Object Oriented Circuit-Generators in Java

Michael Chu   Nicholas Weaver*   Kolja Sulimma   André DeHon
John Wawrzynek

Abstract

Generators, parameterized code which produces a digital design, have long been a staple of the VLSI community. In recent years, several Field Programmable Gate Array (FPGA) design tools have adopted generators, as it is a convenient way to specify reusable designs in a familiar programming environment. We have built a generator framework in Java as a basis for programming reconfigurable devices and as a tool to be embedded in larger development systems. In addition to the conventional benefits of generators, this powerful framework allows for partial evaluation, simulation, specialization, and easy inclusion of other automatic services. In order to verify the utility of this system, we have implemented several applications using this framework and compared them with implementations using schematic capture and HDL synthesis. Our system runs significantly faster and produces comparable or superior results when mapped to a target FPGA.

1 Introduction

Generators are a method of digital design where the designer writes a program which, when executed, creates an instance of the design. Generators have been used for the past 20 years within the VLSI community because it is easy to specify parameterized and reusable designs. By allowing the programmer to specify bit width, area, performance, and other parameters as arguments to his routine, a single generator can be easily reused under different circumstances.

As a simple example, it is equally easy to describe an $N \times M$ bit multiplier as it is to specify a 16-bit multiplier by simply writing a loop which builds up the multiplier to the desired size. Furthermore, the multiplier can use different implementation strategies under different circumstances, in a way transparent to the generator's user. Anyone wishing to utilize this design simply calls the generator and specifies the desired parameters, allowing the single design to be easily used and reused wherever a multiplier is required.

Furthermore, most generator systems produce high quality results because they can embed designer expertise on how to best construct a particular instance of some design. Instead of specifying a behavioral computation and requiring an automatic tool to search for a good structural implementation, most generators directly compose their desired structure. This allows a well-designed generator to produce high-quality designs with minimal mapping time. The quality of the generator is, of course, highly dependent on the generator writer's skill and foresight in anticipating its domain of use. A poorly written generator would produce a worse structure than a compiled behavioral description.

Another advantage of many generator systems is the ability to embed the framework within an existing and familiar programming language. C, C++, Scheme, Java, or practically any other language can be used to create a generator system. This way a programmer does not have to learn a new language and syntax in order to build reconfigurable computing designs, reducing one of the psychological barriers which need to be overcome before reconfigurable computing becomes mainstream. As an additional benefit, the programmer may use the full expressive power of the language to express and encapsulate his choices.

These benefits are well understood and have been the rationale behind several systems including Digital's PamDC[4], its predecessor[2], the PAM-Blox system[7] and National Semiconductor's D4 language[8]. Other development systems, such as the Cadence tool-suite, encourage developers to create adhoc generators such as short scripts which tile small components to form a large array. Even more, such as the Xilinx LogicBlox[9] use generators which are hidden from the user. All these existing systems share a common philosophy, where the programmer describes the computation as a series of small logic

---

*Address correspondence to nweaver@cs.berkeley.edu. This work is sponsored in part by the DARPA ACS program under grant DABT63-96-C-0048. Further support comes from NSF grant CDA 94-01156 and the California State MICRO Program.

units and connections, essentially building up netlists out of primitive components.

Our system provides several benefits, including easy composition of generators, a great deal of flexibility in specifying implementation decisions, convenient portability of initial designs and the development environment between FPGA families, and the ability to create default behaviors for a variety of optimizations and to override these behaviors with domain specific optimizations. Our system makes no distinction between primitives and user specified components, allowing generators to easily compose other generators. Since even primitives are treated as generators, any design which is not specifically optimized for a target FPGA can be easily ported to a different FPGA by simply changing which libraries the design utilizes. The most powerful feature of our system is the ability to embed powerful optimizations and other services into our system. For all such services, a default behavior is defined which can be applied to all generators, but any specific generator can override the default behavior to provide an application specific version.

We have built our generator system using object oriented techniques within Java. In our design philosophy, the programmer composes subcomponents which are objects in the programming system. These objects represent more than a simple netlist: they are also capable of other operations including partial evaluation and simulation. Furthermore, the system retains the programmer's design hierarchy, allowing other automated services to act on this information.

