Generation Of Synthetic Floating-point Benchmark Circuits

Thomas C. P. Chau[1], Sam M. H. Ho[2] and Philip H.W. Leong[1]
[1]Department of Computer Science and Engineering,
[2]Department of Electronic Engineering,
The Chinese University of Hong Kong
{cpchau,phwl}@cse.cuhk.edu.hk mhho@ee.cuhk.edu.hk

Peter Zipf and Manfred Glesner
Institute of Microelectronic Systems,
Technische Universitt Darmstadt (TUD)
{zipf,glesner}@mes.tu-darmstadt.de

Abstract

Synthetic Floating-Point (SFP), a synthetic benchmark generator program for floating-point circuits is presented. SFP consists of two independent modules for characterisation and generation. The characterisation module extracts key dataflow statistics of an arbitrary software program. Generation involves producing randomised circuits with desired statistics which are either the output of the characterisation module or directly generated by the user. Using the basic linear algebra subprograms (BLAS) library, Whetstone benchmark and LINPACK benchmark, it is demonstrated that SFP can be used to generate floating-point benchmarks with different user-specified properties as well as benchmarks that mimic real computational programs.

1 Introduction

The continued improvement in component density of field-programmable gate arrays (FPGAs) devices has naturally led researchers in reconfigurable computing to study floating-point (FP) problems (e.g. [5, 7, 14, 17]). In order to better understand the applicability of existing FPGA devices to such problems as well as develop new domain-specific devices and CAD tools, representative FP benchmark circuits are necessary (e.g. [2]).

Most existing FPGA benchmarks, including the widely used MCNC benchmarks [16], were developed for logic circuits which are vastly different to FP circuits as, among other things, they have different operator logic, bus widths, logic depth and pipeline arrangements. This work aims to address a shortage of FP benchmark circuits available to the research community through a synthetic benchmark generation program which can produce unlimited numbers of FP circuits of given specification.

Synthetic circuit generation for logic applications has been reported in the literature. Hutton et al. [10] identified key physical properties of combinational circuits and generated randomised circuits with those properties. The technique was later extended to sequential circuits by adding flip-flops to the circuit model [9]. Wilton et al. [15] described a stochastic circuit generator that combines logic with memory to better model entire systems. Kundarewich et al. [13] improved on Hutton's work by introducing a hierarchy of circuits through clustering and employing iterative techniques in the generation process to provide better control over the generated circuit characteristics. Lemieux et al. [8] developed a new approach for generating semi-synthetic circuits which replaces part of a real circuit with a modified version of the same circuit. These benchmark circuits specifically target the testing of incremental place and route CAD tools.

In this paper we present a floating-point synthetic circuit generator, *synthetic floating point (SFP)*, which to the best of our knowledge is the first such reported program. Like previous approaches [9, 13], SFP is made from separate characterisation and circuit generator programs. However, rather than cloning existing circuits [8, 9, 13], a program's dataflow features are extracted from the assembly language output of a standard Fortran, C, or C++ compiler, allowing the creation of circuits which mimic software FP programs. In circuit generation, the synthetic circuit generator produces a randomised benchmark circuit in VHSIC hardware description language (VHDL) with specified statistical properties. Generation differs from previous work is that interconnections are word rather than bit-based, pipelined FP

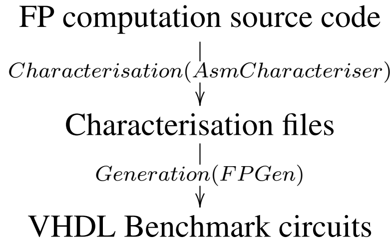

Figure 1. SFP synthetic circuit generation process.

operators rather than lookup tables are the computational primitives, and memory is supported.

The rest of this paper is organised as follows. Section 2 describes our characterisation work that extracts circuit characteristics. Section 3 describes the circuit generation process. Section 4 describes our results and analysis. Conclusions are drawn in Section 5 and possible avenues for future work are given in Section 6.

2 Characterisation

The process of generating benchmark circuits requires statistical information describing the desired properties of the output. In order to obtain this information, characterisation information is extracted from a software implementation of a similar computation. This process is illustrated in Figure 1.

