Valgrind provides a convenient base to build upon, although performance is currently an issue.

FloatWatch operates as a tool in the Valgrind framework, much like Cachegrind and Callgrind [Weid04]. Valgrind reads x86 and PowerPC binaries, converting them to an intermediate representation consisting of simple operations. 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.

The FloatWatch instrumentation tool adds instrumentation code to track the results of floating point operations in the target application, optionally inserting single precision versions of double precision operands, then tracking the difference between single and double results.



states.f

```
      subroutine states(ixj,ibj,iby,iky,w,iboff,ikoff,nstates,

      barsgn=1.0d0
      if(ibasis.eq.3.and.x.eq.1) barsgn=-1.0d0

      FINT=FINTK(W*(W-1)/2+X)
      value=dble(isign)*barsgn

c       write(6,*) 'add h(u)',u,v,h(u),FINT*VALUE,VR(V)
       H(U)=H(U)+FINT*VALUE*VR(V)
       H(V)=H(V)+FINT*VALUE*VR(U)
ccc       write(6,*) 'add h(v)',v,FINT*VALUE*VR(u)

       if(wrtmat) then
c        write(6,*) '<',u,'
!        write(91,212) u,v,
       endif

      else
ccc       write(6,*) 'add t
             sden1(w,x)=sde
      endif
```

Figure 1: Exploring profile results using the FloatWatch user 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.

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. The FloatWatch post-processor takes this raw output and the application source files, producing an HTML+JavaScript report which 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 same technique may be used to track integer values if required, however scientific applications are the focus here.

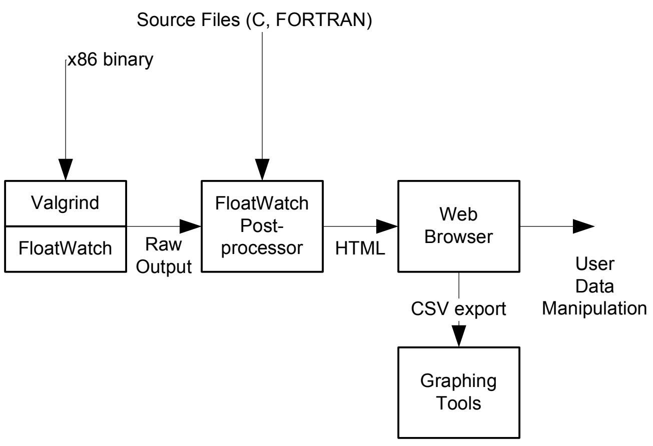


Figure 2: Block Diagram of the FloatWatch Tool Chain.

3 Optimisation Opportunities

The data produced by FloatWatch helps guide the process of optimisation for targets with a variety of floating point configurations. 32-bit vs 64-bit comparisons provide an intuition as to the safety of execution on a GPU, for example. Analysis of data ranges may guide the implementation of a custom floating point unit which is more efficient in the active ranges.

This section provides an overview of some possible modifications or optimised implementations.

3.1 Optimised Floating Point Unit

For some of our real-world applications, ranges are the same or similar across a wide variety of real-world datasets. With knowledge of likely value ranges available it becomes possible to design optimised floating point units, which have their highest performance within these ranges. Outside the "standard" ranges a smaller, slower or software implementation could be used instead. This is similar to the implementation of denormal floating point numbers in many microprocessors, which resort to software emulation of floating point operations when denormal numbers are seen.

The example program MORPHY [Pope96] illustrates one of the potential optimisations: the results of most operations fall within the orders of magnitude of $[\pm 2^0, \pm 2^{-4}]$. Knowing this allows the floating point unit to be simplified by reducing some of the most expensive parts – the barrel shifters required for operand alignment and post-operation normalisation. A shifter capable of 4-bit rather than 52-bit shifts may be used, reducing congestion and resource usage. Values outside this range may be aligned via multi-cycle shifting or software emulation, as they occur infrequently enough to prevent the penalty being significant.

3.2 Removal of Excessive Zero Values and Denormal Numbers

Our profiling of the SpecFP95 'mgrid' benchmark indicates a large number of zeroes or denormal values in the results. Calculating zeroes implies input of zeroes which is, in general, a waste of computing resources. Calculations with denormal numbers have an adverse effect on performance because typical processor designs implement it in software, whilst custom hardware designs do not implement it at all.

