Unifying FPGA Hardware Development

Jacob A. Bower, Wei Ning Cho and Wayne Luk
Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2AZ, UK
{jab00,wnc04,wl}@doc.ic.ac.uk

Abstract

In current FPGA development environments complex projects often end up in an ad-hoc tangle of programming systems; examples include Perl, Makefiles, and Verilog and/or VHDL. To combat this we develop an approach to FPGA development in which a single specification is used to combine: high- and low-level description of custom hardware, parameterisation of existing IP and project build. In this paper we present an abstract overview of our unified approach and a prototype implementation called YAHDL, composed of a set of libraries written in the object-oriented software language Ruby. To explore YAHDL's effectiveness we apply it to an existing project, creating FPGA hardware designs for floating-point Monte Carlo simulations. With this case-study we show it is possible to use YAHDL to simplify the generation of application specific instances of our Monte Carlo architectures while achieving performance in the 200-300MHz range.

1. Introduction

Ever since their introduction, FPGAs have offered the tantalising prospect of an alternate platform for computation somewhere in between the flexibility of software and the raw parallelism of hardware. Today this prospect seems ripe to explode into practical reality, with current FPGAs already having sufficient resources for applications utilising extensive floating-point operations and running in the hundreds of MHz [15].

However, before FPGAs can take off as a platform for general computation in the real-world, a new class of "FPGA engineers" is required to program them. An FPGA engineer's task is to both understand a specific application and map it into an FPGA. To achieve the mapping part of this process in a realistic time-frame with a maintainable and high-performance result, an FPGA engineer will rely on libraries of pre-made and possibly third-party Intellectual Property (IP) such as floating-point units, memory and host processor interfaces etc. This IP needs to be highly parametrised and easily composable in order to maximise applicability and allow optimal performance for a specific application instance.

Unfortunately current FPGA development environments lack a systematic approach to creating hardware systems composed of a diverse range of IP. At the heart of this problem is a heavy reliance on the Verilog and VHDL HDLs for creating synthesisable hardware. These HDLs only provide very low-level parameterisation features in the form of variable width I/O ports, conditional and repeated entity instantiation. Often for highly-custom applications parameterisation at a higher-level is required, for example instantiation of components based on a high-level domain-specific specification or entities with interfaces which automatically vary depending on the other entities to which they are connected. Furthermore VHDL and Verilog have no direct support for working with IP that has to be dynamically re-generated by external programs depending on parameterisation.

Many projects work around the limitations of current VHDL and Verilog-based FPGA environments by using external programming systems. It is not uncommon in complex projects, particularly those involving high-level parameterisation or design-space exploration, to find HDL source is generated and built using ad-hoc combinations of external programming systems such as Perl, C++, Makefiles, etc. The problem with such ad-hoc approaches is that systems can become difficult to understand, modify or maintain.

Our solution to the current ad-hoc treatment of the complete FPGA design process is to adopt a unified approach to FPGA development environment. In our approach, we use a single specification to combine high- and low-level description of custom hardware, project build and parameterisation of existing IP regardless of implementation language or system. The contributions of our approach include the following:

- An abstract overview of our unified approach based on specification of, and build for, a hardware hierarchy with 'internally' and 'externally' generated IP. (Section 3)
- An implementation of our approach called YAHDL. This implementation allows hardware build and design using both internally specified structural designs

1-4244-1472-5/07/$25.00 © 2007 IEEE

and external static or dynamically generated IP in VHDL, Verilog or blackbox-netlist formats. (Section 4)

- A case-study showing YAHDL can be used to create floating-point based designs running in the 200-300MHz range on a Xilinx Virtex 5 device. This case-study is based on and compared to previous work creating Monte Carlo simulations in FPGAs. (Section 5)

We begin our discussion by first considering existing work tackling parametrised FPGA development below in Section 2, and conclude our work in Section 6 with a summary and discussion of future work.