To generate circuits having properties similar to computational problems we use actual floating-point computational software as a reference. Source code in almost any language including Fortran, C and C++ can be used as input to the characterisation program. The codes are first compiled to assembly language. Although a SPARC machine target was used in this work, any other RISC machine could be used. A program, *AsmCharacteriser*, takes assembly source codes as input, and outputs characterisation files describing circuit features.

AsmCharacteriser scans through the instructions, and then builds a graph resembling the dataflow of the computation. A node in the graph represents a floating-point operator, a register or a primary input/output (I/O) in the circuit, primary I/Os being unidirectional ports of a subcircuit. Nodes contain information about an operator's identity, type, operation performed, etc. and edges represent transfer of data between nodes.

A block is a code segment between two labels in the assembly language output, an example being given in Figure 2. Blocks in the source code are turned into *clusters*. A cluster is a subgraph that groups nodes and edges together and is similar to a subroutine in a program. Only multiplication, division, addition and subtraction operations are mapped from the assembly into floating point nodes. As an example, the block between labels .LL429 and .LL412 in Figure 2 is turned into the left-hand cluster of Figure 3. Information for building the circuit graph, such as the number of nodes contained, number of primary inputs, number of primary outputs, number of memory blocks, fanout statistics, etc. is stored within a cluster.

The entire graph models a *circuit*. It groups all the clusters together and captures general information concerning the circuit, such as total number of clusters, total number of nodes, etc.

The branch instructions in the source program are used to determine the interconnections of the clusters. Figure 3 illustrates three blocks represented as clusters and are interconnected based on the original program flow of the source code. In order to preserve the original properties of the source code dataflow, branching is handled in three ways,

- Blocks with no floating-point instructions are ignored. This is because such blocks are usually control instructions, and load/store instructions. We assume they do not contribute to the floating-point data path.
- If the block has floating-point instructions but no branch as the last instruction, it is assumed that the next floating-point block is connected to the current block, i.e. control flow proceeds to the next block. The case where a non-FP block with branch occurs between two FP blocks is neglected for simplicity.
- If a floating-point block has branch instructions to other blocks, connections are made between the blocks. This is analogous to broadcasting a computation result from one subcircuit to several other subcircuits.

In order to describe the characterisation process, several terms are first defined as in reference [9]. The *delay level* of a node is the maximum delay of the directed graph from a primary input to the node itself. *Shape* is the number of objects at each delay level and a *level node* contains all of the individual nodes at each delay level.

The characterisation file describes statistical and structural characteristics of the circuits, in which the parameters are grouped into six types [9].

1. Circuit Size and I/Os
   These characteristics describe the overall features of a circuit and include: number of nodes, number of edges, number of primary inputs, number of FP nodes, number of flip-flops, number of memories, number of clusters, and maximum delay. They are specified for each cluster as well as the overall circuit.

```
⋮

.LL429:
    ldd [%o0-16], %f16
    fabss    %f18, %f10
    fmovs    %f19, %f11
    fabss    %f16, %f8
    fmovs    %f17, %f9
    fcmped   %f8, %f10
    nop
    fbl  .LL133
    ldd [%fp-104], %f20
    fdivd    %f18, %f16, %f14
    ldd [%fp-32], %f30
    fmuld    %f30, %f14, %f28
```

```
    fmuld    %f20, %f14, %f24
    fmuld    %f18, %f14, %f26
    fsubd    %f20, %f28, %f12
    ldd [%fp-32], %f20
    faddd    %f26, %f16, %f10
    faddd    %f24, %f20, %f8
.LL412:
    fdivd    %f12, %f10, %f12
    fdivd    %f8, %f10, %f22
    fmovs    %f12, %f20
    fmovs    %f13, %f21
    std %f22, [%fp-32]
.LL135:

⋮
```

```
⋮
.LL133:
    fdivd    %f16, %f18, %f6
    ldd [%fp-32], %f4
    fmuld    %f20, %f6, %f10
    fmuld    %f16, %f6, %f2
    fmuld    %f4, %f6, %f0
    fsubd    %f10, %f4, %f12
    faddd    %f2, %f18, %f10
    b    .LL412
    faddd    %f0, %f20, %f8
.LL115:

⋮
```

Figure 2. A block is a segment of code between two labels. This example code segment is from 'ztrsm'.

Figure 3. Illustration of three blocks translated into clusters, and connected via registers.