Identifying use of denormal numbers allows optimised hardware to be produced for such numbers, or the underlying code to be modified to avoid them, if possible.

3.3 Alternative Representations

A variety of alternative representations are available in custom hardware, including a wide range of floating point formats, fixed point and variations such as dual fixed point.

3.3.1 IEEE 32-bit float (vector instructions)

At its simplest level, a program may be converted from 64-bit to 32-bit floating point, providing possibilities for vectorisation by hand or with a vectorising compiler. FloatWatch provides information about the accumulated error should parts of the program be run in single precision.

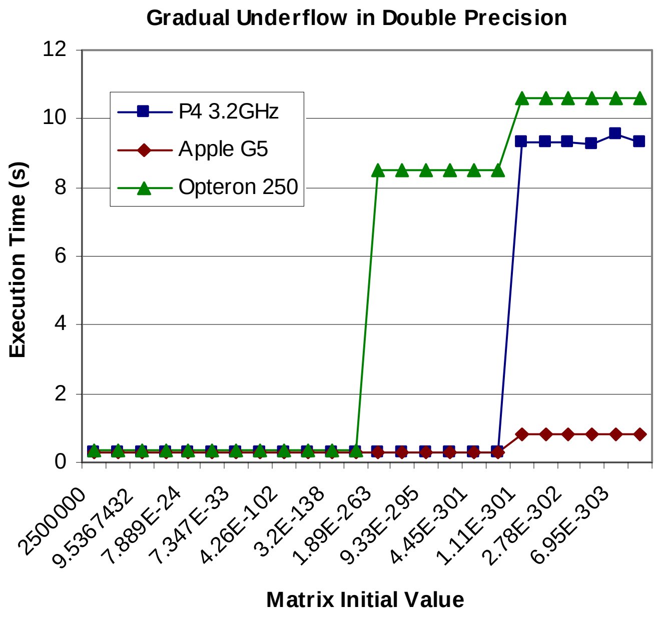


Figure 3: The Effect of Gradual Underflow. Convolution with a kernel containing progressively smaller values. Performance on Intel and AMD processors falls off dramatically as values enter the subnormal range. On the Opteron, performance begins to fall off even before this range.

3.3.2 Fixed point

In applications where a very narrow range is required, fixed point arithmetic may be appropriate. Using pure integer arithmetic provides a dramatic performance improvement or decrease in cost when compared to floating point. FloatWatch is unable to guarantee that values outside a particular range will not appear, so appropriate handling of exceptional cases is required.

Many of the test runs we have performed show symmetry around 0, allowing the sign of a fixed-point number to be represented in the standard way for a processor or custom design. For those with asymmetric value ranges a solution such as Dual Fixed-point may be suitable.

3.3.3 Dual fixed-point

Dual fixed-point (DFX) [Ewe04] is a variation on standard fixed point arithmetic, used where two distinct ranges of values must be represented. It occupies the middle ground between the flexibility of floating point and the efficiency of fixed-point.

Fixed-point, by definition, has the decimal point in a fixed place determined by the implementation. In DFX a single bit is reserved as selector, allowing one of two positions for the decimal point to be selected. Figure 4 compares 64-bit floating point against 64-bit DFX.

