APPRASE: Automatic parallelisation of Fortran to run on an FPGA

Brian Farrimond
Department of Computing,
Liverpool Hope University, UK,
farrimb@hope.ac.uk

John Collins
SimCon Ltd UK,
j.collins@simcon.uk.com
Department of Computing,
Liverpool Hope University,
collinj2@hope.ac.uk

Ashutosh Sharma
Department of Computing,
Liverpool Hope University, UK
sharmaa@hope.ac.uk

Keywords: FPGA programming, Fortran, Migration

Abstract

It is generally recognised that getting programs to run on a Field-Programmable Gate Array (FPGA) is not trivial. In spite of this, the positive attributes of FPGAs mean that there is currently much activity trying to unlock the opportunities they provide. This paper presents the APPRASE pipeline as a solution that simplifies the programming task in the context of migrating Fortran code to run on an FPGA. APPRASE is an acronym for the Automated Pipeline for the Parallelisation of RAndom Scalar Expressions. The APPRASE project is based at Liverpool Hope University UK

To overcome the difficulties of developing programs on an FPGA, the authors propose writing just one program. This program, referred to as the Hydra engine, is an interpreter which executes, in a highly parallel manner, the calculations implemented in the users original Fortran code transformed within the APPRASE pipeline.

The APPRASE pipeline begins with the analysis of the user's Fortran within the Fortran software engineering toolkit, WinFPT. WinFPT is able to transform automatically the Fortran code so that it is amenable to the next stages in the pipeline which generate highly parallel executable code that is interpreted and executed by the Hydra Engine.

A prototype is currently under development. A simulation has already been created which runs programs that consist of sums, products and conditional branches. Establishing the prototype will provide a framework for further development that can handle arrays, sub-program and function calls.

1. INTRODUCTION

This paper describes the APPRASE pipeline which aims to simplify the migration of Fortran codes onto FPGAs. The APPRASE project is hosted at Liverpool Hope University.
(http://www.hope.ac.uk/school-of-computing/apprase.html).

1.1. The potential of FPGAs

High Performance reconfigurable computing [Buell et al 1996] employing field-programmable gate arrays (FPGAs)[Trimberger 1994] has attracted considerable interest in the last few years.[Buell et al 2007] FPGAs are semiconductor devices that contain many thousands of logic blocks that can be programmed to connect together and perform the functions of simple logic such as AND and OR gates or more complex combinatorial logic such as adders, multipliers and decoders. An FPGA is described as reconfigurable since its logic blocks can be re-programmed indefinitely. Additionally, FPGAs include blocks of memory that are also reconfigurable. The potential of FPGAs lies in their inherently fine grain architecture provided by their configurable logic blocks [Wain et al., 2006]. Their highly parallel nature can be used for faster processing of hotspots found in complex algorithms. For example, the Maxwell Supercomputer in Edinburgh is essentially an IBM BladeCentre Cluster with FPGA acceleration. [Baxter et al 2007a] An important feature is that it has a much lower power consumption than equivalent machines with conventional architectures.

FPGAs have undergone rapid development in the past few years, in particular by increasing the number of logic blocks and size of memory. In fact, Moore's law is more closely followed by FPGAs than application-specific integrated circuits (ASICs) [Underwood 2004]. According to Mercury Computer Systems Inc., [Mercury Computer Systems Inc. 2008] FPGAs perform better than RISC processors in multiprocessing systems providing reduced overall system size, weight, cost and power consumption. Thus FPGAs are being used in various fields such as aerospace and defence, automotive, broadcast, consumer, data processing and storage, wired and wireless communications. [Xilinx Inc, 2008].

1.2. The problems associated with FPGAs

However, the task of writing new software for FPGA based machines is time consuming and expensive. [Baxter et al., 2007b]. The experience reported at Daresbury Laboratory is typical: *"Our initial experiences of FPGA programming have been largely frustrating. High performance*

reconfigurable computing is still very much in its infancy, with programming standards and portability between platforms still some way off. This results in a situation where a lot of time and effort can be spent writing software in a soon to be forgotten language for a soon to be forgotten machine" [Wain et al., 2006, p18]

Building and executing a program to run on an FPGA is a non-trivial task as it involves a multi-step process [Baxter et al., 2007b].