2. Circuit Shape
   The shape of a circuit is defined as the number of objects at each FP node delay level. Node shape specifies the number of nodes at each delay level of a cluster. Input, output, latched node, primary output, adder, subtracter, multiplier, and divider shapes are defined in the same way. For example, the node shape of cluster 1 in Figure 3 is $\{4, 1, 3, 3\}$.

3. Edge Length Distribution
   The length of an edge between two nodes A and B is defined as the difference between the delay level of node A and that of node B. Edge length distribution is characterised in three ways: intra-cluster, inter-cluster input, and inter-cluster output edge length distributions. For example the intra-cluster edge length distribution of cluster 1 in Figure 3 is $\{3, 8, 3, 3\}$, representing 3 length 0 edges, 8 length 1 edges, 3 length 2 edges and 3 length 3 edges.

4. Inter-cluster Characterisation
   Two matrices are used to represent the number of connections between each and every cluster, one for edges connecting to FP nodes, another one for those connecting to registers. These are scalar values which are used in the randomised circuit generation process.

5. Fanout
   Fanout of a node is defined as the number of edges leaving the node itself. We characterise five types of fanout information: average and standard deviation of fanout of primary input, average and standard deviation of fanout of flip-flops, and fanout histogram. The

histogram is the number of nodes with a particular fanout number, e.g. $\{3, 4, 2, 2\}$ in cluster 1 of Figure 3 means there are 3 nodes with fanout 0, 4 nodes with fanout 1, 2 with fanout 2 and 2 with fanout 3.

6. Other Characteristics
   Other entries in the configuration file include: memory type, memory depth and bus width. These are described in more detail in the next section on circuit generation.

3 Circuit Generation

The synthetic floating-point benchmark circuit generation process is presented in this section. The input is a characterisation file which could be extracted from software via AsmCharacteriser or a similar program. Alternatively, it could be directly generated by the user. Output is a synthetic floating-point benchmark circuit in structural VHDL format. Although adaptable to other platforms, the output format presented is targeted for Xilinx Virtex 5 FPGAs.

Our random circuit generation program, *FPGen*, is based on the Cgen program of Kundarewich et al. [13] which was modified to satisfy the needs of floating-point benchmark generation. This required the changes described below.

3.1 Generation properties

1. Floating-point operators.
   Since the output is a floating-point circuit, the functional units are floating-point operators instead of LUTs used in traditional circuit generation for logic circuits. Xilinx Floating-point Operator cores [12] are instantiated to provide functions such as addition, subtraction, multiplication and division.

2. Memory blocks.
   The generation process is able to generate circuits that make use of memory blocks in the FPGA. Our target platform, Xilinx Virtex 5, provides two different types of memory: distributed and block SelectRAM. FPGen can use either. Memory properties, such as depth, width and registering of outputs are configured using the Xilinx CORE Generator [11].

   Distributed SelectRAM uses LUTs in FPGA slices as small, high bandwidth asynchronous memory distributed throughout the chip. As illustrated in Figure 4, a specific number of distributed memory blocks are assigned to each cluster, as defined in the characterisation file. Within each cluster, the memory block is connected with functional nodes through a bus.

   Block SelectRAM are dedicated blocks on the FPGA and hence provide memory without sacrificing logic

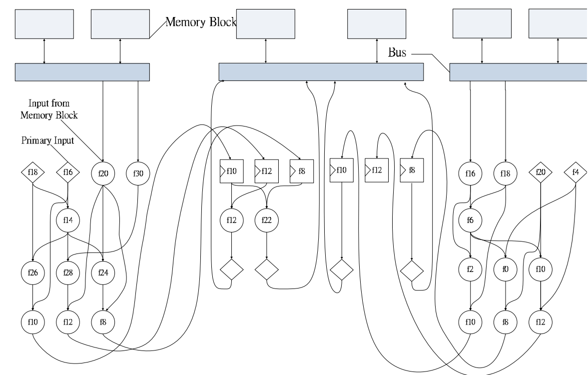

Figure 4. Distributed SelectRAM circuit model.

