# Architectural Exploration of Reconfigurable Monte-Carlo Simulations using a High-Level Synthesis Approach

J.G.F. Coutinho
Imperial College London
Department of Computing
180 Queen's Gate
London SW7 2AZ, UK
jgfc@doc.ic.ac.uk

D.B. Thomas
Imperial College London
Department of Computing
180 Queen's Gate
London SW7 2AZ, UK
dt10@doc.ic.ac.uk

W. Luk
Imperial College London
Department of Computing
180 Queen's Gate
London SW7 2AZ, UK
wl@doc.ic.ac.uk

## ABSTRACT

This paper describes an approach for automatically generating programs that can be compiled into efficient hardware architectures, and illustrates its application in producing designs for Monte-Carlo simulation that are optimised for user requirements in resource utilisation or execution time. Monte-Carlo simulations are widely used in many financial applications and embedded systems, such as option pricing and portfolio evaluation. The intrinsic parallel nature of these applications, together with their inherent computational complexity, make them ideal candidates for acceleration using reconfigurable hardware. We use the Haydn approach to automatically derive simulation architectures from a C-like description, which describes the functionality of the design without focusing on hardware details such as timing. This approach can also exploit resource sharing within pipelined architectures, allowing the trade-offs between resource utilisation, parallelism, and execution time to be rapidly, safely, and automatically explored. To illustrate our approach, we present: (1) a model of the GARCH walk simulation in Haydn-C to perform design exploration, (2) a design-flow which supports interactive and batch modes for deriving these architectures, and (3) an evaluation of our approach targeting different CPU and FPGA devices. Our results show that an automatically generated Xilinx Virtex-II design operating at 180MHz is 36 times faster than a 3.2GHz Pentium 4, and 4.4 times faster than a 2.2GHz Quad Opteron system.

## Categories and Subject Descriptors

B.6.3 [**Design Aids**]: Automatic synthesis, Hardware description languages

## General Terms

Hardware Design, Design Exploration, Monte-Carlo simulations

## Keywords

FPGA, High-level synthesis, Pipelining, Resource sharing Design exploration, GARCH walk


*APGES 2007,* Oct. 4th 2007, Salzburg, Austria.


## 1. INTRODUCTION

Reconfigurable devices, such as FPGAs, are increasingly popular for implementing computationally-intensive applications, often for embedded systems and other computationally-intensive applications. The key advantage of reconfigurable technology is that they combine the performance of dedicated hardware with the flexibility of software, without the cost and risk associated with circuit fabrication. In particular, performance can be achieved by exploiting the application's inherent parallelism. Furthermore, reconfigurable devices can be reused many times over for implementing different hardware architectures, thus offering far more flexibility than ASIC solutions.

As reconfigurable technology makes progress in capacity and performance, there is an increasing need for high-level design methods and tools that can effectively address the growing complexity of hardware design to improve designer productivity. Such tools must enhance design maintainability and portability as system requirements evolve, and should facilitate design exploration so that various trade-offs, such as performance versus resource utilisation, can be examined.

To address these concerns, we are developing Haydn [5], a novel hardware compilation approach which offers designers a way to capture both cycle-accurate data-paths and high-level designs. Both manual and automated optimisation transforms can be used separately or in combination, so that one can achieve the best compromise between development time and design quality: some of our automatically generated designs are comparable in performance to hand crafted designs.

In this paper we illustrate how to exploit Haydn to describe and implement financial Monte-Carlo simulations in reconfigurable hardware. Monte-Carlo simulations are popular in financial applications, as they are able to value multi-dimensional options and portfolios without an exponential growth in run-time and memory use. Due to their high computational load and intrinsic parallelism, they are ideal candidates for acceleration using reconfigurable hardware. We automatically generate several architectures with different tradeoffs in resource utilisation and performance. The main contributions of this paper are:

- a parameterised design model of the GARCH random walk using the Haydn-C language, which allows developers to control the amount of resources and experiment with different timing configurations (Section 3).
- a fully automated hardware design-flow that operates in interactive and batch mode (Section 4).
- an evaluation of the proposed approach (Section 5). In particular, we automatically generate 57 architectures that target

five FPGA devices with different resource and timing configurations, and compare the performance against five CPUs. For instance, the fastest hardware architecture running on a Xilinx Virtex-2 at 180Mhz can run on-average 4.4 times faster than a Quad Opteron 275 running at 2.2Ghz, and 36 times faster than a Pentium 4 running at 3.2Ghz.

# 2. THE HAYDN APPROACH

This section covers our approach: using the Haydn-C language for high-level descriptions which can be used to automatically generate optimised hardware architectures that meet user requirements. The generation process is guided by resource and scheduling information, and designs can be transformed from high-level behavioural descriptions to low-level structural descriptions, all within Haydn-C. In the following, Sect. 2.1 first explains our motivation; Sect. 2.2 then describes our methodology. Sect. 2.3 covers hardware descriptions and their interpretation. Finally Sect. 2.4 provides an account of source-level transformation.

## 2.1 Motivation

Current design tools that target reconfigurable devices fall into two different camps, namely *behavioural* and *structural* approaches. Each has its own benefits and drawbacks.

The behavioural approach usually employs a hardware description language that is similar in syntax and semantics to popular software application languages, such as C. The goal of behavioural hardware compilers is to derive one or more hardware implementations from a high-level description (a process known as *high-level synthesis*), abstracting from low-level details such as timing and resource utilisation to allow developers to focus on algorithmic details. However, high-level synthesis often suffers from lack of user control and transparency over the implementation process. For instance, in the event of generating an unsuitable design, there is little a designer can do, except to experiment with behavioural constraints, or to manually alter the generated design.