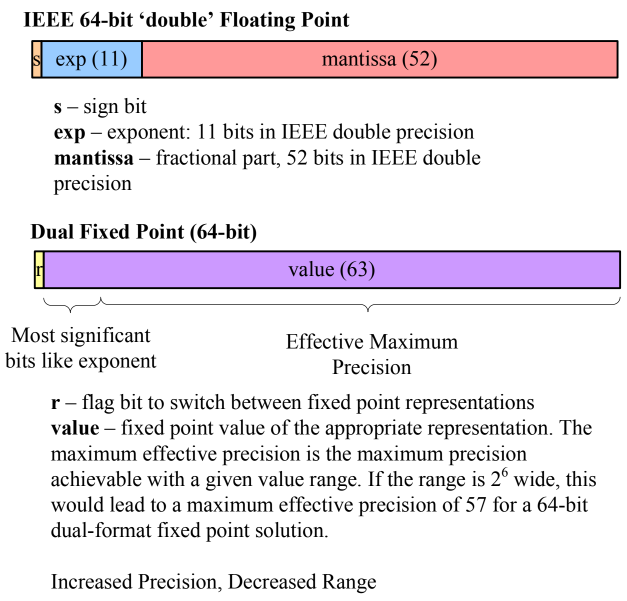

IEEE 64-bit 'double' Floating Point
s
exp (11)
mantissa (52)
s – sign bit
exp – exponent: 11 bits in IEEE double precision
mantissa – fractional part, 52 bits in IEEE double precision
Dual Fixed Point (64-bit)
r
value (63)
Most significant bits like exponent
Effective Maximum Precision
r – flag bit to switch between fixed point representations
value – fixed point value of the appropriate representation. The maximum effective precision is the maximum precision achievable with a given value range. If the range is 2^6 wide, this would lead to a maximum effective precision of 57 for a 64-bit dual-format fixed point solution.
Increased Precision, Decreased Range

Figure 4: Double precision floating-point and dual fixed point representations.

3.4 Dynamic Representations

Custom floating point units and a variety of alternative representations require a custom hardware implementation. The profiling results shown in the next section also reveal the possibility of dynamic representations, where the representation could change at each phase in the program or even for each line. Implementing a new ASIC for each possibility would be prohibitively expensive, consuming vast amounts of time and resources. Reconfigurable architectures provide a solution to this problem, allowing the generation of custom hardware designs at compile time, to be loaded in sequence at run-time.

One of the biggest advantages with reconfigurable architectures is the ability to generate a complete pipeline for a block of code, converting between representations within the pipeline itself to preserve the maximum accuracy possible. Different pipelines can be also be loaded onto a reconfigurable device as the program progresses, or as the execution context changes. The line-by-line refinement possible with FloatWatch allows this characteristic to be identified.

4 Results

We have profiled a sample of applications from different sources, including both "real-world" applications and benchmarks. This section describes the applications and their profiling results.

4.1 MORPHY

MORPHY [Pope96] 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 7 show 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 number of calculations performed varies widely 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 very narrow, indicating the possibility of a 64-bit fixed point or DFX implementation. Finally, the ranges are similar across the three datasets tested so far. Further work is being carried out to determine why this should be the case.

4.2 "ydl_pij" (Molecular Mechanics)

"ydl_pij" is an iterative solver for computational chemistry, using the Molecular Mechanics - Valence Bond [BM03] method. The code has many uses, including modelling magnetism and other electromagnetic properties and is currently being used in the Department of Chemistry at Imperial College. Figure 8 show the key sections of the graph of value ranges on a variety of datasets. The number of each test indicates the number of electrons.

This small graph does not adequately illustrate one of the key features of this application: while values concentrate in ranges that are not excessively wide, there are a large number of values spread across a much wider range. This makes them almost unnoticeable on a full-size graph, however this long tail indicates that a specialised implementation for a narrow range of values would not be appropriate - the application would spend a large proportion of its time executing with out-of-range data, which would most likely be implemented using a slow but cheap method.

Viewing the graph develop over time may reveal that the low-level wide range of values disappears after the first few iterations, allowing an alternative representation to be used later in the program.

4.3 SpecFP95 'mgrid'

The SpecFP95 'mgrid' benchmark is simplified multigrid solver which calculates a 3D potential field. As with MORPHY, it has two primary ranges. In this case the ranges are evenly spread with the exception of a spike at each end. A pronounced spike in the centre, indicating zero or denormal numbers points to possible unneccessary performance problems which could be reduced.

4.4 SpecFP95 'swim'

The SpecFP95 'swim' benchmark is a weather predictor based on shallow-water equations, using finite difference approximations. It is the only single precision benchmark in SpecFP95, however on x86 most calculations occur in double precision, with the result converted to single for storage only. It has several interesting features not seen on previous (pure double-precision) test applications.

The graph shows two primary ranges of results, one either side of the centre. A saw-tooth form is seen, with 4 "teeth", indicating four separate sub-ranges. Work is currently taking place to analyse this trend over time – it may be that each sub-range corresponds to a different iteration of the algorithm, in which case some dynamic modification of the representation used may be possible.

4.5 Summary of Results

Table 1 summarises the potential optimisation techniques which could be used on each application.

Application	Optimisation
MORPHY	Conversion to dual-fixed point format
ydl_pij	Few likely candidates – temporal profiling may reveal options
mgrid	Potential to remove zero or denormal numbers
swim	Phase-dependent customisation of floating point units/representation

Table 1: Optimisation Options for Floating Point Applications

5 Current and Future Work

FloatWatch is currently able to return useful results, however they can only act as an insight so far. Many additions are planned to improve the decisions which can be made on the results.