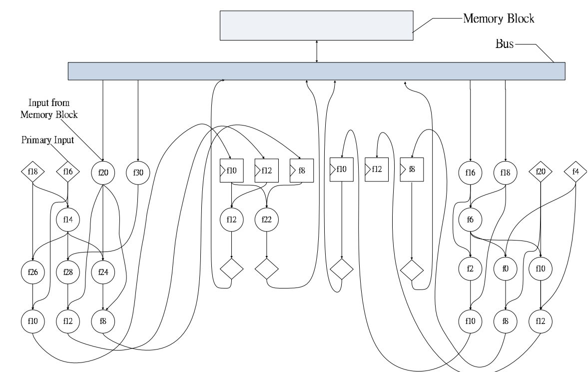

Figure 5. Block SelectRAM circuit model.

   resources. Figure 5 shows a centralised block SelectRAM connected to functional nodes via a common bus.

3. Datapath width.
   Output floating-point circuits use word-based connections where wordlength is user-specified to allow for different floating-point number formats. Both IEEE single precision and double precision formats are currently supported.

3.2 Generation process

The generation process used by FPGen involves four steps which are described below using the graph of Figure 3 (which in turn was extracted from the code in Figure 2) as an example.

In the first step of the generation process, individual nodes at each delay level are aggregated into level nodes, and edges are assigned to create a delay structure as illustrated in Figure 6.

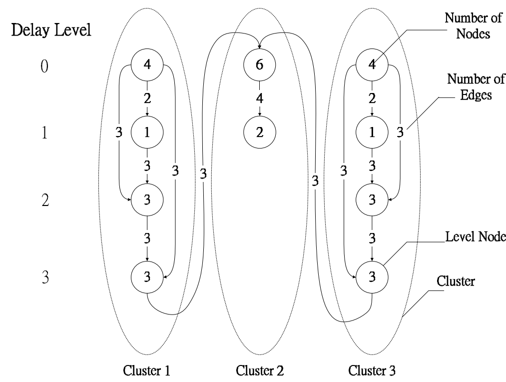

Figure 6. Delay structure of circuit in Figure 3 which cluster 1 has 4 primary inputs, 1 node at delay level 1, 3 nodes at delay level 2 and 3 nodes at delay level 3.

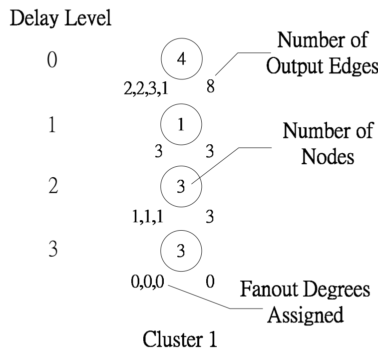

Figure 7. Assignment of fanout distribution to level nodes.

Secondly, the fanout distribution of each cluster, as specified in the characterisation file, is assigned among level nodes as shown for our example in Figure 7.

Thirdly, the level nodes are split into individual nodes, resulting in a pre-edge assignment structure. Each delay level has individual nodes with fanout assigned. Figure 8 shows the pre-edge assignment structure for the example.

Finally, edges are assigned between individual nodes to complete the circuit graph. Memory blocks are allocated to every cluster according to the memory model selected, resulting in Figure 9 which is the generated circuit for the dataflow extracted from Figure 2. This is essentially a randomised version of Figure 3 with memory.

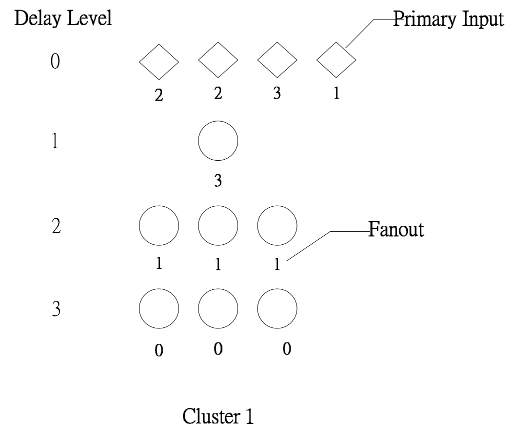

Figure 8. Pre-edge assignment structure.