On the other hand, structural languages, such as languages at the register-transfer level (RTL) and other cycle-accurate description languages give developers more control over low-level implementation. At this level of abstraction, developers are able to make decisions that are left to the compiler in the behavioural approach. In particular, developers are able to fine-tune their hardware implementations to achieve an optimal solution. However, the structural design methodology can have two major disadvantages over high-level synthesis, namely low productivity and poor maintainability, which make it highly ineffective for implementing large designs, and to perform design exploration. The lack of productivity is due to the fact that many implementation details and architectural decisions have to be provided at design time.

## 2.2 Methodology

To overcome the limitations of current design tools, we are developing Haydn, which combines the benefits of both behavioural and structural approaches. In particular, we have devised a source-level transformation process which can transform a behavioural-level design to multiple alternative structural-level designs, and from a structural-level design to either structural or behavioural-level descriptions. We can then exploit the strengths of each design level as follows:

- **Behavioural to Structural.** Developers can start the hardware design process with a simple behavioural description without providing architectural details such as timing and architecture-specific resources. Next, the high-level synthesis tool is used to derive an architecture with the aid of high-level annotations to guide the transformation process. This way, developers can rapidly obtain an optimised solution without worrying about low-level hardware details.

- **Structural to Structural.** Once developers have created the first implementation, they can systematically improve design performance or find the best tradeoff between resource area and execution time by modifying constraint parameters, running the source-level transformation process and verifying the performance of the generated design. Alternatively, they can manually modify and fine-tune their designs without computer intervention.

- **Structural to Behavioural.** Structural designs can be very difficult to understand. This is because the high-level design and data-flow becomes obscured by the implementation specific details of resource-binding and timing. In this case, a generic C description can be automatically produced without including low-level details, so that is easier to read, modify, verify and subsequently optimise.

## 2.3 Hardware Description and Interpretation

The Haydn-C hardware description language has been developed to support the above methodology. Haydn-C is based on the Handel-C language [4], but contains significant differences. It shares the same subset of ANSI C, such as assignment, conditional, and loop statements. Like Handel-C, it adds the `par` block to express explicit parallel computations, and flexible word-length sizes when declaring variables. However, unlike Handel-C, Haydn-C is a component-based language, like VHDL and Verilog: rather than declaring functions, developers write components with an explicit port interface. We believe that this feature makes it easier to import and export library blocks (such as IP cores), and work with other HDL tools. Haydn-C also supports other hardware constructs, such as the `pipe` and `wire` data structures, which carry data across pipeline stages and within a combinatorial cycle respectively. Furthermore, it also provides a source annotation facility to control the source-level transformation process, such as describing resource and schedule constraints, the type of transformation, and also the code to optimise.

There are two sets of interpretation rules applied to a Haydn-C description. The first, the *structural* interpretation is based on Handel-C timing semantics [4], and maps all operations in the design to a fixed time order. Because hardware synthesis and simulation processes adhere to the structural interpretation rules, developers are able to exert control over the quality of their designs. In other words, users can derive the schedule for a design, and change the schedule by revising the design. For instance, in Fig. 1**(c)**, when operating under the structural interpretation rules, the code is mapped to a fully pipelined schedule, where every operation executes concurrently, and completes execution in one clock cycle. This is because according to the structural interpretation rules all statements in a parallel block start execution simultaneously and each assignment statement executes in a single clock cycle. Note that we define the initiation interval (II) as the number of cycles required to output a result after the previous result. Hence, the design in Fig. 1**(c)** has an initiation interval of 1. If this code is restructured as shown in Fig. 1**(e)**, the new description is now mapped to a pipelined structure that shares an adder and produces a result every other cycle (II=2). In this case, operations in step 1 are executed in parallel, followed by operations in step 2. While we have saved one adder, we generate a result at half the rate of Fig. 1**(c)**.