This system provides a general framework for building generators and a set of target specific component libraries. The programmer composes the library generators in order to create his own generator, by defining a Java object which contains slots for the subcomponents and connections. This object's constructor must initialize the subcomponents and connect them together. This object must also contain a method for connecting a components inputs and outputs to the subcomponents which make up the design. Finally, it may contain routines to override the default behavior of any automatic service in our system.

2 Generators in Java

Java[3] is an object oriented programming language designed by Sun Microsystems. Among its notable features are a fully dynamic nature, strong typing, "pure" object oriented methodology, a powerful higher order function abstraction[1], a well supported and robust exception handling mechanism, and a solid metadata interface.[2] Our generator framework exploits these features in order to ease the task of generator specification and enable the construction of powerful services. These language features are only present in languages like Java, Smalltalk and Common Lisp.

The programmer creates a Java class which is inherited from our supplied base-class. When the class is instantiated, it creates its subcomponents, connects them together, and instantiates its outputs. The class also provides a means of attaching its inputs and outputs to the subcomponents. This separation between object creation and defining connections is necessary to allow the complex optimizations which may involve moving and removing components. Furthermore, the class may override methods and provide other methods to supplant or assist automatic services, but is not required to do so. Once the object is instantiated, it builds up a data structure with itself and any subcomponents.

This generator environment provides methods which can access individual subcomponents or apply a user-defined function over all subcomponents. Similarly, this generator environment provides methods for examining all inputs and outputs and accessing all components which are connected to a given input or output. To provide much of this functionality, the generator's runtime maintains a comprehensive set of backlinks and crosslinks in an automatic manner. Therefore, although the programmer only creates forward links, the programmer and all services have access to backlinks for every pointer in the system once the structures have been updated.

This mechanism allows the generator framework and any service to traverse, analyze, and manipulate a design in any manner desired, including tracing all connections, find all components in a design of a specified type, cleanly remove a component from the datastructure, or even replace a component with another component of a compatible type. Although such functionality is expected in any system which relies on a custom compiler, we achieve this without having to perform any modifications to the Java compiler, language semantics, or Java runtime environment. A language such as C++ would require extensive changes to the compiler to enable these analyses or extensive programmer intervention to maintain the data struc-

[1] Java provides anonymous subclasses which are semantically equivalent to `lambda`.

[2] Contained in the package `java.lang.reflect`, which allows a Java program to examine an object's structure and methods.

tures used to perform the analysis of a design[3].

The services themselves are comparatively easy to write, because our runtime system includes several abstractions which make accessing the detailed structure of objects convenient. As a simple example, logic trimming was implemented as an automatic service, requiring roughly 40 lines of Java to implement. In order to define a new automatic service or optimization, a new method must be defined in our provided base-class which implements the desired manipulations. Then, if a separate behavior is required for any generators, these generators must override the new method to provide the correct generator specific behavior.

Since everything in the framework is written in Java, it can be compiled using any Java 1.1 compliant complier and is executable as an application in any Java 1.1 runtime. It uses no native methods or language extensions not present in the Java 1.1 specification. Thus, our system is fully portable between different operating systems and differing compilers, as a consequence of Java's inherent portability.

3 Specifying a Design

A design or component of a design is specified as a Java object which inherits from our base-class, `GenComponent`. It needs to contain slots for the external inputs and outputs and slots for the subcomponents which make up the design. The object's constructor should instantiate its subcomponents, attach any internal components, and instantiate the object's outputs. Also, if the object uses any subcomponents other then `LogicFunction`, it must provide a routine called AttachWires(), which attaches the component's inputs to whatever subcomponets exist.

A simple example[4] is a full adder and the composition of a full adder to create a multi-bit adder.

```
package generator.tutorial;
import generator.*;
import generator.xc4000.*;
```

[3] This is because C++ lacks a metadata interface, which allows a program to discover the structure of objects within the system. A C++ system would require the programmer to register a reference to each slot with the system in order to allow the system to trace and restructure the pointers within a component or would require the generator system to understand how the compiler lays out the objects.