2. Existing Work

The lack of intrinsic support for creating and composing parametrised FPGA designs has already been the focus of much previous work. Of particular interest are projects exploring the creation of new hardware design languages or programming systems for FPGAs.

Some authors have approached this problem by creating dedicated hardware description languages for FPGA design. At the lowest-level, systems have been developed which primarily allow extensive meta-programming of structural hardware design. Examples of these languages include: Ruby [7] Pebble [8], Quartz [14], and Confluence [5]. Other languages allow specification of FPGA designs in a behavioural fashion where the underlying implementation is determined by a compiler such as SAFL [12] and to some extent Handel-C [3]. These custom hardware languages focus exclusively on the problem of describing parametrised hardware designs for synthesis. Constructs for automating the integration of external IP and other external build issues are left widely unaddressed.

Rather than creating dedicated hardware description languages, some authors have created libraries for existing software languages forming programs which when run generate hardware designs. Such libraries can be considered Domain-Specific Embedded Languages (DSELs) [6] for hardware design. The main advantage of this approach is extensive meta-programming of a design is provided through already existing language constructs. Examples of Hardware-DSELs include: JHDL [1] (Java), PamDC [11] (C++), PyHDL [17] (Python interface to PamDC) and myHDL [4] (Python).

Some authors find that DSELs for hardware description allow complex systems to be built [13]. This extra complexity is obtained by leveraging the general purpose nature of a software language to implement wider application functionality whilst remaining tightly coupled to actual hardware design. For example, ASC [10] (A Stream Compiler) built on-top of PamDC allows the generation of a complete FPGA platform from a minimal specification of core stream-based functionality. A considerable range of systems have also been built on JHDL supporting features from run-time reconfiguration [2] to generation of hardware design through a web-based interface [16].

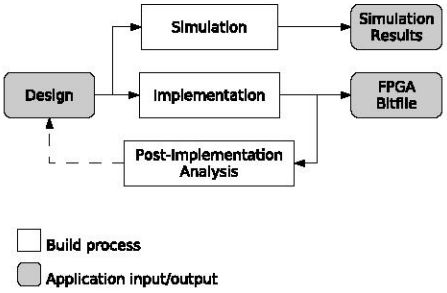

Figure 1. High-level build flow requirement for our unified FPGA development environment.

3. Our Approach

While previous work focuses on creating parametrised hardware IP, little is done to add intrinsic support for building complex FPGA systems which include IP created with multiple languages and methods. In this section we describe our approach to creating an FPGA development environment allowing: hardware design, IP integration and project build within a single unified specification.

We begin by presenting the motivation for our work from which we derive a set of requirements. We then give an abstract overview of our environment based on these requirements.

3.1. Motivation and Requirements

The key potential FPGA advantage for computation compared to software processors is hardware-level parallelism and optimisation without the cost of creating a custom ASIC. However implementing high-performance computation hardware, even in ASIC, is non-trivial. As such to develop custom high-performance systems in a reasonable time-frame it is necessary to compose a system from pre-made high-performance IP. As an FPGA design can be highly application specific, this IP must be highly-parametrised to maximise applicability while enabling specific optimisations.

Once we have a library of FPGA IP at our disposal, we need a way of composing applications with it. Such composition consists of both the instantiation of parametrised IP and custom logic to implement their interaction for a

specific application. Furthermore to fully leverage FPGAs customisation advantage, the composition process itself must also be parameterisable to allow tailoring not only to an application but also to specific instances of that application. Parametrised composition also allows design-space exploration, for example balancing possible hardware designs with actual performance that can be achieved from available FPGA devices.

In addition to specifying the composition of a system we also need to specify the build of this system. While for simple systems this can be achieved with a GUI, a scripted build is beneficial for more complex systems. Scripted build is particularly of benefit to: systems partially or fully generated from a high-level specification, automated design-space exploration and parametrised generation of third-party IP.