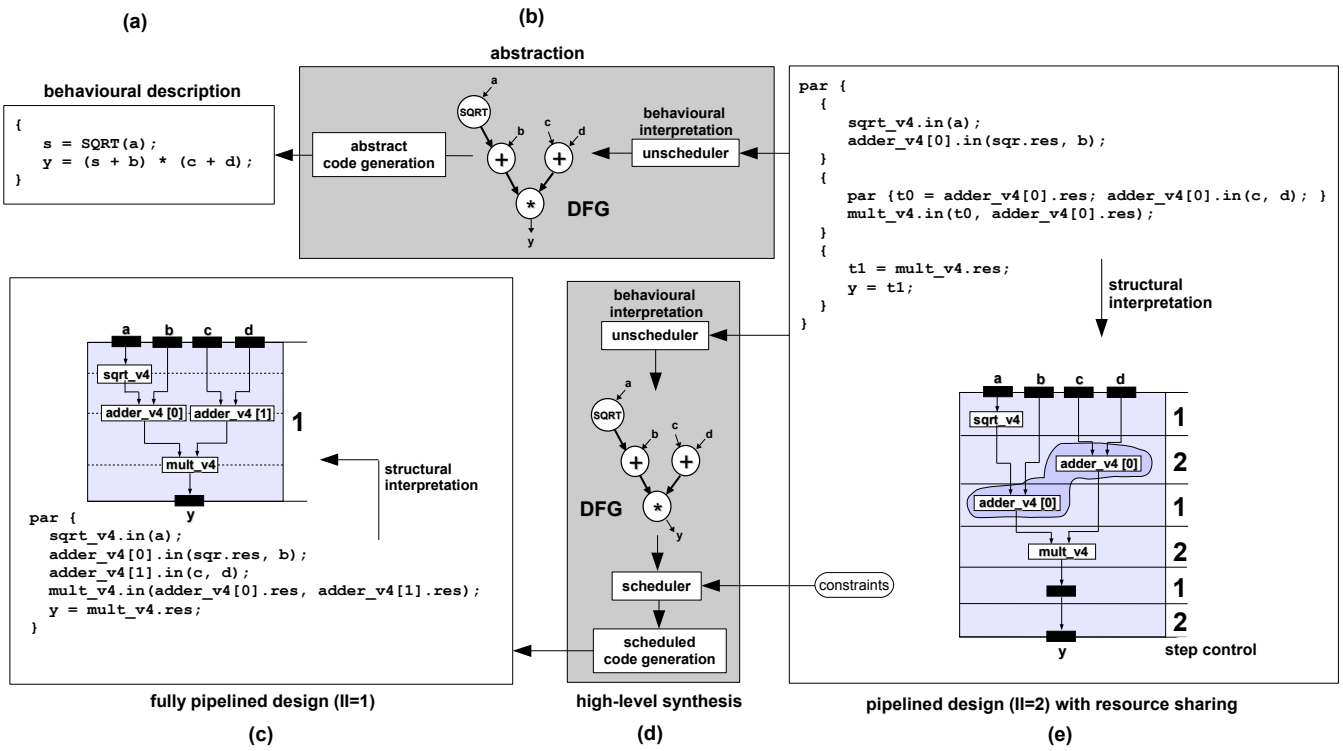


**Figure 1: Example of design modelling and source-level transformation using the Haydn approach.**

In order to perform source-level transformations, we relax the structural interpretation rules, which would otherwise be difficult to satisfy. In this case we follow the *behavioural* interpretation rules, which map a Haydn-C description into a dataflow graph (DFG). The DFG captures the behaviour of a design by representing program operations and their dependencies. Unlike the structural interpretation where operations are assigned to a fixed time-order, operations in a DFG are partially ordered by their dependencies. An example of a DFG is shown in Fig. 1**(b)**.

A Haydn-C description can reference operators and resources. Operators refer to abstract operations, whereas resources are associated to architectural implementations. In Fig. 1**(a)**, we use the `SQRT` operator to represent the square root operation without committing to a particular implementation, and thus, the design cannot be synthesised to hardware. In contrast, the code in Fig. 1**(c)** can be synthesised to hardware because it contains the resource reference `sqrt_v4`, which is a library block targeting the Virtex-4 device that implements the square operation. The key point is that developers have the option to use operators or resource references in their Haydn-C descriptions according to the abstraction level they wish to program.

In our approach, we define a Haydn-C description as *behavioural* (Fig. 1**(a)**) when the code is sequential (does not contain parallel blocks) and does not reference resources. On the other hand, the code is considered *structural* when no operators are used, and therefore the design can be synthesised to hardware (Fig. 1**(c)** and Fig. 1**(e)**). Note that resource and operators can share the same name. In this case, it is possible that a behavioural description can also be structural.

## 2.4 Source-Level Transformations

The source-level transformation process receives a Haydn-C description as input, and produces a new restructured Haydn-C design. There are two types of transformations: *high-level synthesis* (HLS) and *abstraction*. The HLS transformation (Fig. 1**(d)**) transforms a behavioural or structural Haydn-C description into a structural Haydn-C design. For instance, the code in Fig. 1**(a)** or Fig. 1**(e)** can be automatically restructured to produce the code shown in Fig. 1**(c)** when given the instruction to generate a pipelined architecture with II=1, and allocated enough resources to fully parallelise the design. Similarly, the HLS process can derive the code shown in Fig. 1**(e)** from either Fig. 1**(a)** or Fig. 1**(c)** when instructed to share an adder using a pipelined architecture with II=2.

Note that the functionality of the code shown in Fig. 1**(e)** is not completely obvious, and therefore it can be difficult to maintain if we wish to correct or update the algorithm. For this purpose, we use the abstraction transformation (Fig. 1**(b)**) to transform the structural design (Fig. 1**(e)**) into a behavioural description (Fig. 1**(a)**) in order to better expose its functionality. In particular, the abstraction process sequentialises the parallel program, and converts all resource references to operators.

To support HLS and abstraction, the source-level transformation process is composed by two modules: the unscheduler and the scheduler. The unscheduler module generates a dataflow graph from a Haydn-C description using the behavioural interpretation rules. The scheduler, on the other hand, performs the reverse: takes as input a dataflow graph, binds each operator to a resource, and places each resource in a time-order. The code generated reflects that new schedule. The HLS process (Fig. 1**(d)**) combines both unscheduler and scheduler modules. The abstraction process (Fig. 1**(b)**), on the other hand, relies only on the unscheduler module.