[4] And somewhat artificial since adders are provided as array-specific library components.

```
public class FullAdder extends GenComponent{
  public InputWire a, b, cin;
  public OutputWire cout, sum;
  public LogicFunction summation;
  public LogicFunction carry;

  public FullAdder(){
    summation = new LogicFunction("a ^ b ^ cin");
    carry = new LogicFunction("(a & b)|(a & cin)"
                               +  "|(b & cin)");
    cout = carry.o;
    sum = summation.o;
  }

  public void attachWires(){
    summation.o = sum;
    carry.o = cout;
  }
}

public class Adder extends GenComponent{
  public InputBus a, b;
  public InputWire cin;
  public OutputBus sum;
  public OutputWire cout;
  public FullAdder[] adders;
  int width;

  public Adder(int width){
    this.width = width;
    int i = 0;
    Wire [] sumWires = new Wire[width];
    adders = new FullAdder[width];
    adders[0] = new FullAdder();
    sumWires[0] = (Wire) adders[0].sum;
    i++;
    while(i < width){
      adders[i] = new FullAdder();
      adders[i].cin = (Wire)
                        adders[i-1].cout;
      sumWires[i] = (Wire) adders[i].sum;
      i++;
    }
    cout = adders[--i].cout;
    sum = new Bus(sumWires);
  }

  public void attachWires(){
    int i = 0;
    if(a != null){
      for(; (i < width) &&
            (i < a.getWidth()); ++i){
```

```
adders[i].a = a.getWire(i);
      }
    }
    for(;i < width;i++){
      adders[i].a = null;
    }
    i = 0;
    if(b != null){
      for(; (i < width) &&
    (i < b.getWidth()); ++i){
adders[i].b = b.getWire(i);
      }
    }
    for(;i < width;i++){
      adders[i].b = null;
    }
    if(cin != null){
      adders[0].cin = cin;
    }
    i = 0;
    if(sum != null){
      for(; (i < width) &&
    (i < sum.getWidth()); ++i){
adders[i].sum = sum.getWire(i);
      }
    }
    for(;i < width;i++){
      adders[i].sum = null;
    }
    adders[i-1].cout = cout;
  }

}
```

This example illustrates most of the major features of specifying a simple design in our generator system. The full adder contains three wires which are used as inputs, two which are used as outputs, and two logic functions for the carry and sum logic. The constructor is fairly simple, just instantiating the logic functions and assigning its outputs. The logic function itself takes a string as an argument and parses it according to a simple grammar. All the variables in the expression refer to the inputs of the enclosing component, which the logic function will automatically attach to when necessary[5].

The multi-bit adder's constructor takes a single argument, the bit width of the adder desired. It then creates an array for holding the full adders, connects the carry chain[6], and collects all of the outputs into a single bus. The constructor could be replaced with a more sophisticated version which chooses an implementation strategy based on the size of the adder and the desired performance.

Also required is an `attachWires` method, which goes through and attaches the wires contained in the two input busses to the constituent adders. It needs to make sure that it does not try to dereference inputs and since inputs may change, it also must insure that any old inputs are correctly replaced. Thus it insures that all inputs which are not driven are correctly set to null. A programmer who is confident in how his generator will be used may omit some of the sanity checks, but such a practice is highly discouraged.

4 Benefits over HDL synthesis

Our generator system offers numerous benefits over HDL synthesis, including ease of specifying a specialized design, the ability to perform partial evaluation and other high level transformations, and superior runtime performance.

Most HDLs include simple iteration constructs for building up a design based on simple parameters, such as the `generate` functionality contained within VHDL, but it is fairly awkward to do more complex specializations. For example, it may be comparatively difficult to specify a specialized multiply-by-constant such as the one described in [6], where the multiplier consists of table lookups which are based on the desired constant. Our system makes it comparatively easy to specify such designs, because the program can instantiate arbitrary components based on its program. Even combinational logic can be built up and specified at runtime, by concatenating a string to describe the desired logic.

In addition to specializing a design when it is created, our generator system allows partial evaluation, which allows a design to further specialize itself as additional information becomes available. In partial evaluation, a series of constants are presented to the design. Portions of the design then use the information to both respecialize itself and compute more constants. Partial evaluation is more sophisticated than simple constant propagation, because it allows changes to the structure to improve efficiency.