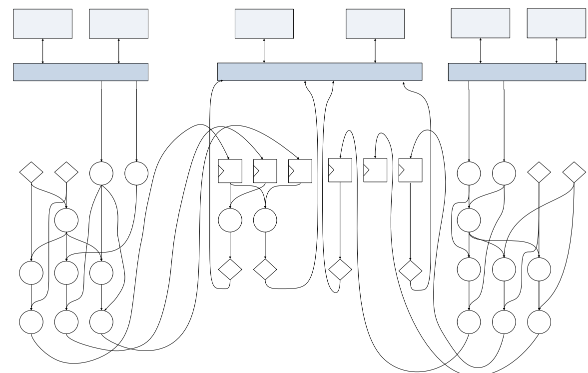

Figure 9. Final randomised output graph.

4 Results

The Basic Linear Algebra Subprograms (BLAS) [3] are an important library which is extensively used in high performance computing (HPC). First introduced in the early 70's, it provides standard building blocks for performing basic vector and matrix operations. A large number of HPC applications use BLAS, and all major vendors supply optimised BLAS routines for their machines. Since BLAS is one of the most extensively used FP libraries, it was chosen to test our approach. We do, however, note that the approach is general and can be applied to any other floating-point software.

The reference implementation of BLAS, in Fortran77, was compiled into SPARC assembly and served as the inputs to SFP. AsmCharacteriser and FPGen were used to generate circuits based on BLAS. We then synthesised the VHDL circuits to obtain the results reported.

BLAS library functions are grouped into three levels. Level 1 are vector-vector operations, level 2 are vector-

matrix operations, and level 3 are matrix-matrix operations [3]. We chose several functions from each level: 'axpy', 'rot' are from level 1; 'gemv', 'tpsv' and 'trmv' from level 2; and 'gemm', 'symm', 'trsm' and 'trmm' from level 3, as summarised in Table 1. In BLAS, each function is available in 4 different precisions: single (s), double (d), complex single (c) and complex double (z) and we refer to these as s-type, d-type, c-type and z-type.

In addition to BLAS, the Whetstone [4] and LINPACK [6] benchmark programs are also characterised. Whetstone benchmark is a common benchmark program that evaluates the floating-point arithmetic performance of computers. LINPACK benchmark is a program that measures system's performance in solving linear equations using Gaussian elimination with partial pivoting. Each benchmark program is a single Fortran source code file and thus the compiled assembly code contains all the floating-point instructions required for characterisation. Single and double precision versions of Whetstone and LINPACK benchmark programs are available from reference [1].

From BLAS library, nine functions at 4 precisions are available. Together with the single and double precision vesions of Whetstone and LINPACK, there are 40 different designs in total. Our results assume a default configuration of single precision (32-bit), 64 word memories and distributed RAM unless otherwise specified.

4.1 Characterisation

In this section we report on the characterisation information extracted from the BLAS library, Whetstone benchmark and LINPACK benchmark using AsmCharacteriser.

Table 2 summerises the parameters extracted from our targeted library and benchmarks, which show that a wide range of circuits can be characterised for benchmark generation. The size of circuit varies from 20 nodes to about 600 nodes, performing a number of different functions.

The relationship between I/Os and the size of circuit is shown in Figures 10 and 11. An approximately linear relationship between the number of I/Os and the number of nodes can be seen.

The node shapes of characterised BLAS functions for different levels of BLAS are presented in Figures 12, 13 and 14. We see that within each level, most of our samples, show similar shapes, zdot, csrot in Figure 12 being notable exceptions. Generally the shapes are decreasing from primary inputs, except zdrot and sgemv have peaks in the second delay level. Level 1 decreases in steps while level 2 and level 3 decrease smoothly. These figures show that shape can capture different properties of functions. Moreover, it highlights the similarity between different functions of the same BLAS level since similar shapes are observed.

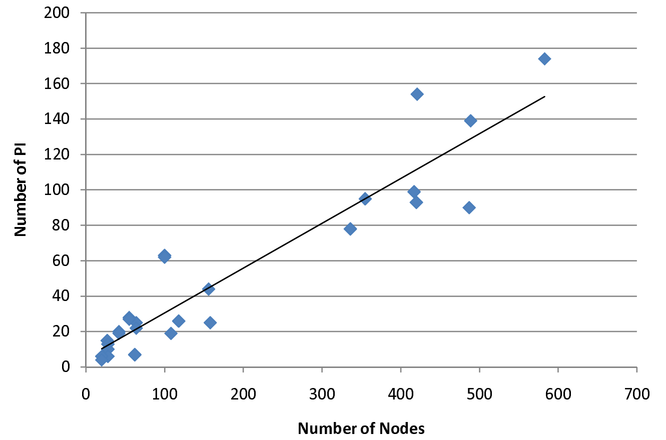

Figure 10. No. of primary inputs v.s. no. of nodes.