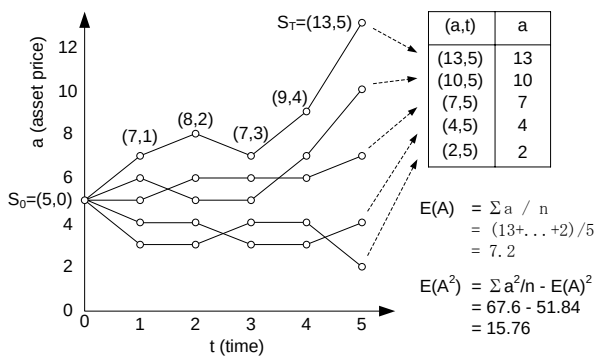


**Figure 2: Example of path based simulation.**

# 3. MODELLING

This section focuses on designing a GARCH asset path simulator in Haydn-C. In Section 3.1 we provide an overview of the Monte Carlo simulation framework, and Section 3.2 presents the parameterised behavioural description of the GARCH model.

## 3.1 Monte-Carlo Simulation Framework

To capture financial simulations such as option pricing and portfolio valuation, we use the framework presented in [10]. The types of simulations this framework addresses are those that analyse the aggregate properties of the paths of several independent random walks. In particular, the random walks we focus on start from a common starting state ($S_0$), then a stochastic transition function ($f : \mathbf{S} \mapsto \mathbf{S}$) is applied until a terminal condition is satisfied, ending the path.

An example of path based simulation is shown in Fig. 2. In this example, all paths start at the state $S_0$, but then diverge as time progresses along the horizontal axis, due to the stochastic nature of the state transition function. Once the termination condition of $t = 5$ is reached the current level of each path is extracted, allowing an estimate of the expected terminal path level to be made. Each individual path may produce a relatively inaccurate estimate of the true expected terminal level, but as more paths are combined the estimate will asymptotically approach the true expected terminal path level.

In this paper we focus on GARCH (Generalized Autoregressive Conditional Heteroskedasticity) asset paths. The GARCH model captures a relatively complicated class of random walk, used in advanced financial models to emulate the time-varying nature of volatility. This captures the memory of asset prices, where large changes in price are often followed by more large changes. In particular, the state and transition functions are defined as follows:

- $\mathbf{S} = \{(\sigma \in \Re^+, \epsilon \in \Re, v \in \Re^+, t \in \boldsymbol{N}_0)\}$
- $f(\sigma, \epsilon, v, t) = (\sigma', \epsilon', v + \mu + \epsilon', t + 1)$

  $\sigma' \leftarrow \sqrt{a_0 + a_1\epsilon^2 + a_2\sigma^2}$

  $\epsilon' \leftarrow \sigma'\mathbf{R}$

where $a0$, $a1$, $a2$ and $\mu$ are parameters to the simulation, and $t$ defines the time. $\mathbf{R}$ refers to a variable carrying a random value. The simulation results are then aggregated to produce a final result, which usually involves estimating the mean and variance of the relevant components of the path states. More details about GARCH walk and other random walk and option pricing path types can be found elsewhere [10].

```
1  design garch_walk {
2     in  bit 1 rng_load_enable;
3     in  bit 1 rng_load_data;

5     in  uint 12 in_data_t;
6     in  float 32 in_data_sigma;
7     in  float 32 in_data_eps;
8     in  float 32 in_data_v;

10    in  float 32 parm_a0;
11    in  float 32 parm_a1;
12    in  float 32 parm_a2;
13    in  float 32 parm_mu;

15    out uint 12 out_data_t;
16    out float 32 out_data_sigma;
17    out float 32 out_data_eps;
18    out float 32 out_data_v;

20    in  bit 1 in_valid;
21    out bit 1 out_valid;

23    code {
24       @HLS.run;

26       float 32 sigma;
27       float 32 eps;
28       float 32 v;
29       float 32 R;

31       RNG(rng_load_enable,
32           rng_load_data, R);
33       sqrt(parm_a0 + parm_a1
34            * in_data_eps
35            * in_data_eps + parm_a2
36            * in_data_sigma
37            * in_data_sigma, sigma);
38       eps = sigma * R;
39       out_data_v = in_data_v
40                    + parm_mu + eps;
41       out_data_sigma = sigma;
42       out_data_eps = eps;
43       out_data_t = in_data_t + 1;
44       out_valid = in_valid;
45    }
46 }
```

**Listing 1: Behavioural Haydn-C description of the GARCH walk algorithm. The annotation in line 24 specifies which code block is to be restructured by the high-level synthesis process (lines 23–45).**

## 3.2 Haydn-C Model

The behavioural Haydn-C description of the GARCH walk asset path (Section 3.1) is shown in Listing 1. The interface section (lines 2–21: LST 1) specifies the input and output ports, including the type and word-lengths of each port. The code section (lines 23–45: LST 1) describes the functionality of the design without focusing on low-level hardware details, such as how each operation is implemented or how they are placed in time-order. Also note that operators such as `RNG` (random-number generator) and `sqrt` (square-root) are used, as well as single-precision floating-point ports and variables. Apart from minor details, this code is similar to a software description.