Our experience has shown that implementation of complex systems including the features above usually involves an ad-hoc combination of external programming systems, for example Perl, TCL and Makefiles etc. This ad-hoc approach has many problems, including: scripts can become coupled to specific tools or IP generators even though others are available (e.g. XST vs. Synplify Pro), large ad-hoc scripts which have evolved over time become difficult to maintain, and system-level parameterisation becomes awkward to automate as distributed design elements need to interact.

From the combination of the features required for creating high-performance FPGA systems outlined above, and from our experience in achieving these systems using current tools, we derive the following central requirements for a new FPGA development environment:

- Easy integration of existing static or generated IP, regardless of their underlying design system.
- Meta-programmable hardware design allowing custom logic, composition and parameterisation of hardware blocks.
- A scripted build-environment enabling the flow outlined in Figure 1 in which simulation is used for validation and implementation may be iteratively improved.
- All of the above captured in a single coherent programming system.

3.2. Abstract FPGA Environment

Central to our FPGA development environment requirements is the use of a single meta-programmable specification capturing all features of FPGA development. We believe fundamentally these goals are best achieved by creating such a specification as a Domain Specific Embedded Language (DSEL). In other words, using an existing software language augmented with libraries of methods for designing FPGA-based systems. We propose the use of a DSEL, rather than a dedicated language because we see generation, parameterisation and integration of IP as a combined software and hardware design problem. Using an existing software language as a base language for a our specification enables us to tackle the software part of this problem. Furthermore, as shown in Section 2, the feasibility of creating FPGA designs using a DSEL is demonstrated by existing systems such as JHDL, PamDC, myHDL and pyHDL.

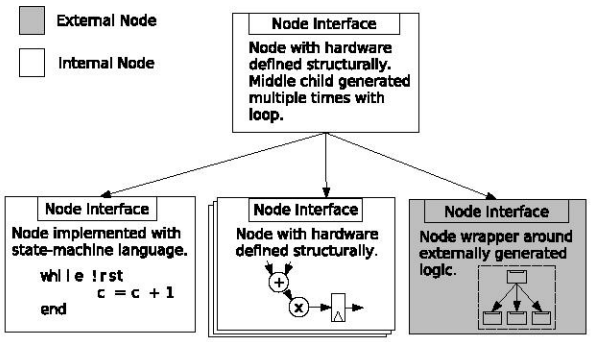

Figure 2. Example hardware hierarchy expressible in our proposed DSEL.

In order to meet our requirements of easy integration of external IP and custom hardware generation, we propose the bulk of our DSEL functionality revolves around creating a hierarchy exemplified in Figure 2. This figure represents a meta-programmed specification for hardware using a typical hierarchical approach with the added novelty that nodes are classified as either *Internal* or *External*.

In our hierarchy, internal nodes allow us to create custom meta-programmed hardware generated using structural hardware design primitives in our DSEL similar to existing systems such as PamDC or JHDL. Due to the software nature of internal nodes they may also use or create further specialised constructs using the base DSEL software language enabling higher-level design. The execution of these higher-level constructs can eventually invoke low-level constructs resulting in actual hardware generation. Figure 2 shows an internal node with a brief high-level state-machine description as an example of this.

External nodes in our hierarchy provide our mechanism for integrating third-party IP. These external nodes can either be references to external hardware source code not specified directly in our DSEL for example VHDL or Verilog files, or a specification to invoke an external program to generate hardware for this node. Combining these external nodes with those created internally is our key to achieving

unified integration of new and existing IPs.

To achieve our requirement of a scripted build process, we propose the remainder of our DSEL focus on implementing and controlling a *Build Manager* for processing our hierarchy of Internal and External nodes. The basic task of this build manager is to implement the flow shown in Figure 1. To implement this flow in an abstract sense, our build manager will have two phases of operation: *Hardware Generation* and *System Build*. During hardware generation, our build manager will traverse through our DSEL created hierarchy, generating hardware as it passes each node. For an external node this hardware generation will result in the build manager receiving a list of external files needed to implement its branch of the hierarchy. These external files must be in a format the build manager knows how to integrate, for example a common HDL or a black-box netlist. If the external implementation of this node is not encapsulated in a format the build-manager can work with directly (for example in an another HDL language like Confluence or Handel-C), then an external node may execute an external program to generate appropriate files (such as Handel-C compilation to Verilog). After the build manager has fully traversed a hierarchy it will then enter its System Build phase in which all internal and externally generated IP are compiled together using standard tools for simulation or synthesis and place-and-route etc.