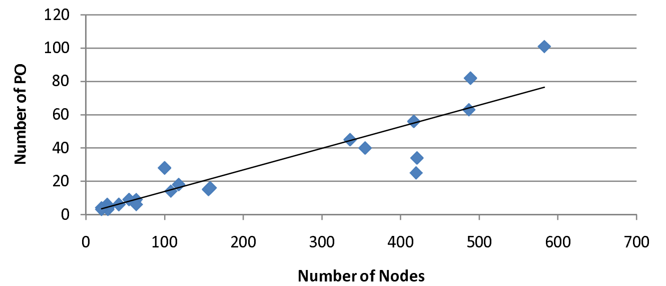

Figure 11. No. of primary outputs v.s. no. of nodes.

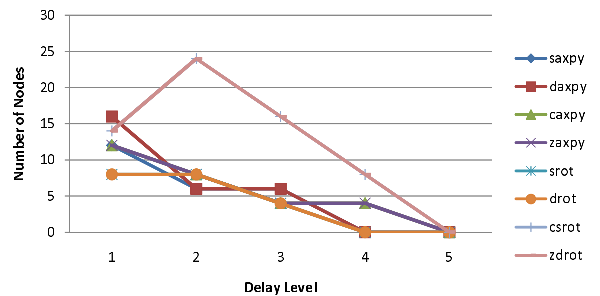

Figure 12. Node shape of BLAS level 1 subroutines.

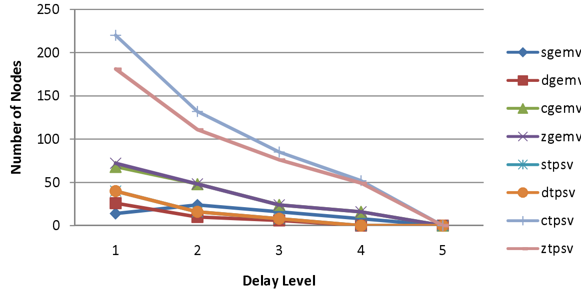

Figure 13. Node shape of BLAS level 2 subroutines.

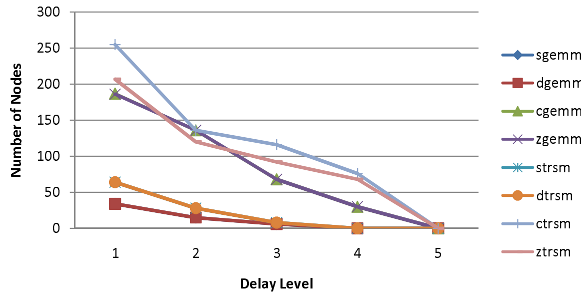

Figure 14. Node shape of BLAS level 3 subroutines.

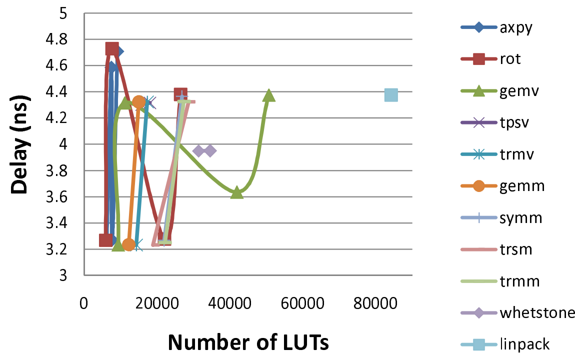

Figure 15. Delay v.s. no. of LUTs.

Table 1. BLAS functions used for synthesis

BLAS func.	Level	Description
axpy	1	calculate y = a*x + y
rot	1	apply Givens rotation
gemv	2	matrix vector multiply
tpsv	2	solving triangular packed matrix problems
trmv	2	triangular matrix vector multiply
gemm	3	matrix matrix multiply
symm	3	symmetric matrix matrix multiply
trsm	3	solving triangular matrix with multiple right hand sides
trmm	3	triangular matrix matrix multiply

4.2 Synthesis

In this section, synthesis results are presented. Circuits were generated using FPGen from the characterisation files of the tested BLAS subroutines. The targeted device for synthesis was a Xilinx Virtex 5 XC5LVX50T, which has 48 DSP48 blocks (each DSP48E contains a 25 x 18 multiplier, an adder, and an accumulator), 28800 LUTs and flip-flops. Xilinx ISE 9.1i was used throughout.