Most HDL compilers can perform such optimizations only on primitive components. For example, if

[5] We eventually plan to extend the functionality to include pointer and array references and more sophisticated operations

[6] The explicit cast is necessary because InputWire and OutputWire are both supertypes of Wire yet always point to Wire objects. Since Java does not support automatic type coercion rules, it is necessary for the programmer to explicitly cast back-and-forth when assigning an input from an output or vice-versa.

there is no primitive multiplier, an HDL implementation can't observe that one of the inputs is constant and convert the design to a more efficient form. Instead, it can only eliminate some of the adders as they are optimized away. Our system allows any designer of a generator to specify a method for partial evaluation. If such a method exists, it is given the opportunity to restructure the component. This aspect of our system is detailed later in the paper.

Finally, our system offers impressive runtime performance, even when running in a bytecode interpreter. On designs which may take a couple of minutes for an HDL compiler to synthesize, our system requires only a few seconds. The speed comes from several factors, including the generally simple and streamlined nature of generators, that the generators themselves are precompiled pieces of code, and that only optimizations which a programmer believes may benefit a design are invoked. This, combined with the ability to easily perform specializations, opens the potential to runtime reconfiguration, where an FPGA-like device is dynamically reconfigured for the given task.

However, our system does contain some awkward features not present in HDLs. Although generators are an excellent way to specify high quality datapaths, they are generally very awkward for specifying control logic and irregular structures. Furthermore, like most generator systems, it is highly dependent on the skill of the programmer. Although a good programmer can produce superior designs, a poor programmer would probably get significantly better results in an HDL system. These are some of the reasons why we expect our system to mostly be integrated into larger development environments as opposed to a stand-alone development platform.

5 The Library Structure

By defining an array-specific library of components with a common, array-independent interface, our system provides for portable designs. All high-level components, such as adders, counters, multipliers, and similar blocks, possess a library-independent format and operation, allowing designs to be ported by replacing the underlying library.

However, most FPGA architectures contain unique features and there is no prohibition on libraries containing low-level components, designers can use these low-level components, although their existence is not guaranteed on other platforms. The higher level library components can take advantage of such array specific features within their default implementations, and may also present additional, array-specific parameters to the user. This is because the generic forms in our base library will instantiate array-specific subcomponents when possible. Thus, the generic counter will instantiate and benefit from array specific adders.

component	functionality
AddSub	Adder/Subtractor
BinOp	A binary operation
Comparator	A comparator
Counter	A binary counter
Decoder	A decoder
FlipFlop	D Flip-Flops
IOBlock	Array I/O
LogicFunction	Arbitrary logic
MinMax	MinMax testing
Multiplier	An array multiplier
Mux2_1	2 input mux
ScanReg	A scan chain register
Shifter	A multibit shifer
SignExt	Portable sign extender

Table 1: The components of the base library.

Currently, the generator framework has been used to construct a library for the Xilinx 4000E series[9] of Field Programmable Gate Arrays. This library provides several high-level components, including adders, counters, flip-flops, and multipliers. These components are all parameterized, allowing them to implement functional blocks of arbitrary bit-width. A sample of the available library components is listed in Table 1. Also included are several low-level components, including tri-state buffers, and Xilinx input/output pads. Additionally, all Xilinx library elements can create XNF netlists, perform partial evaluation, and simulate synchronous designs.

However, we have designed our library structure to facilitate portability of our system between gate array families. All the common parts in the array specific libraries inherit from generic versions contained within the `parts` package. These generic versions include all the simulation, partial evaluation, and similar functionality common to all implementations. Additionally, these generic versions create their internal structure out of array specific components. Thus, the generic counter will use array-optimized adders if available.

This behavior allows the initial porting to a new FPGA family to occur relatively quickly. All the porter needs to do to target a new FPGA is provide target specific netlist routines for `BinOp`, `FlipFlop`, and `IOBlock`. Afterwards, the porter can further refine the other components to take advantage of array specific features such as carry chains and tri-state buffers, as well as provide components which directly express these array features.