The hardware implementation process involves circuit designing, converting the code to a Hardware Description Language (HDL), synthesis, simulation, translation, placing and routing followed by on-chip verification and debugging. This process takes a long time and any mistakes during the process result in costly time delays. [Wain et al., 2006]

The programming overhead is mainly due to the steep learning curve of HDLs and I/O low level programming [Wain 2006]. Until recently projects had to be developed in close coordination with hardware/electronic engineers adding to the cost of human resources [Baxter et al 2007b]. The recent advent compilers for C like languages (such as Dime-C and Handel-C) for FPGAs has meant that most of the low-level programming can be abstracted for a software engineer. However, programming constraints such as working with double precision floating point numbers coupled with the absence of write-once/run-anywhere facilities, mean that it is still difficult to migrate large scientific programs onto FPGAs. This is especially true if the programs are in non-C languages such as Fortran as the cost of migration is high both in terms of time and money [Overbey et al., 2005]. The shrinking number of Fortran programmers who can refactor programs for High Performance Computing needs and the absence of cross platform standards for FPGAs make it more difficult to reach a wider audience of users (Baxter et al., 2007b).

1.3. The proposed solution

Recognising the difficulties of developing programs on an FPGA, the authors propose writing just one program which has been named as the Hydra Engine. This paper describes the logical design of the Engine and indicates the plans for its future development.

The paper is organised as follows. The next section describes the overall strategy of the pipeline. Section 3 explains how Hydra instructions to be executed by the Engine are generated from the user's original Fortran code. Section 4 describes the format of Hydra instructions. Section 5 describes the Hydra Engine operating as a program that fetches and executes Hydra instructions. Section 6 describes the execution of a typical Hydra instructions. Section 7 discusses implementation issues raised by the work done so far and charts future directions.

2. THE APPRASE PIPELINE STRATEGY

The APPRASE strategy is to maximise the potential for parallelisation of FPGAs. It does this by transforming the user's Fortran code into highly parallel Hydra instructions which can themselves be executed in parallel. Thus, like operations that can be executed simultaneously are transformed into a Hydra instruction consisting of an op code and an operand including multiple inputs and outputs. Suppose the FPGA is configured to contain 100 adders; then 100 simultaneous additions can be executed on the 100 adders using 200 inputs and obtaining 100 outputs. The addresses of the 200 inputs and space for 100 outputs form the operand part of the Hydra instruction. The second level of parallelisation is that unlike instructions can be executed in parallel. Thus, while the 100 additions are being executed, 100 multiplications and 100 divisions might also be executing if 100 multipliers and 100 dividers are also implemented on the FPGA.

The transformation of the user's Fortran starts with the WinFPT Fortran software engineering toolkit [WinFPT 2008] followed by the application of scheduling algorithms. These transformation steps are described in the following section.

3. TRANSFORMING FORTRAN INTO HYDRA INSTRUCTIONS

Tranformation is carried out by:

- using WinFPT to modify the Fortran to meet certain restrictions;
- using the f2Hydra WinFPT plug-in to translate the modified Fortran into untagged Hydra code;
- using DAREA scheduling algorithms to tag the Hydra code;
- using Hydra2Bin to translate the Hydra code into a binary format for execution on the Hydra Engine in the FPGA

3.1. Modifying the Fortran code with WinFPT

WinFPT is a suite of tools for writing, maintaining and migrating Fortran programs. It has several hundred commands for checking, reporting on and modifying Fortran source code. WinFPT is used to modify the users code to conform to the following restrictions:

Variables with only one (known) value are changed to DATA statements

For example

```
g = -9.81
```

is replaced by

```
DATA g /-9.81/
```

Similarly, variables which are modified in loops are also initialised in DATA statements. This enables the values to be planted directly into the Hydra program.