We attempted to implement the 40 different designs described earlier using single precision operators. The double precision versions of Whetstone and LINPACK, along with the c-type and z-type versions of some functions in BLAS level 2 and level 3, ran out of memory during synthesis and were left out of the study. Figure 15 shows delay against the lookup table (LUT) resource utilisation. As can be seen, a large range of design sizes can be created. There is no trend of delay varying with problem size because the floating-point operator cores are pipelined. It is also apparent that changing precision affects circuit size differently for different subroutines e.g. gemv's resource usage increased more than rot from real to complex. In general, the higher the complexity or level of the function, the larger the change in resource usage from real to complex e.g. axpy and rot are level 1, whereas gemv is level 2.

As mentioned earlier, the synthesis experiments were all conducted using single precision floating point operators. We further investigated the effect of using single precision operators for s-type circuits and double precision for d-type circuits. This is the closest approximation to real-world computations using these precisions. The double precision versions of the Whetstone and LINPACK benchmarks ran out of memory during synthesis. From Figure 16, we see that for d-type circuits, LUT usage increases by 200-300%, this being due to the difference in resource utilisation between single and double precision operators in our floating point library.

Table 2. Summary of the main AsmCharacteriser parameters extracted from the BLAS library and benchmark programs.

	No. of Nodes	No. of FP Nodes	No. of PI	No. of Edges	No. of Clusters	Max. Delay	Avg. Fanout PI (Std. D)	Avg. Fanout DFF (Std. D)
saxpy	27	6	15	24	3	2	1.20 (0.748)	(no DFFs)
daxpy	28	7	13	24	3	3	1 (0)	1.67 (0.943)
caxpy	28	12	10	32	2	3	1.60 (0.490)	2 (0)
zaxpy	28	13	6	32	2	4	1.67 (0.471)	1.67 (0.471)
srot	20	8	6	24	2	2	2 (0)	2 (0)
drot	20	9	4	24	2	3	2 (0)	2 (0)
csrot	62	40	7	96	2	3	3.43 (0.495)	3.43 (0.495)
zdrot	62	40	7	96	2	3	3.43 (0.495)	3.43 (0.495)
sgemv	42	10	20	32	10	3	1 (0)	1 (0)
dgemv	42	10	19	32	10	3	1 (0)	1 (0)
cgemv	156	73	44	176	12	4	1.70 (0.456)	1.54 (0.498)
zgemv	158	72	25	176	12	4	1.68 (0.466)	1.56 (0.497)
stpsv	64	15	25	48	16	3	1 (0)	1 (0)
dtpsv	64	18	22	48	16	3	1 (0)	1 (0)
ctpsv	489	187	139	538	52	4	1.84 (0.722)	1.49 (0.650)
ztpsv	417	180	99	472	40	5	1.85 (0.687)	1.71 (0.672)
sgemm	55	12	28	42	13	3	1 (0)	1 (0)
dgemm	55	12	27	42	13	3	1 (0)	1 (0)
cgemm	421	200	154	468	28	4	1.47 (0.499)	1.91 (0.378)
zgemm	420	209	93	468	28	4	1.47 (0.499)	1.65 (0.478)
strsm	100	8	63	72	28	2	1 (0)	1 (0)
dtrsm	100	8	62	72	28	2	1 (0)	1 (0)
ctrsm	583	227	174	656	62	4	1.68 (0.465)	1.57 (0.495)
ztrsm	487	217	90	560	46	4	1.73 (0.442)	1.71 (0.490)
whetstones	118	47	26	130	11	15	1.54 (0.887)	1.41 (0.782)
whetstoned	108	65	19	130	11	16	1.53 (0.993)	1.79 (1.41)
linpacks	355	167	95	334	27	17	1.07 (0.261)	1.06 (0.379)
linpackd	336	167	78	334	27	17	1.10 (0.303)	1.26 (0.551)