6 Simulation

One of the powerful features of the generator framework is the ability to perform cycle-accurate simulations of synchronous designs and automatically reference the results with a behavioral description of the design, all within the generator framework. The simulation routines have the ability to accept either an input vector or an object which generates patterns, and automatically generate warnings when inconsistencies are developed.

The simulator currently operates in a simple manner. It first sets the chosen wires to the specified values, either by using the supplied input-vector or by calling the test-pattern generator. The simulator then examines all components which use the chosen wires as inputs. If any of these components implement a simulate routine, this routine is called.

The `simulate()` method is contained in all library components and may be included in user-design components, examines the component's inputs and, if possible, assigns outputs. If the output was unassigned, the value is now assigned and the simulation routines for any connected components are called. If the output was already assigned, nothing happens unless the previous assignment was to a different value, which causes the simulator to issue a warning.

This process iterates until no more changes occur, by simply maintaining a queue of objects which need to be examined and processing that queue until it is empty. When the process is complete, all wires which are unassigned issue further warnings. Then all current values are transfered to a slot recording the value at the last cycle, the current values are reset to unassigned, the input wires are set to the new value, and the process repeats.

Since this is integrated into the generator framework, there is no need for an external simulator for functional testing. Furthermore, since high-level components are concurrently simulated with low-level components, this allows a generator-writer to compose a simple behavioral description in Java and have it compared with the structural description, all within the unified framework.

The only major caveat is that the simulator only works properly on fully synchronous designs or combinational logic, since the runtime system can't differentiate between asynchronous feedback and inconsistencies in the design. Although this might be awkward, we don't find this to be a serious limitation, as most "well formed" FPGA-based computational tasks do not posses asynchronous signals, with the exception of asynchronous flip-flop resets which will generate spurious warnings when used.

7 Partial Evaluation

One of the significant potential advantages for reconfigurable computing is the ability to reduce the complexity and increase the performance of a design by specializing around known or rarely changing inputs. As a simple example, a Finite Impulse Response (FIR) filter is significantly smaller when specialized around its coefficient weights. By performing a series of table lookups and adds instead of general purpose multiplications, it is possible to make the design both smaller and faster. This is one of the primary means which reconfigurable computing expects to see major performance advantages over conventional computation.

We accomplish such specializations in two ways: 1) a generator can accept parameters for specialized designs, and 2) a generator can restructure itself to form a specialized instance when some of its inputs become known, either due to some runtime data available after the generator is instantiated or revealed as a side-effect of other generators being optimized. The first method is obvious and intrinsically available in all generator systems, the second is rather unique to most FPGA development systems and generally referred to as partial evaluation.[7]

Partial evaluation is accomplished by first assigning any inputs which are of known value. Then, each component which uses these connections has a chance to examine its inputs and, if possible, rearrange its internal structure, calculate its outputs, or remove itself completely. This process iterates until no more changes can be accomplished. Since outputs can be defined but not undefined, this process is guaranteed to converge.

This is a significantly more powerful technique than the standard constant-propagation supported by most toolsets. For example, a multiplier composed of a series of AND gates and adders will be reduced by constant propagation if the multiplier is a constant but

[7] Partial evaluation has a long history within programming language systems.

not if the multiplicand is constant. With partial evaluation, the multiplier can restructure itself if either the multiplier or multiplicand is a constant. Furthermore, if the restructuring takes advantage of the algebra of multiplication when it conducts its restructuring, it will guarantee that at most $n/2+2$ adders are required to multiply a number by an n-bit constant. Even more sophisticated multipliers, such as those involving table lookups, could be implemented.

It is also insufficient to do such transformations solely when a component is created. For example, if only constructor specified specializations were allowed, an FIR filter which wishes to benefit from specialized multipliers would need to provide a separate set of constructors for constant coefficients. And any component which uses the FIR filter also would need to acknowledge these specializations. Furthermore, if the multiplier has specialization routines added to it, the FIR implementation would need to be changed to take advantage of the additional functionality. By using partial evaluation, it is possible for the user of an FIR component to benefit from specialized multiplications, even when the FIR component itself does not consider such specializations.