In order to use operators and resources in Haydn-C, we must first define them (Listing 2). In lines 1–4: LST 2 we provide an example of operator definition, where the input and output ports are specified. Note that the type and word-length size of each port are optional. An example of resource specification is provided

```
1  operator sqrt {
2     in float 32 x;
3     out float 32 y;
4  }

6  resource rfsqrt(lat) [LATENCY: lat; OP:sqrt] {
7     in float 32 a;
8     in bit 1 clk <- __clock;
9     out float 32 result;

11    code {
12       // hardware/simulation code
13    }
14 }

16 rfsqrt(fsqrtA[2] ~ ipCoreA, 28);
17 rfsqrt(fsqrtB[3] ~ ipCoreB, 12);
```

**Listing 2: An example of operator and resource specification. In this case, we define the square-root operator, and instantiate five square-root resource units.**

```
1  // design parameters
2  set V2;           // FPGA device
3  set NRNG = 1;     // RNG resources
4  set NSQRT = 1;    // SQRT resources
5  set NADDERS = 4;  // ADDER resources
6  set NMULT =5;     // MULT resources
7  set cII = 1;      // initiation interval

9  // resource constraints
10 ifset ('V2') {
11    set family = "XilinxVirtexII";
12    set part = "XC2V6000FF1152-4"

14    dRNG(frng[NRNG] ~ NRG_v2, 0);
15    rfsqrt(fsqrt[NSQRT]~SQRT_v2,28);
16    rfadd(fadd[NADDERS]~ADD_v2,13);
17    rfmult(fmult[NMULT]~MULT_v2,6);
18 }
19 ifset ('V4') {
20    set family = "XilinxVirtex4";
21    set part = "xc4vsx55ff1148-12";

23    dRNG(frng[NRNG] ~ NRG_v4, 0);
24    rfsqrt(fsqrt[NSQRT]~SQRT_v4,28);
25    rfadd(fadd[NADDERS]~ADD_v4,16);
26    rfmult(fmult[NMULT]~MULT_v4,11);
27 }
28 ifset ('S3') {
29    set family = "XilinxSpartan3";
30    set part = "xc3s2000fg900-4";

32    dRNG(frng[NRNG] ~ NRG_s3, 0);
33    rfsqrt(fsqrt[NSQRT]~SQRT_s3,28);
34    rfadd(fadd[NADDERS]~ADD_s3,13);
35    rfmult(fmult[NMULT]~MULT_s3,8);
36 }

38 // time constraints
39 @HLS.options(II: cII);
```

**Listing 3: Parameterised section of the GARCH walk design, where constraints such as the number of resource units for each operator and the initiation interval can be specified.**

in lines 6–14: LST 2. In general, the resource specification provides the characterisation of a resource in terms of latency, initiation interval, slice utilisation, and the related operator. In line 6: LST 2, we define the `rfsqrt` resource with parameter `lat` which corresponds to its latency, and specify that it implements the `sqrt` operator. Furthermore, the resource specification also includes the port interface which must match the operator interface, and an optional code section (lines 11–13: LST 2) for hardware implementation and simulation. Once the resource specification is defined, we can instantiate one or more resources (lines 16–17: LST 2) by providing the resource instance name, the number of units and parameter values. For instance, in line 16: LST 2 we instantiate two square-root units with latency of 28 cycles, and associate them to a third-party library block `ipCoreA`. On the other hand, in line 17: LST 2 we instantiate three square-root units with latency of 12 cycles based on library block `ipCoreB`.

In order to derive architectures with different timing and resource configurations using the Haydn-C model shown in Listing 1, we first need to parameterise it. In lines 2–7: LST 3, we set the parameters in order to select the FPGA device we wish to target, the number of components to use for each resource type, and the initiation interval constraint. The library blocks are instantiated according to the selected FPGA device. Note that library blocks targeting different FPGA devices have different timing and resource configurations. For instance, the floating-point multiplier of a Virtex-2 has a latency of 6 cycles (line 17: LST 3), whereas the corresponding Virtex-4 version has a latency of 11 cycles (line 26: LST 3). The high-level synthesis process must know this information in order to generate an architecture with the right timings. Hence, to generate a different architecture, the user need only set the constraints in lines 2–7: LST 3 and run the high-level synthesis process.

# 4. DESIGN-FLOW

Once a design has been modelled and parameterised, application developers are able to automatically generate various architectures with different trade-offs in performance metrics, such as resource utilisation and execution time, by changing the information that guide the generation process. The interactive mode (Fig. 3) gives the developer full control over the design-flow. The design-flow is comprised of three processes: source-level transformation, hardware synthesis and hardware simulation. The developer can trigger them at any point of the design process. The source-level transformation process is guided and constrained by source-annotations, such as in line 39: LST 3 which specifies the initiation interval, and resource instantiations which generate annotations with information regarding the number of units and latency (lines 10–36: LST 3).

While the above annotations define the constraints for the architecture we wish to attain, the source-level transformation itself is triggered by the `@HLS.run` annotation (line 24: LST 1), which starts the high-level synthesis process. In general, the source-level transformation process scans for blocks of code that are enclosed by curly brackets and which contain annotations with requests for a particular action, such as high-level synthesis, abstraction, or loop transformations. In this case, the block is removed from the rest of the code, analysed and the new transformed code is placed instead (Fig. 3). Developers can immediately perform hardware synthesis with the new code block incorporated, or perform a cycle-accurate simulation of the hardware design in a software platform. Additionally, developers can manually revise the code or request another source-level transformation based on new constraints (lines 2–7: LST 3).