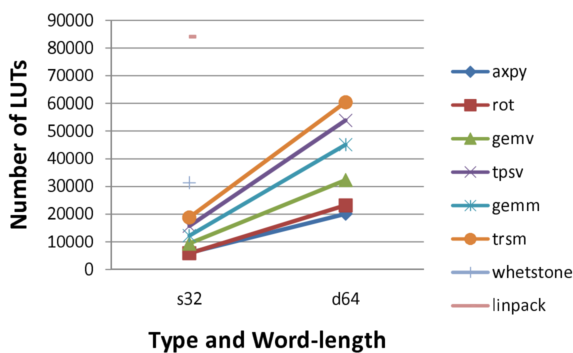

Figure 16. Resource utilisation for single and double precision circuits generated from BLAS.

5 Conclusion

In this paper, SFP, a new circuit characterisation and synthetic circuit generation tool for floating-point applications was presented. A software program to be mimicked is first compiled to assembly language and then statistical and dataflow information is extracted from it. A random circuit with similar structural properties and dataflow can then be generated. Another method of using this system is to directly specify the desired properties and generate a circuit from it without a characterisation step.

Experiments using the BLAS library, Whetstone benchmark, and LINPACK benchmark showed that a wide range of circuits with different properties can be produced. A benchmark set with 40 different designs ranging in size from 20-600 nodes, 6-227 floating point nodes, maximum

delay of 2-17 levels and fanout of 1-3.4 was created and most of the circuits of reasonable size synthesised. We hope that the tools and benchmarks can be used to compare different applications, high-level synthesis tools, CAD tools, floating-point libraries and FPGA architectures that use floating-point arithemetic.

6 Future Work

There are many possible avenues for further research in this area, particularly in developing more realistic dataflow and memory extraction techniques. In the future, we hope to implement better memory models supporting independently parameterised memory configurations and interconnection patterns as well as explore new ways to combine and split circuits to improve on their diversity. The current system is specific to the Xilinx FPGA family but we intend to add support for Altera and VPR [2]. As new ways to characterize floating-point real applications are developed, SFP can be modified to better mimic floating-point circuits.

Acknowledgement

The authors gratefully acknowledge the support of the Germany/Hong Kong Joint Research Scheme, grant G_HK007/07 – "Flexible Scheduling of Algorithms."

References

[1] Benchmark Programs and Reports, Netlib. *http://www.netlib.org/benchmark/*.

[2] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers Norwell, MA, USA, 1999.

[3] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw.*, 28(2):135–151, 2002.

[4] H. Curnow and B. A. Wichman. A Synthetic Benchmark. *The Computer Journal*, 19(1):43–49, 1976.

[5] M. deLorimier and A. DeHon. Floating-point sparse matrix-vector multiply for FPGAs. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 75–85, New York, NY, USA, 2005. ACM Press.

[6] J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM, 1979.

[7] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *Proc. FPGA Conference*, pages 86–95, 2005.

[8] D. Grant and G. Lemieux. Perturber: semi-synthetic circuit generation using ancestor control for testing incremental place and route. *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 189–196, 2006.

[9] M. Hutton, J. Rose, and D. Corneil. Automatic generation of synthetic sequential benchmark circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(8):928–940, 2002.

[10] M. Hutton, J. Rose, J. Grossman, and D. Corneil. Characterization and parameterized generation of synthetic combinational benchmark circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(10):985–996, 1998.

[11] X. Inc. CORE Generator System User Guide. *EDA tool documentation. San Jose, CA*, 1998.

[12] X. Inc. *Floating-point operator v4.0*, 2008.

[13] P. Kundarewich and J. Rose. Synthetic Circuit Generation Using Clustering and Iteration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(6):869, 2004.

[14] K. Underwood and K. Hemmert. Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 219–228, 2004.

[15] S. J. E. Wilton, J. Rose, and Z. Vranesic. Structural analysis and generation of synthetic digital circuits with memory. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(1):223–227, 2001.

[16] S. Yang. Logic synthesis and optimization benchmarks user guide version 3.0. *MCNC, Jan*, 1991.

[17] L. Zhou and V. Prasanna. High performance linear algebra operations on reconfigurable systems. In *Proc. SuperComputing Conference*, 2005.