5.1 Improved Verification

The data generated by FloatWatch over multiple runs is only able to give an intuition about the general behaviour of a piece of code with multiple datasets. No verification is possible at present, so "fall-back" options must always be provided in the case that previously observed behaviour is not repeated with a new dataset.

One of the most useful possible extensions is to provide a verification framework, implementing techniques such as search-based testing to produce more rigourous results. The goal here is to look at real-world behaviour rather than theoretical behaviour, however any improvement in the confidence one can have in the results is desirable. The BitSize [Gaff04] tool provides similar functionality to this.

5.2 Extended Simulation

At present FloatWatch is only able to track the error for 32-bit vs 64-bit floating point, rather than for the vast array of floating- and fixed-point point formats available when creating custom hardware. A planned future extension is to allow plug-in modules for custom floating-point formats, providing a method of experimentation without resorting to RTL-simulation.

The proposed solution would allow the user to add a custom simulation object into the FloatWatch system, with results presented in the same way as at present. The option of testing several different custom formats at once would also prove useful.

Related to this is the ability to dynamically swap data formats as the profiled application progresses, simulating the possibility of dynamic reconfiguration on an FPGA for example.

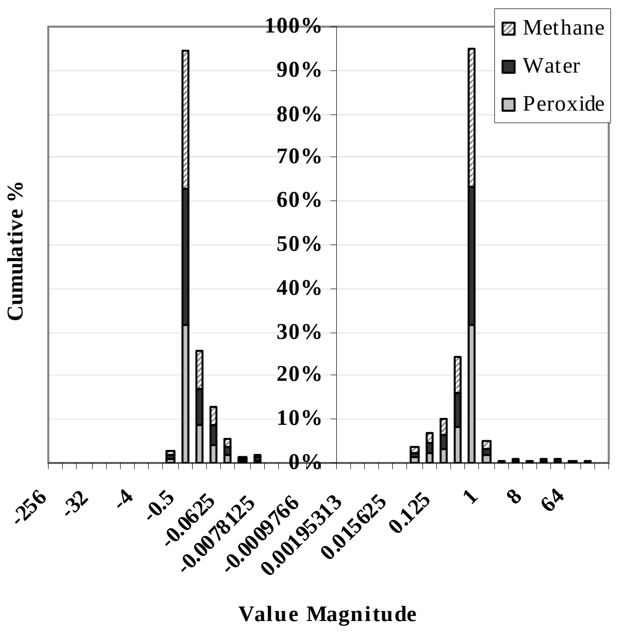


Figure 5: Profile Results for 'Morphy', showing core range. Values sporadically fall out of this range.

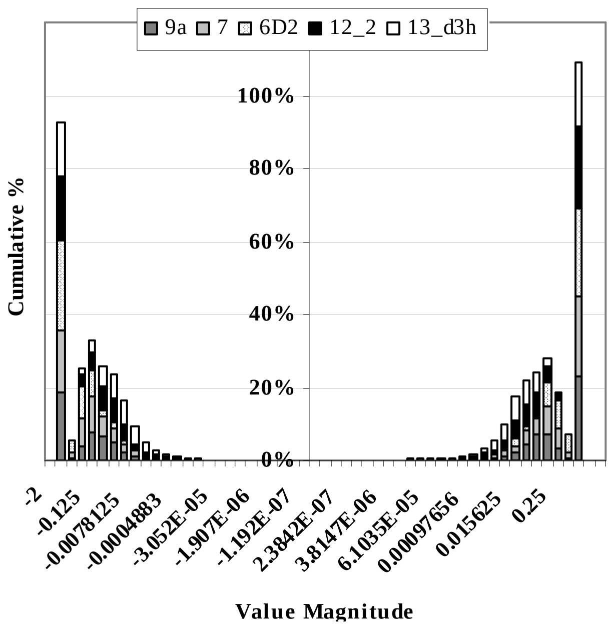


Figure 6: Profile results for MMVB code ("ydl_pij"). This graph represents the range with the highest concentration of values.