Single assignment to variables

All re-assignments to a variable are replaced with single assignments to new variables. Thus for example:

```
REAL*4 X
:
X = 7.7
:
X = A*B
```

is replaced by:

```
REAL*4 X_L1, X_L2
:
X_L1 = 7.7
:
X_l2 = A*B
```

The technique of making multiply assigned variables unique is based on the authors' development of the concept of symbol lives [Farrimond 2007] and it enables the optimisation algorithm to schedule the separate assignments independently.

Triplets only

All assignments involving expressions are reduced to triplets or separate function calls. Thus:

```
A = B*C + D*E*F + G*SQRT(H)
```

becomes

```
T1 = B*C
T2 = D*E
T3 = T2*F
T4 = SQRT(H)
T5 = G*T4
T6 = T1+T3
A = T6+T5
```

This enables parallelisation to be carried out at the finest granularity. Here, T1, T2, T4 can be calculated simultaneously. Later pipeline stages will identify and implement this level of parallelisation.

Replacement of loops with IF .. GOTO

All types of loop are replaced with the IF .. GOTO construct. Thus,. for example,

```
      DO I=1,10
         S = S+I
      ENDO
```

becomes

```
      I=1
100   CONTINUE
      S=S+I
      IF (I > 10) GOTO 200
      GOTO 100
200   CONTINUE
```

This simplifies the implementation of loops in the Hydra code because all control constructs can be handled in the same way.

Replacement of arguments

Sub-program arguments are replaced by copying operations to and from variables in COMMON blocks.

In-line expansion and loop unrolling

Small sub-programs are expanded in-line and small loops are unrolled.

3.2. Using f2Hydra to generate untagged Hydra

A WinFPT plug-in called f2Hydra is used to generate untagged Hydra from the modified Fortran. Here is the modified Fortran that computes numerically the trajectory of a projectile under gravity.

```
   DATA g /-9.81/
   DATA dt /1.0E-3/
   DATA v /100.0/      ! Initial condition v
   DATA h /0.0/        ! Initial condition h

100CONTINUE
! FPT  vf=v+g*dt       ! Next frame value v
   t1=g*dt
   vf=v+t1

! FPT  hf=h+v*dt       ! Next frame value h
   t2=v*dt
   hf=h+t2

   IF (h < 0.0) GOTO 200  ! Stop when we hit
                          ! the ground
   v=vf             ! Update integral for v
   h=hf             ! Update integral for h
   GOTO 100

200CONTINUE
   END
```

The corresponding generated untagged Hydra code is:

```
Initial
   g=-9.81E0
   dt=1.0E-3
   v=100.0E0
   h=0.0
End Initial

100
   r4_product   *   t1     = g * dt
   r4_sum       *   vf     = v + t1
   r4_product   *   t2     = v * dt
   r4_sum       *   hf     = h + t2
   r4_lt        *   h_lt   = h < r4_zero
   branch       *   h_lt, 300

   r4_sum       *   v      = vf
   r4_sum       *   h      = hf
   goto                    100
300
   halt         *
```

The Initial block identifies values to be planted directly into the binary version of the Hydra code. 100 and 300 are labels used in identifying the flow of control. The remaining lines are Hydra instructions in the following formats

<instruction type>	*	<triplet>
branch	*	<condition> <destination if true>
goto		<label>
halt	*	

The * symbols are placeholders for tags to be added by the DAREA scheduler in the next stage. The instruction types used here are:

r4_product	multiplication of two REAL*4 numbers
r4_sum	addition of two REAL*4 numbers
r4_lt	comparison of two REAL*4 numbers

Note that r4_zero symbolises zero stored as a REAL*4 number.

In

```
branch        *   h_lt, 300
```

the program branches to label 300 if h _lt is true, otherwise it goes to the next instruction.

What this stage has done is identify the types of operations to be carried out when executing the program. The next stage identifies the sequence in which they are to be carried out and opportunities for parallelisation.

3.3. Using the DAREA scheduler to generate tagged Hydra

The untagged Hydra is now processed by the DAREA scheduler which identifies the parallelism in the Hydra code and tags the operations with labels accordingly. Tagging does two things:

- replaces the * symbols with labels referred to as tags. Operations with the same tags can be executed simultaneously;
- identifies a linked list of tags that defines the sequence in which the operations can be carried out.