To implement scriptability, our build manager must itself be implemented in the language forming our DSEL. Automation/scriptability of the build manager then arises through its pragmatic invocation and parameterisation using the base language for our DSEL. For example, automated design space exploration can be achieved by invoking the build manager multiple times with appropriately parametrised hierarchy instances. In this way we can realise the feed-back loop shown in Figure 1. In order to ensure a full specification of all build processes, it is is important that our build manager and node implementations completely incorporate functionality to invoke and generate any necessary input files for external processes.

4. Environment Implementation

In order to realise our FPGA development environment we have created a DSEL called YAHDL (Yet Another Hardware Design Library) using the dynamic object-oriented software language Ruby[1] [9]. Our current implementation includes the following functionality:

1. Classes for creating a hierarchical hardware design including internal and external nodes.

[1]Not to be confused with the declarative hardware description language also called Ruby.

```
class Adder < HDesign
  # Constructor with width parameter
  def initialize(in_width)
    # Call default constructor for HDesign
    super()
    # n-bit inputs
    hInput in_width, "in_a"
    hInput in_width, "in_b"
    # (n+1)-bit output
    hOutput (in_width + 1), "out"
    # Store width in an instance variable
    @in_width = in_width
  end
end
```

Listing 1. Interface definition for an n-bit adder.

2. Classes for describing internal nodes using structural design with output in either Verilog or VHDL.

3. A configurable build manager with modules for targeting Xilinx FPGAs.

We choose Ruby as our base software language as it has a number of language features which allow a neat implementation of our DSEL. Specifically we take advantage of Ruby's: Text processing capabilities to generate HDLs and configuration files, functions for scripting/managing external processes, object-orientation (we map nodes in our hardware hierarchy directly onto Ruby objects), and operator overloading to simplify specification of internal nodes. In general we also believe Ruby has a simple, familiar and clean syntax which helps keep specifications in our DSEL clear.

In YAHDL, we compose a hardware hierarchy using Ruby objects implementing classes descended from a base `HDesign` class. This `HDesign` class includes methods for specifying node I/O interface ports which are common to both internal and external nodes. Listing 1, shows an example class descended from `HDesign` implementing an interface for a variable-width adder node. From the listing, keywords `hInput` and `hOutput` are Ruby function calls to methods implemented in the `HDesign` base class. These methods create named I/O ports for parent nodes and internal logic to connect to. For external nodes I/O ports must match top-level ports in external hardware sources. As I/O ports are created using regular Ruby method calls they can be generated using standard Ruby control flow constructs such as for-loops, and if-statements. In the Listing 1 example, the width in bits of the adder I/O ports is controlled by a parameter passed to the class constructor.

In YAHDL, composition of a hierarchy is implicitly driven by internal nodes instantiating child internal or external nodes as part of a structural de-

```
# Possible design for Adder example
@out[] = HAdd.new(@in_a, @in_b)

# Adder via overloaded operators
@out[] = @in_a + @in_b

# Registered adder
# (assumes a clk input)
hRegister (@in_width + 1), "reg", @clk
@reg[] = @in_a + @in_b
@out[] = @reg

# A 4-bit counter using Adder instance
hRegister 4, "reg", @clk
add_inst = hComponent Adder.new(4)
add_inst.in_a[] = 1
add_inst.in_b[] = @reg
@reg[] = add_inst.out[0..3]
# (Count can be read from @reg)
```

Listing 2. Examples of structural hardware constructs for internal nodes

sign. Internal nodes are implemented by extending a sub-class of `HDesign` called `HDesignStructural`. `HDesignStructural` includes a number of methods for creating structural hardware designs consisting of: wires, Boolean logic, basic integer arithmetic units and child `HDesign` instances. Classes extending `HDesignStructural` must provide a method called `design` as an execution entry point for generating their internal hardware during a build managers design traversal process.