To aid the interactive mode, we developed a feedback tool that provides developers information about the high-level synthesis pro-

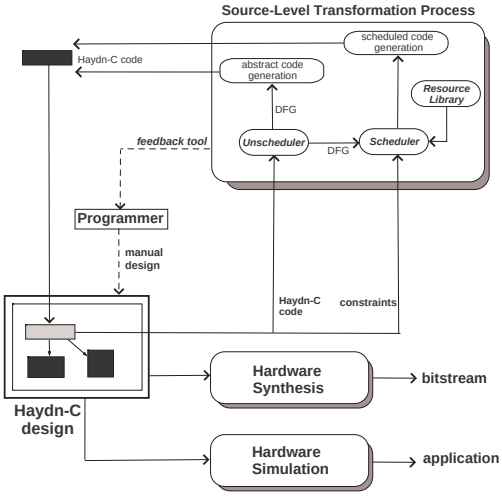


**Figure 3: Design-Flow in interactive mode.**

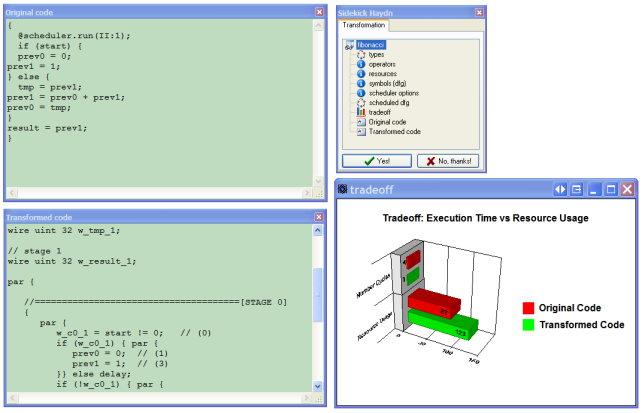


**Figure 4: Feedback tool which is invoked automatically after performing a source-level transformation in interactive mode. The two windows on the left show the old (top window) and new code respectively. The bottom right window shows a tradeoff estimate between resource usage and execution time, and the top right window enables developers to view other information about the design, including the DFG, symbols, operators and resources.**

cess (Fig. 4). Hence, developers have the opportunity to compare the old and new transformed code, analyse a chart showing an estimated tradeoff between resource usage and execution time, visualise the dataflow graph or scheduled pipelined map, and all other design attributes, such as all defined types, symbols, resources and operators. While the interactive mode provides more control and allows more hands-on experimentation when generating the architectures, it can be cumbersome if we wish to automatically generate several architectures. For this purpose, we configure the design flow to work in batch mode.

In batch mode (Fig. 5), the design flow automatically generates architectures based on the constraint meta-data file. Hence, rather than manually setting the design constraints (lines 2–7: LST 3) in interactive mode, we produce a file with the constraints for all architectures we wish to attain. For each architecture, we specify the number of resource to be allocated, the initiation interval, and the FPGA device we wish to target. To generate the meta-data file, we use the following formula [9] to compute the number of resources to use for each operator based on a particular initiation interval ($II$):

$$r(op, II) = \left\lceil \frac{n(op)}{II} \right\rceil$$

where $n(op)$ corresponds to the number of operators in the design. For instance, if a design has five multipliers, then for $II = 1$ we use five multiplier resources, for $II = 2$ we use three, and for $II = 5$ we use only one. This way, designs with a higher initiation interval should use less resources than those with lower initiation intervals, albeit running slower.

Once the meta-data file has been generated, we automatically feed the parameters for a particular architecture to the Haydn-C model to instantiate the resources with the appropriate number of units, and other annotations to guide the high-level synthesis process. Once the high-level synthesis process generates a new architecture in Haydn-C, we proceed with logic synthesis to produce the netlist representation of the kernel. We also produce in parallel the netlist description of the top-level design, which is responsible for instantiating the platform interface, global logic and the GARCH walk kernel. The place-and-route process combines both netlist representations plus any library block referenced by the kernel and top-level design, such as floating-point and random number generators, producing the final bitstream file.

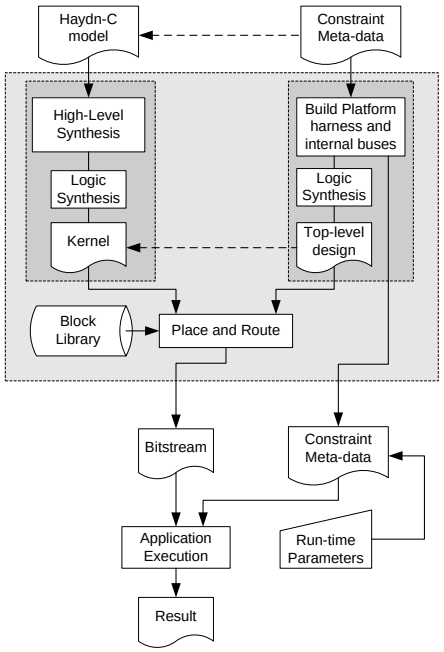


**Figure 5: Design-Flow in batch mode. The two dark gray boxes generate a netlist representation of the kernel and the top-level design (logic synthesis). The light gray box performs hardware synthesis starting with a Haydn-C description and producing a bitstream file to configure the FPGA.**

**Table 1: Constraints used to derive all GARCH walk kernels.**

| Contraint | Values | Description |
|---|---|---|
| FPGA family | V2, V4, S3 | Virtex2, Virtex4 or Spartan-3 |
| Number of cores | 1..2 | Use single or duplicated kernel |
| II | 1..10 | Initiation Interval |
| NADDER | 1..8 | Number of addition resources |
| NMULT | 1..10 | Number of mult. resources |
| NSQRT | 1..2 | Number of square-root resources |
| NRNG | 1..2 | Number of RNG resources |

Associated with this bitstream file is the meta-data file which explains to the software host which kernels are contained within the bitstream file, and how the kernel parameters are mapped onto the global bus. At runtime a software component can use the meta-data to find and load the correct bitstream file, set the unbound simulation parameters, initialise all the random number generator seeds, and then execute the simulation paths.