For the projectile example, the tagged Hydra looks like this:

```
Initial
   g=-9.81E0
   dt=1.0E-3
   v=100.0E0
   h=0.0
End Initial

100
   r4_product r4_prod_1    t1 = g * dt
   r4_sum     r4_sum_2     vf = v + t1
   r4_product r4_prod_1    t2 = v * dt
   r4_sum     r4_sum_2     hf = h + t2
   r4_lt      r4_lt_3      h_lt = h < r4_zero
   branch     branch_4     h, 300

   r4_sum     r4_sum_5     v = vf
   r4_sum     r4_sum_5     h = hf
   goto                    100

300
   halt       halt_6


link
   start            r4_prod_1
   r4_prod_1        r4_sum_2
   r4_sum_2         r4_lt_3
   r4_lt_3          branch_4
   branch_4         r4_sum_5, halt_6
   r4_sum_5         r4_prod_1
end link
```

Notice that several instructions have the same tags e.g. r4_prod_1 is the tag for both t1=g*dt and t2=v*dt. This means that the two multiplication instructions are executed simultaneously.

The link structure at the bottom shows the sequence in which the groups of instructions identified by tag names are to be executed. For each tag listed in the first column, the second column gives the next tag in the execution sequence. The branch_4 tag is a special case that gives two alternative next in sequence tags according to whether the branch test is true or false.

3.4. Using Hydra2Bin to generate the binary program for execution on the FPGA

The final stage is to use another process, Hydra2Bin to translate the tagged Hydra into Hydra instructions into binary format.

While developing the pipeline, we have employed an intermediate format that expresses the binary program in XML and developed a simulator that is able to read both XML and binary versions of a Hydra program and execute it in a pseudo-parallel manner. The XML aids in debugging the generation of the Hydra instructions.

We explain the structure of the binary instructions in the next section.

4. HYDRA INSTRUCTION FORMAT

There are two types of Hydra instruction:

- calculation instructions
- control instructions

Calculation instructions make use of the various adders, multipliers and other devices available within the Hydra

Engine. Control instructions direct the sequence of control such as branch and halt instructions.

4.1. Hydra calculation instructions

A Hydra calculation instruction consists of a single op code with multiple parallel input arguments, perhaps moderated by multiple parameters, and space for multiple parallel output results.

		Purpose
	header block	op code
		next address
		waitflag
		addressofinputs
		addressofoutputs
	parameter block	
		:
	input block	
		:
	output block	
		:

Table 1: Hydra calculation instruction format

The instruction is divided into four blocks:

header block

This contains the op code, the address of next instruction to be executed, a waitflag, and the addresses of input and output blocks.

parameter block

This contains values to be used to affect the operations on the input values e.g. the scale ratios in fixed format arithmetic

input block

This contains, for example, the addresses of pairs of values to be added or multiplied together according to the op code type

output block

This contains space for the results of the operations carried out on the inputs.

4.2. Customising the binary code generation

The numbers of parameters, input values and output values for a particular kind of calculation are determined when the Hydra Engine is built. The numbers will be determined by how many devices of each calculation instruction type are placed on the FPGA by the Engine build. The DAREA scheduler will need to know these numbers (which can be called collectively the Hydra Engine configuration) in order to add tags appropriately. Suppose, for example, that the Hydra Engine is given 100 adders. If DAREA spots that 120 additions can be executed simultaneously then it will create two Hydra add instructions to be executed in sequence: the first instruction will carry out 100 additions then the second instruction will carry out the remaining 20 additions.

5. THE HYDRA ENGINE

The Hydra Engine that will execute the Hydra instructions on the FPGA can be regarded as a fetch-execute engine.

Generally, only unlike instruction types can be executed in parallel. e.g. an adder instruction can execute on the adder array while a multiplier instruction simultaneously executes on the multiplier array. However, two adder instructions cannot use the adder array simultaneously.