Low-level hardware generation within internal node `design` methods is implemented by forming a graph of objects implementing classes descended from a base `HLogicNode` class. Examples of these logic elements include: combinatorial logic elements (`HAdd`, `HXor`, etc.), and registers (`HRegister`). To improve the readability of graph construction we take advantage of Ruby's operator overloading. Some examples of creating structural hardware are shown in Listing 2. In order to generate files for use in synthesis or simulation a pretty-printer is called by the build manager to translate graphs of `HLogicNodes` into a target format. Currently we provide Verilog and VHDL pretty-printers the choice of which is specified globally with a build manager parameter.

In YAHDL we create external nodes by instantiating classes descending from a `HDesignBlackBox` class, itself a `HDesign` descendant. Implementations of `HDesignBlackBox` must provide a method called a `doBuild` which returns a list of files to be used by the build manager during its system build phase. In this way an

```
class AdderExternal < HDesignBlackBox
  def initialize
    super()
    hInput 4, "in_a"
    hInput 4, "in_b"
    hOutput 5, "out"
  end
  def doBuild(build_manager)
    return [HVHDLFile.new("adder4.vhd")]
  end
end
```

Listing 3. Example external node referencing a 4-bit adder in an external static VHDL file.

```
class AdderExternal < HDesignBlackBox
  # Constructor as in Listing 1
  def doBuild(build_mngr)
    # Lists of input and output files
    # used by the build manager
    infiles = []
    outfiles = ["adder.edf"]
    command = "addgen -w " + @in_widths
    # Use build manager to run a command
    build_mngr.run(command, infiles, outfiles)
    # Return list with output netlist
    return [HEDIFFile.new("adder.edf")]
  end
end
```

Listing 4. Example external node invoking an external adder generator.

external node can either become just a reference to external HDL or pre-compiled netlists by returning their file-names directly from the `doBuild` method. Alternatively external nodes can generate source files or netlists by invoking external programs in their `doBuild` implementation. Listings 3 and 4 show two examples of external nodes: a fixed 4-bit adder in a static VHDL file and a parametrised adder in EDIF format generated by an external program called 'addgen'.

In YAHDL we provide a basic build manager class implementing the design traversal and system build functionality described in Section 3.2. On construction this build manager object is parametrised with further classes which implement vendor specific or user controlled facets of the build process. Current parameters include: target output format for internal designs, HDL synthesis tool and bitfile implementation tool-flow. Currently we have support for Verilog and VHDL for output internal designs, XST and Synplify Pro as HDL synthesis tools and standard ISE tools for bitfile implementation (ngdbuild, map, par, etc.). We choose to use VHDL and Verilog as targets for our in-

```
device = HXilinxV2Part.new(6000, "ff1152")
design_root = Adder.new(4)
makeXSTNetlist(device, design_root)
```

Listing 5. Example build with XST only.

ternal designs as this allows us to leverage the optimisation features already present in synthesis tools supporting these languages. All facets of external tool invocation for build are codified within YAHDL/Ruby. This includes automated generation of any data, script and configuration input files needed to drive these programs. For example project scripts for synthesis tools and constraint files for place-and-route etc.

To simplify the use of our build-manger we have created a set of utility methods for common instantiations including: complete bitfile generation, HDL synthesis to black-box net-list, and HDL generation only. These utility methods all take the root of a hardware design hierarchy and some require additional parameters, for example target FPGA for bitfile generation. Listing 5 shows an example of building a black-box netlist (.ngc file) with Xilinx XST targeting a Virtex II device for a 4-bit adder for a design based on the example interface in Listing 1.