# 5. EVALUATION

To evaluate our approach, we use the design-flow in batch mode, as explained in Section 4, in order to automatically generate architectures with different timing and resource configurations. In addition to the Haydn-C model shown in Listing 1, we use a two core model where we duplicate the number of operators and the number of ports to provide more opportunities for resource sharing. Table 1 enumerates the constraints and range of values used to automatically generate 57 FPGA architectures. Note that not every combination of constraint values result in a synthesisable architecture. For instance, they can generate an over-mapped architecture where the number of slices exceed the number available in the FPGA device.

To complete hardware synthesis, we convert the Haydn-C model into a Handel-C description, and then use the DK4.1 compiler for logic synthesis. Floating-point cores from CoreGen 7.1 are adopted for all mathematical operations using the IEEE single precision floating point format. Random number generators are implemented using linear piecewise approximation. The resulting components are then placed and routed using ISE 7.1 with the default optimisation options. The FPGA devices chosen belong to the Virtex-2 (xc2v2000 and xc2v6000), Virtex-4 (xc4vlx40 and xc4vlx100) and Spartan-3 (xc3s2000) families.

We evaluate performance using MStep/s, a measure of the number of times the state transition function can be executed by a device per second. If a design contains $i$ simulation kernels, and can execute at $f$ MHz, then it can achieve $i \times f$ MSteps/s. Although this is a measure of peak sustained performance, it is close to achieved performance since the only overhead unaccounted for is that introduced by parameter setup and the retrieval of results. This overhead will be of the order of 100-1000 cycles for every 10M-1000M cycles of path generation, so the impact will be minimal.

Fig. 6 compares design performance against resource utilisation for architectures targeting the Virtex-2 and Virtex-4, as the initiation interval varies. As expected, architectures with single-cores use less resources than those with two cores. Furthermore, designs with a higher initiation interval occupy less resources since resource sharing is higher, although they perform poorly. Not all architectures provide the best tradeoff between performance and resource usage. For instance, the two core version xcv2000 with II=10 occupies more resources than the one core version xc4vlx100 with II=5 while performing slower. Furthermore, the one core version xc4vlx100 with II=1 is not the fastest, but it does provide the best tradeoff in resource utilisation and performance. Given the number of architectures generated automatically, it is now possible to quickly find the fastest design up to a given area, or the smallest design that can sustain a certain clock frequency. A comparison of the fastest and smallest design between architectures with the same FPGA device and core number is shown in Table. 2. The numbers in bold correspond to the overall fastest and smallest design.

**Table 2: Comparison in performance and size between the fastest and smallest design with the same device and core number. The number of cores is shown in parenthesis.**

| Device | Resources (slices) | Performance (Msteps/s) | Fastest vs Smallest |
|---|---|---|---|
| xc2v2000 (c1) | 5111 | 183 | 2.3x faster |
| xc2v2000 (c1) | 3279 | 77 | 1.5x smaller |
| xc2v2000 (c2) | 8609 | **363** | 11.4x faster |
| xc2v2000 (c2) | 4223 | 32 | 2.0x smaller |
| xc3s2000 (c1) | 5632 | 134 | 6.0x faster |
| xc3s2000 (c1) | 3256 | 22 | 1.7x smaller |
| xc3s2000 (c2) | 6484 | 86 | 2.0x faster |
| xc3s2000 (c2) | 4378 | 42 | 1.5x smaller |
| xc2v6000 (c1) | 4910 | 182 | 6.2x faster |
| xc2v6000 (c1) | 3238 | 29 | 1.5x smaller |
| xc2v6000 (c2) | 9538 | 328 | 10.9x faster |
| xc2v6000 (c2) | 4610 | 30 | 2.0x smaller |
| xc4vlx40 (c1) | 4833 | 272 | 5.5x faster |
| xc4vlx40 (c1) | 3432 | 49 | 1.4x smaller |
| xc4vlx40 (c2) | 5914 | 225 | 3.7x faster |
| xc4vlx40 (c2) | 4240 | 60 | 1.3x smaller |
| xc4vlx100 (c1) | 4269 | 293 | 6.0x faster |
| xc4vlx100 (c1) | **3112** | 48 | 1.3x smaller |
| xc4vlx100 (c2) | 6779 | 198 | 3.4x faster |
| xc4vlx100 (c2) | 4434 | 57 | 1.5x smaller |

**Table 3: Performance comparison between software processors and the fastest derived FPGA design.**

| Family | Max. Freq (Mhz) | Performance (Msteps/s) | Speedup |
|---|---|---|---|
| Pentium 4 | 2000 | 9.8 | 1x faster |
| Pentium 4 Xeon | 2600 | 11.3 | 1.14x faster |
| Pentium 4 | 3200 | 20.8 | 2.1x faster |
| Athlon MP | 2100 | 21.1 | 2.1x faster |
| Quad Opteron 275 | 2200 | 81.5 | 8.2x faster |
| FPGA: xc2v2000 (c2) | 180 | 362.7 | 36.7x faster |

Table 3 compares the performance of the fastest hardware implementation against several general-purpose processors. The software is developed in C++, and compiled using the Visual Studio 2005 compiler with all optimisations turned on, and all floating-point transformations allowed. The random numbers are generated using a combination of the Mersenne-Twister URNG to provide random integers [8], and the Ziggurat method to convert the random uniform integers into Gaussian random numbers [7]. The results show that the two core version of xc2v2000 device at 180Mhz can run on-average 4.4 times faster than a Quad Opteron 275 running at 2.2Ghz, and 36 times faster than a Pentium 4 running at 3.2Ghz.