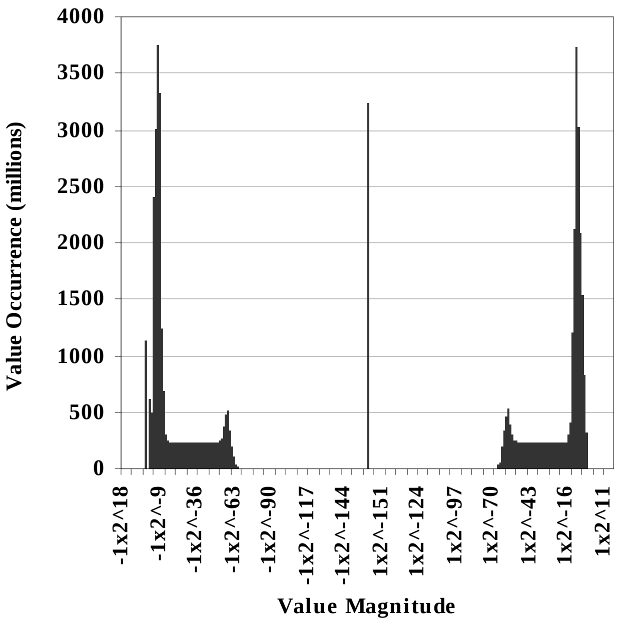


Figure 7: Profile Results for 'mgrid', showing the full range of values – the obvious key ranges at either side are very wide given the scale.

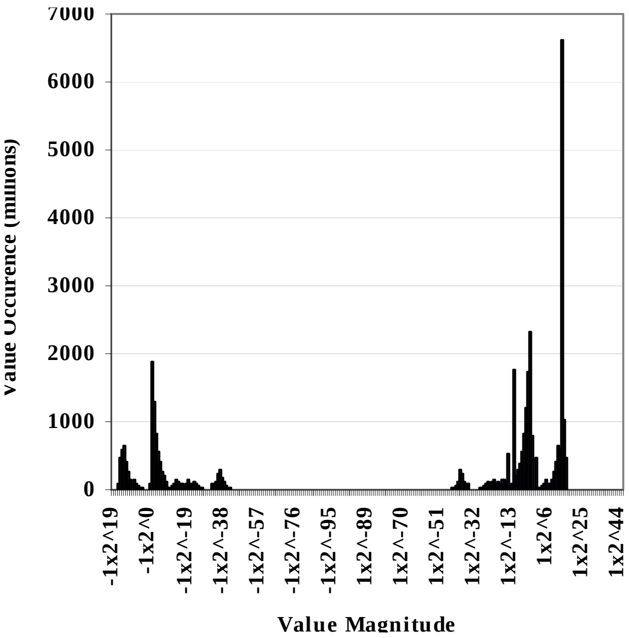


Figure 8: Profile Results for 'swim', showing the full range of values.

5.3 Using the Data

We intend to use the data gathered from our test runs to generate custom FPGA designs or GPU programs to accelerate our real-world applications and benchmarks. A number of further test runs are required, followed by an implementation of the most likely candidate applications.

6 Conclusion

The FloatWatch tool can provide a valuable insight into the floating point behaviour of a variety of scientific applications, regardless of implementation language. At present all implementation decisions based on the data generated by the tool would require a fallback, as no conclusive proof of the value ranges used is offered. A number of future enhancements will expand the capabilities of the system.

The behaviour of applications varies widely, with some using a very narrow range of values for all tested datasets, whilst others had a wide range of values, or narrow ranges which were dataset-dependent. All tests so far have revealed near-symmetric behaviour around zero, with approximately balanced numbers of positive and negative values. The FloatWatch tool has helped identify promising candidates for implementation with optimised floating-point formats, and various reconfigurable designs are currently being developed.

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.

[Ewe04] C. Ewe, P. Cheung, and G. 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.

[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.

[Hofs05] H. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society.

[Neth03] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[Pope96] P. Popelier. MORPHY, a program for an automated "atoms in molecules" analysis. *Computer Physics Communications*, 93:212–240, Februari 1996.

[pub05] *Automating Custom-Precision Function Evaluation for Embedded Processors*, 2005.

[Styl05] H. STYLES AND W. LUK. Compilation and Management of Phase-Optimized Reconfigurable Systems. In *Proc. International Conference on Field Programmable Logic*, pages 311–316, 2005.

[Weid04] J. WEIDENDORFER, M. KOWARSCHIK, AND C. TRINITIS. A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In *International Conference on Computational Science*, pages 440–447, 2004.