In addition to implementing the basic features from our abstract FPGA environment, our build manager has additional functionality for managing external processes executed by external nodes. Listing 4 includes an example of how our build manager is used to invoke an external process by specifying the command to run, input files and expected output files. The build manager uses this information to try and minimise build-times by only running identically parametrised external programs once. The build manager achieves this by keeping a database of external commands executed including hashes of their input files. This database by default persists across build manager executions. An external program will be run only if one of the input files has a different hash or the command parameters have changed. This is an important feature as it can greatly reduce the number of external build processes which need to be called when using design-space exploration or when certain types of generated nodes are used repeatedly in a large system.

5. Case Study

To evaluate our FPGA development approach we use YAHDL to re-create an architecture previously developed for accelerating floating-point, Monte-Carlo based financial applications [15]. In this section we review our Monte-Carlo architecture and compare our original implementation with our new version using YAHDL.

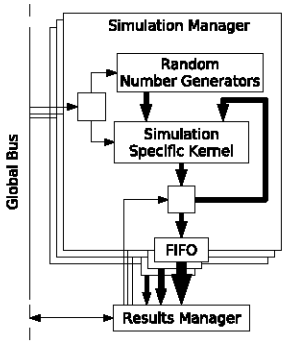

Figure 3. Architecture for Monte Carlo simulation.

5.1. Monte-Carlo Architecture Overview

Monte-Carlo simulations are ideal candidate for FPGA implementation as they have high computational complexity, require little communication bandwidth, and have a regular structure. To explore this potential, we develop an architecture targeted at FPGAs for creating a specific form of Monte-Carlo simulations commonly found in financial applications. The focus of our architecture is a combination of maximising re-usability while allowing high processing throughput.

The form of Monte-Carlo simulation we optimise for is characterised by a simulation kernel of the form: $x_{i+1} = f(x_i, g, R)$. In this kernel, x is some form of state which is iteratively updated by a function f. This f function is deterministic with no internal state, and each application of f depends only on: the current value of x, global constants g and a set of IID (Independent, Identically Distributed) random numbers R which are freshly generated per application. Many simulations conforming to this pattern can be implemented in hardware with hazard-free pipelines with no feed-back other than the state being iterated upon.

We both hide and take advantage of hardware pipelines needed for these simulations using an architecture shown in Figure 3. At the centre of this architecture is a simulation specific implementation of f which we call the 'Simulation Kernel'. We maximise the utilisation of simulation kernels by filling them with concurrently running independent simulations in a C-Slow style with independent simulation iterations in alternate pipeline stages. We further maximise performance for a single FPGA device by encapsulating simulation kernels in self-contained 'Simulation Managers' which can be easily replicated within a single FPGA device. Final results from all independent simulations are accumulated by a shared 'Results Manager' which computes the result of the overall Monte-Carlo simulation. Set-up of a simulation and final results acquisition is performed over a 'Global Bus'.

Circuit	Area (Slices)	Speed (MHz)	FPUs
8-bit exponent, 24-bit mantissa			
DualVAR	4718	254	13
GARCH	6349	203	11
RndJump	4011	242	9
LogNormal	2244	262	7
6-bit exponent, 17-bit mantissa			
DualVAR	4156	232	13
GARCH	5776	209	11
RndJump	3794	255	9
LogNormal	2008	323	7
6-bit exponent, 8-bit mantissa			
DualVAR	3899	254	13
GARCH	5189	216	11
RndJump	3603	241	9
LogNormal	1641	329	7

Table 1. YAHDL Monte-Carlo hardware results.

5.2. Implementations Comparison

Previously to implement our Monte-Carlo framework we used a combination of: Handel-C, Xilinx CoreGen and Makefiles. While the majority of our design was implemented in Handel-C, it was primarily constructed using structural composition with almost none of Handel-C's behaviour inference being used. Despite this, we used Handel-C as its macro features allowed us to heavily parametrise our design. Our Makefiles provided a front-end interface to compilation, allowing us to specify a target device, amount of pipeline replication, and selection of simulation kernel implementation.