In order to hold multiple fetched instructions, the Hydra Engine has multiple sets of Instruction Registers, Input address registers and Output address registers. The Hydra Engine is able to fetch an instruction op code and the instruction's waitflag value. If the waitflag value is 0, this means that the next instruction in sequence can be executed at the same time as the current instruction so the next instruction is fetched as well from the address provided in the next address part of the header block. Multiple sets of instruction registers and associated registers are made available to enable multiple fetches to be stored.

When a waitflag value of 1 is identified, fetching stops and the fetched instructions are executed in parallel. Once this is done, the next instruction is fetched.

5.1. Configuring the Hydra Engine FPGA program

As described in the previous section, the Hydra Engine will be configured to have particular numbers of adders, multipliers etc. It will also be configured to be able to fetch a particular number of instructions ready for parallel execution. We now discuss the two configurations.

5.2. Configuring the fetch

In order to be able to store numerous fetched instructions, a number of sets of registers are required. Each set consists of:

- an instruction register
- an input address register
- an output address register

- a memory address register

The fetch algorithm utilises a pointer to the current set of registers. After fetching an instruction, the pointer is incremented.

The number of sets of registers is fixed when the Hydra Engine is built. It is expected that there would be a trade off between increasing the number of sets to enable more instructions to execute in parallel and increasing the complexity of the circuitry. Experiments are required to find the optimum number of sets.

5.3. Configuring the execute

It is envisioned that the Hydra Engine will provide arrays of adders, multipliers, dividers, sine, cosine, exponential calculation devices etc. The number of parallel devices is fixed when the Hydra Engine is built.

5.4. Conceptual model of Hydra Engine

Figure 1 shows the logical structure of the Hydra Engine.

M is the memory in which the Hydra binary code is loaded.

PC is the program counter. It is initialised to point to the first instruction in the Hydra program.

On each fetch, it receives the address of the next address part of the fetched instruction. It may be overwritten by the execution of a branch instruction.

MARI is the memory address register used for fetching instructions.

WAITFLAG is used to determine whether the next instruction can be fetched immediately before the current instruction has been executed.

EXECUTECOUNT and IRCOUNT are used to control multiple fetches.

INPUTADDRESS(1:MAX), IR(1:MAX), MAR(1:MAX) and OUTPUTADDRESS (1:MAX)are the sets of registers to contain the headers of the fetched instructions.

The devices at the bottom named R4_SUM_DEVICE etc are the arrays of parallel calculation and control devices.

The address decoder represents the functionality to enable the registers to perform multiple accesses simultaneously.