# 6. RELATED WORK

There are many examples of behavioural and structural approaches to compiling high-level algorithm specifications into reconfigurable hardware in both industry and academia. Examples of behavioural-

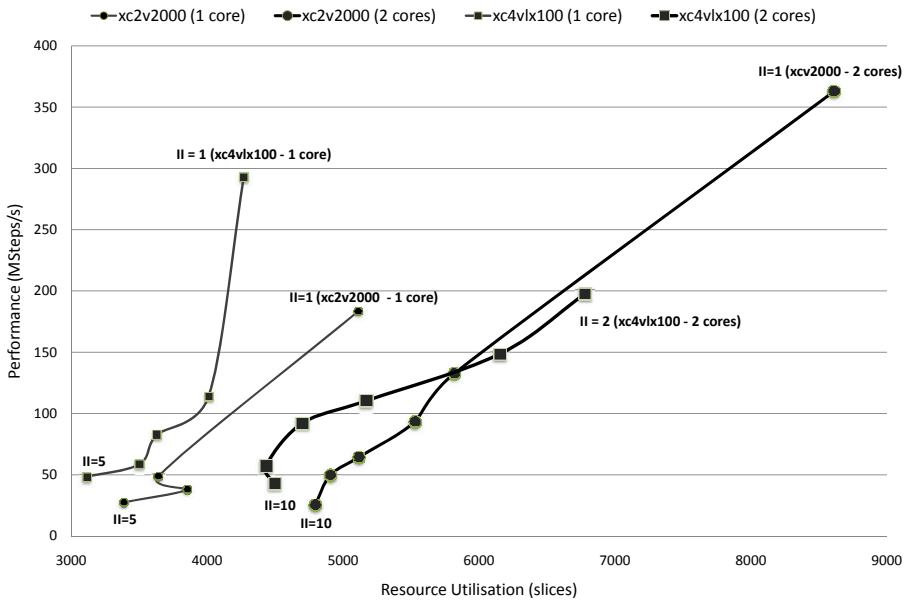


**Figure 6: Design performance against resource utilisation as the initiation interval varies.**

based methodologies include: the Garp approach [2] and SPARK [6]. Approaches based on structural design include: DK4 [3] and Lava [1]. Our approach is unique in that Haydn combines both structural and behavioural design methodologies.

Most of the work that relates to the acceleration of financial simulations focuses on creating a custom design for a specific simulation. An implementation of the BGM method is presented in [11], which uses a scheduling strategy and implementation designed by hand for one specific type of simulation. The methodology presented here is capable of capturing and implementing the published BGM, so in future it should be possible to directly compare the automatically generated version against the previous manually optimised version.

# 7. CONCLUSION

This paper presents the design and implementation of various architectures for Monte-Carlo simulation in order to find the best tradeoff between performance goals such as resource utilisation and execution time. We use our design-flow to automatically generate 57 architectures with different initiation intervals and resource sharing settings. The fastest FPGA architecture runs 37 times faster than a Pentium-4. Note that all FPGA architectures were derived fully automatically once the behavioural design models and constraints were defined.

Future work will focus on improving the scheduling algorithms to minimise resource utilisation and improve routing congestion. Other improvements include an optimised design-flow which can automatically generate constraints based on the results of the last place-and-route report, and to improve the speed of the design process.

# 8. ACKNOWLEDGEMENTS

The support of FP6 hArtes project, Celoxica and Xilinx is gratefully acknowledged.

# 9. REFERENCES

[1] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in Haskell. In *Proceeding of the 3rd ACM SIGPLAN ICFP*, pages 174–184. ACM Press, 1998.

[2] T. Callahan. *Automatic Compilation of C for Hybrid Reconfigurable Architectures*. PhD thesis, University of California, Berkeley, 2002.

[3] Celoxica Ltd. DK Design Suite. `http://www.celoxica.com/dk`.

[4] Celoxica Ltd. Handel-C language reference manual. Technical report, 2004. `http://www.celoxica.com/techlib/files/CEL-W0410251JJ4-60.pdf`.

[5] J. Coutinho, J. Jiang, and W. Luk. Interleaving behavioral and cycle-accurate descriptions for reconfigurable hardware compilation. In *13th Annual IEEE Symposium on FCCM*, pages 245–254, 2005.

[6] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In *Proceedings of the 16th International Conf. on VLSI Design*, pages 461–466, 2003.

[7] G. Marsaglia and W. W. Tsang. The Ziggurat method for generating random variables. *Journal of Statistical Software*, 5(8):1–7, 2000.

[8] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 98.

[9] G. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[10] D. Thomas, J. Bower, and W. Luk. Automatic generation and optimisation of reconfigurable financial Monte-Carlo simulations. In *IEEE Int. Conf. on Application-specific Systems, Architecures and Processors*, 2007.

[11] G. Zhang, P. Leong, C. Ho, K. Tsoi, D. Lee, R. Cheung, and W. Luk. Reconfigurable acceleration for Monte Carlo based financial simulation. In *IEEE International Conference on Field-Programmable Technology*, 2005.