The main area in which our YAHDL and original implementations differ is in the generation of our Simulation Kernels. In our original approach we manually invoke Xilinx CoreGen and individually create floating-point units as needed for our kernels. We then create a separate Handel-C design for each of our different kernels, instantiating our CoreGen units using a series of macro expressions. Our macro expressions join the CoreGen units together forming streaming pipeline data-paths implementing our Simulation Kernels.

While our original implementation is sufficient to demonstrate the effectiveness of our Monte Carlo architecture it has two major conceptual limitations which limit its applicability to real-world industry. The first limitation is our Handel-C simulation kernels bear little resemblance at the source level to the functions they implement. In a real-world scenario it would be impractical for us to expect anyone inexperienced with Handel-C to directly create or modify kernel designs for our framework. The second limitation is any changes needed to our floating-point units require us to re-run CoreGen. Particularly without the use of an external programming system it would be very tedious for us to perform a design space-exploration of the effects of varying floating-point unit precision. Such a study would require re-running CoreGen manually for all floating-point operator and precision combinations.

```
class LogNormalWalk < HEquationDesign
  def initialize(operator_factory)
    super(operator_factory)
    hVariable "x"
    hVariable "R"
    hParameter "mu"
    hParameter "sigma"
    hResult "x_prime"
  end
  def design
    @x_prime[] = @x *
        (1 + @mu + (@sigma * @R))
  end
end
```

Listing 6. Example YAHDL simulation kernel specification.

Our YAHDL implementation simultaneously addresses the low-level nature of simulation kernel design for our architecture and the necessity to manually run CoreGen. We achieve this by creating a new class of internal node for YAHDL called `HEquationDesign`, including methods for specifying simulation kernels at a high-level. This high-level specification uses operator overloading to more closely match kernel hardware design to the desired operation. These specifications are also independent of underlying number format or precisions details. A `HEquationDesign` example is shown in Listing 6 and implementing the kernel: $f(x_{i+1}) = x_i \times (1 + \mu + \sigma R)$, where x_i is an iteration variable, μ and σ are simulation constants (set via our architecture global bus) and R is a random number input.

Our `HEquationDesign` class implementation is itself a YAHDL internal node which extends the `HDesignStructural` class. During a build manager Hardware Generation phase high-level designs captured with out `HEquationDesign` class are compiled to structural YAHDL using a simple As Soon As Possible (ASAP) scheduling algorithm generating a stream pipeline. During this compilation, an 'operator factory' is used to generate arithmetic units. Choice of operator factory is a parameter passed to the constructor of a `HEquationDesign`. Currently we only have one type of operator factory which generates arithmetic units using external nodes which call CoreGen to generate floating-point units. The build of

these units is handled by our YAHDL build manager, ensuring CoreGen is only invoked once across all design builds when new operators or precision is required. Precision for our current operator factory is user specifiable but is constant per implementation.

The rest of the hardware in our YAHDL implementation mirrors our previous Handel-C implementation. For the most part, our structural design is simply re-implemented in YAHDL using regular internal nodes descended from our `HDesignStructural` class. No other external build scripts or parameters are needed, with everything specific to our design specified in Ruby using YAHDL methods. We believe this is a significant improvement compared to our original combination of Makefiles and a relatively unconventional use of Handel-C in which its behavioural features are not used.

In Table 1 we show place-and-route results for Monte-Carlo simulations implemented with our YAHDL framework targeting a Xilinx xc5vsx50t-3. These results are comparable to the results we attained in our original work which showed FPGAs could be used to accelerate these simulations by on average 80 times compared to optimised software running on a 2.66GHz Xeon. We also utilise our YAHDL description's improved ability to vary floating-point precision, to acquire results across a range of floating-point sizes. Within our variations we note when reducing floating-point precision, slice utilisation is clearly reduced, but the effect on clock frequency is less apparent.