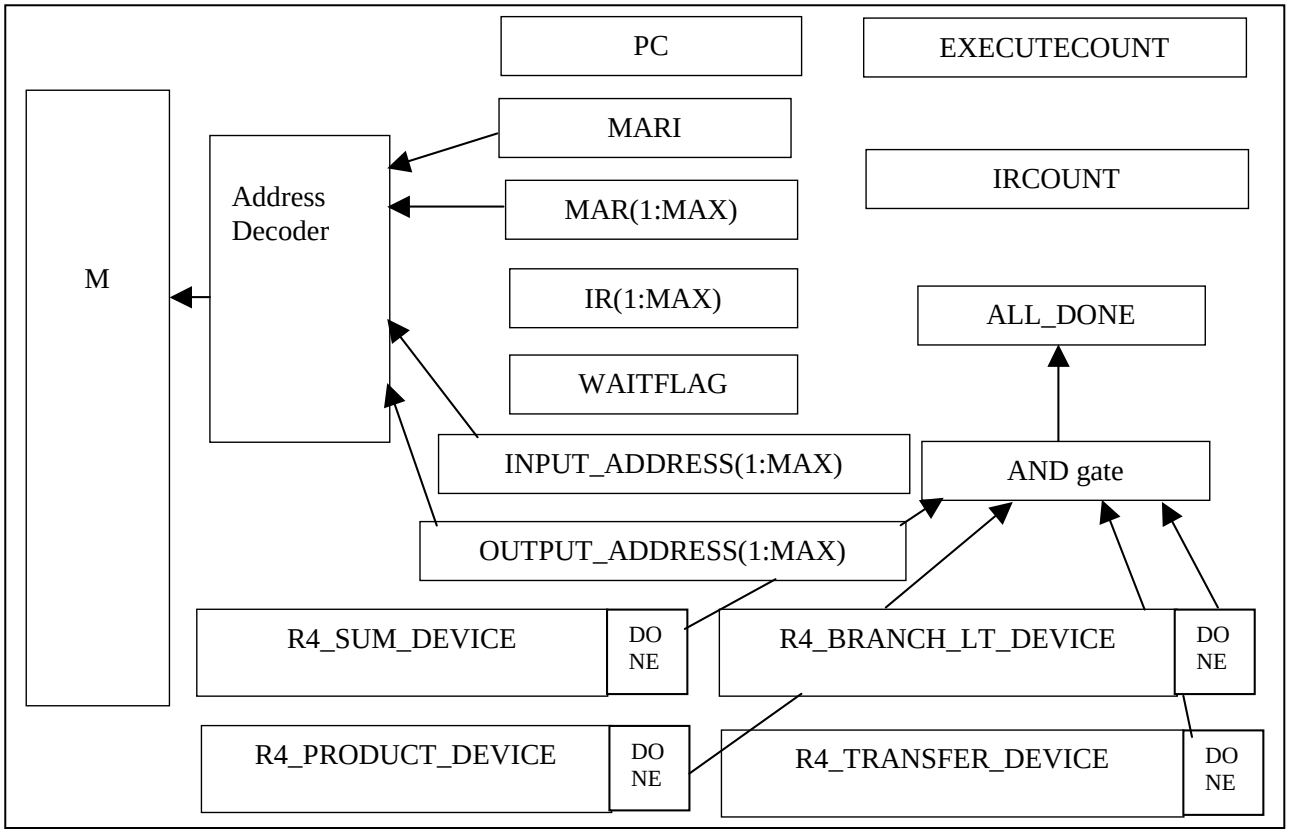

Figure 1: Hydra Engine logical design

6. EXECUTING A HYDRA INSTRUCTION

We shall now outline the execution of a typical Hydra calculation instruction R4_SUM. This instruction executes simultaneous floating point additions on multiple pairs of real numbers. Suppose their are 20 adders in the Hydra Engine.

The Engine will do the following:

- Send the value inside the INPUTADDRESS register for this instruction to the address decoder. The address decoder will connect the set of 40 memory locations starting at the address that was contained in the INPUTADDRESS register to the 20 pairs of inputs to the 20 adders.
- Send the value inside the OUTPUTADDRESS register for this instruction to the address decoder. The address decoder will connect the 20 outputs from the adders to the set of 20 memory locations beginning at the address that was contained in the OUTPUTADDRESS register.
- Trigger the addition in the 20 adders simultaneously. The results will appear in the output destinations simultaneously.

It will usually be the case that there are less than 20 valid additions to be performed in the instruction. This does not matter. Carrying out all 20 additions will not be slower than trying to identify which ones are valid and only executing those. Executing all 20 additions simplifies the complexity of the wiring in the Hydra Engine.

Executing products, divisions etc will be the same. Executing operations such as trig functions will be very similar - using one input per calculation rather than two.

7. CONCLUSIONS AND FUTURE WORK

Migrating Fortran programs of anything above a trivial size to run on an FPGA is very difficult and time consuming. This paper has described an approach to automation of the task that should make it much simpler and enable users to incorporate more of the inherent power of an FPGA in the execution of their programs. The difficulties in current means of programming FPGAs have led programmers to implement only key routines on the FPGA and leave the main program running on the host computer. This reduces the FPGA impact since the communications between FPGA code and the host code is invariably comparatively slow. The APPRASE approach makes it feasible to host the entire user's program on one or more FPGAs, this removing the communications bottleneck. The automated nature of the migration means that the process may take only a matter of minutes instead of the lengthy periods currently required. automation also removes many of the errors that are bound to be present in existing processes that are invariably heavily reliant on manual intervention.

Having established the logical design, developing it into a real working prototype is a major challenge. Nallatech [Nallatech 2008] is supporting our efforts in producing prototype implementations on its hardware platforms. the authors are confident that implementation of a prototype on a real FPGA can be achieved in the near future.

8. REFERENCES

Baxter, R. et al., 2007a. "Maxwell–a 64 FPGA Supercomputer". Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems. AHS-2007

Baxter, R. et al., 2007b. "High Performance Reconfigurable Computing – the view from Edinburgh." Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems. AHS-2007

Buell D., El-Ghazawi T., Gaj K., Kindratenko V., 2007, "Guest Editors' Introduction: High-Performance Reconfigurable Computing," Computer, vol. 40, no. 3, pp. 23-27, (March)

Buell D.A., Arnold J.M. and Kleinfelder W.J., 1996, eds, Splash 2: FPGAs in a Custom Computing Machine, IEEE CS Press.

Farrimond, B. and Collins J. Dimensional Inference Using Symbol Lives, 2007 International Conference on Software Engineering Theory and Practice (SETP07), Orlando, Florida

Mercury Computer Systems Inc. , 2008. "Innovation for Next-Generation Warfare." Available at: http://www.mc.com/uploadedFiles/Innovation-for-Next-Generation-Warfare.pdf .

Nallatech 2008 home web page: http://www.nallatech.com/

Overbey J., Xanthos S., Johnson R., Foote B., 2005. "Refactorings for Fortran and high-performance computing" Proceedings Of The Second International Workshop On Software Engineering For High Performance Computing System Applications p. 37-39, ACM

Trimberger S.M., 1994, ed, "Field-Programmable Gate Array Technology". Springer

Underwood K.D. 2004 "FPGAs vs. CPUs: trends in peak floating-point performance". Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays, FPGA 2004: 171-180

Wain R. et al., 2006, "An overview of FPGAs and FPGA programming; Initial experiences at Daresbury", Tech. rep.,

Computational Science and Engineering Department, CCLRC Daresbury Laboratory, November 2006.

WinFPT 2008 home web page: http://www.simcon.uk.com

Xilinx Inc, 2008. "FPGA and CPLD Solutions from Xilinx, Inc". Available at: http://www.xilinx.com/

Biographies

Brian Farrimond is Principal Lecturer in Computing at Liverpool Hope University, UK. Brian has an MA in Physics from Oxford University, MSc in Computer Science from University of Manchester UK. He is Honorary Research Associate at the University of Cape Town. Besides working at Liverpool Hope since 1992, he has worked as a consultant on a number of software engineering projects in the aerospace industry usually involving Fortran code migration. He has made major contributions to the development of the SimCon WinFPT toolkit (http://www.simcon.uk.com). In addition, Brian has developed the ChurchBuilder 3D Modelling tool for school children (http://www.kiddycad.com) and time map simulations of historical events (http://farrimond.no-ip.org/tmrg)

John Collins is technical director of SimCon Ltd. (http://www.simcon.uk.com), Associate Lecturer at Liverpool Hope University UK and an Honorary Research Associate at the University of Cape Town, attached to the Centre for High Performance Computing. He holds an MA in Physics and an MSc in Mathematical Psychology from Oxford University. He has worked in compiler design and software tool development since 1982, first as Software Development Manager at ADI Inc. in Ann Arbor Michigan (http://www.adi.com/), and later as Technical Director of SimCon. He is the principal author of the underlying engine which analyses and re-engineers Fortran code in WinFPT.

Ashutosh Sharma is Project Development Officer for the APPRASE project. He has worked on FPGAs for the APPRASE project and set up a High Performance Computing facility at Liverpool Hope University. He has an MSc degree in Distributed Systems from University of Liverpool U.K. and Bachelor of Information Technology degree (University of Delhi, India). Prior to working at Liverpool Hope University, he has worked for Sony Computer Entertainment Systems and Government of India's - Center for Development of Telematics. His research interests in addition to FPGAs and High Performance Computing include GSM mobile systems.