6. Conclusion

In this paper we present an abstract FPGA application development environment based on a single specification which unifies integration and build of custom hardware designs. Central to this specification is a description of hardware based on a hierarchy of 'internal' and 'external' nodes, and a build system to implement this hierarchy. We demonstrate the practicality of our approach by creating an implementation of our environment called YAHDL. We apply YAHDL to an existing project for creating floating-point based Monte Carlo simulations in FPGA, and are able to simplify the specification of these simulations, increase flexibility of the design and eliminate the need for ad-hoc external programming systems for build.

In future we plan to further explore the extent to which FPGA application development can be encapsulated in a single unified description. Particularly we will soon be addressing the issue of how to effectively use YAHDL to drive simulation of FPGA hardware designs. We plan to achieve this by enabling YAHDL to invoke and take control of existing commercial simulation packages using tools such as the Verilog Procedural Interface (VPI) simulation/software communication standard. We also hope to further explore how YAHDL can simplify the use of advanced FPGA application features including dynamic reconfiguration.

Acknowledgment. The support by the UK Engineering and Physical Sciences Research Council and Dr David Thomas is gratefully acknowledged.

References

[1] P. Bellows and B. Hutchings. JHDL - An HDL for reconfigurable systems. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pp.175 – 184, 1998.

[2] P. Bellows and B. Hutchings. Designing run-time reconfigurable systems with JHDL. *J. VLSI Signal Process. Syst.*, Vol. 28, No. 1-2, pp.29–45, 2001.

[3] Celoxica. Handel-C. *http://www.celoxica.com/products/dk/*, 2006.

[4] J. Decaluwe. MyHDL: a python-based hardware description language. *Linux J.*, Issue 127, page 5, 2004.

[5] T. Hawkins. Confluence, *http://funhdl.org/*, 2006.

[6] P. Hudak. Modular domain specific languages and tools. *Intl. Conf. on Software Reuse*, pp.134–142, 1998.

[7] W. Luk, S. R. Guo, N. Shirazi, and N. Zhuang. A framework for developing parameterised FPGA libraries. In *Intl. Conf. on Field Programmable Logic*, pp.24–33, 1996.

[8] W. Luk and S. McKeever. Pebble: A language for parametrised and reconfigurable hardware design. In *Intl. Conf. on Field Programmable Logic and Applications*, pp.9–18, 1998.

[9] Y. Matsumoto. *Ruby In A Nutshell*. O'Reilly, 2001.

[10] O. Mencer. ASC: A stream compiler for computing with FPGAs. *IEEE Transactions on CAD of ICs and Systems*, Vol. 25, No. 9, pp.1603–1617, 2006.

[11] O. Mencer, M. Morf, and M. J. Flynn. PAM-Blox: High performance FPGA design for adaptive computing. *IEEE Symposium on FPGAs for Custom Computing Machines*, pp.167–174, 1998.

[12] A. Mycroft and R. Sharp. Hardware/software co-design using functional languages. *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pp.236–251, 2001.

[13] B. E. Nelson and B. L. Hutchings. Using general-purpose programming languages for FPGA design. *IEEE Conf. on Design Automation*, pp.561–566, 2000.

[14] O. Pell and W. Luk. Quartz: A framework for correct and efficient reconfigurable design. *IEEE Conf. on Reconfigurable Computing and FPGAs*, pp. 14, 2005.

[15] D. B. Thomas, J. A. Bower, and W. Luk. Hardware architectures for monte-carlo based financial simulations. *Intl. Conf. on Field-Programmable Technology*, pp.77–380, 2006.

[16] M. J. Wirthlin and B. McMurtrey. IP delivery for FPGAs using applets and JHDL. *IEEE Conf. on Design Automation*, pp. 2–7, 2002.

[17] P. Haglund and O. Mencer, W. Luk, and B. Tai. Hardware Design with a Scripting Language. *Intl. Conf. on Field-Programmable Technology*, pp.1040–1043, 2003.