The programmer can easily provide partial evaluation routines for his components, by simply defining an `evaluate()` method which examines its inputs and calculates its outputs. As an example, the following is a partial evaluation method for the multi-bit adder example.[8]

```
public class Adder extends GenComponent{
  ...

  public void evaluate(){
    long aval = 0, bval = 0, cval = 0;
    boolean valueKnown = false;
    if(a == null || a.isValueKnown()){
      valueKnown = true;
      // we treat nulls as ground
      aval = (a == null) ? 0 :
                 a.getValue();
    }
    if(b == null || b.isValueKnown()){
      bval = (b == null) ? 0 :
                         b.getValue();
    } else {
      valueKnown = false;
    }
    if(cin == null ||
       cin.isValueKnown()){
      cval = (cin == null) ? 0 :
                       cin.getValue();
    } else {
      valueKnown = false;
    }
    if(valueKnown){
      sum.setValue((aval + bval + cval)
           & ((1 << width) - 1));
      cout.setValue(
         ((aval &
           ((1 << width) - 1)) +
          (bval &
           ((1 << width) - 1)) +
          cval)
         >= (1 << width) ? 1 : 0);
      removeComponent();
    }
  }
}
```

[8] Once again, this is a somewhat synthetic example since the logic functions which make up the adder will perform this optimization, although not quite as efficiently.

This combines well with an additional feature of our system, the ability to save a partially completed design for use later. We eventually envision that a design is created and optimized, then stored in an internal representation. When an actual instance is desired, the intermediate representation can be reloaded, additional optimizations can occur, and then the final netlist is created.

8 Application #1, DNA pattern matching

An application implemented in the generator framework was DNA sequence matching, as it is a well understood problem within reconfigurable computing. This problem was one of the primary applications on the Splash systolic array[5]. DNA sequence matching is usually implemented as a dynamic programming algorithm designed to calculate edit-distance, with a systolic implementation requiring $O(m)$ space and $O(n)$ time, with m and n being the length of the strings being compared.

The generator implementation is a 2-bit edit-distance calculation which is specialized both on the string being matched and the initial value for the systolic cell, resulting in a design which requires less then 4 Xilinx CLBs per character matched. This design is completely equivalent to an implementation created using schematic capture, both in design and performance.

The primary advantage of the generator is the ease of creating a specialized instance. Instead of hand selecting or generating a schematic by choosing from 16 separate, not quite identical cell designs, a single cell was created which accepts the appropriate parameters, and another generator was created which instantiated and connected a series of the systolic cells with the correct parameters.

The specialization itself offers significant benefits. The fully specialized version requires 4 CLBs on a Xilinx 4000, while one which can serially load a string would require 5 CLBs, resulting in a 25% savings in area. Since most applications of this edit-distance calculation involve comparing a single string against a large database, this is a natural target for specialization.

The top-level generator contains a main routine which accepts the string being matched and creates an XNF file which implements this function. It accomplishes this task by first creating a `DNAtest` object and then calling the XNF-creation service on the resulting object. Since the time it takes to create a specialized instance is critical, we attempt to have our system operate as fast as possible. The current runtime performance is decent but not excellent, requiring 30 seconds to generate an XNF netlist for a 25 cell example, running under JDK1.1.4 on an UltraSPARC2 200.

9 Application #2, portions of the RAW benchmarks

The Raw benchmarks[1] are a series of micro-benchmarks designed to test reconfigurable devices and development tools. Each benchmark consists of verilog for a single computational cell, verilog for a set of control logic, and a small C program[9] which tiles a specified number of cells together.

We reimplemented several instances of the RAW benchmarks: `life`, `bubblesort`, `jacobi`, `nqueens`, and `mergesort`. For each benchmark we created a generator version, which we used to create an XNF netlist while running in the JDK1.1.4 bytecode interpreter[10]. We timed how long it took for the java program to execute and produce the desired netlist. Ingo Schaefer of Synopsys took the corresponding verilog and mapped it to an XNF netlist using FPGA Express on a 200 MHz Sun Ultra2. We used a comparable 200 MHz UltraSPARC 2 to execute our generator versions of each benchmark. Then we compared the results from the generator and the results provided by Synopsys by running the generated XNF netlists through the Xilinx M1 toolkit version M1.3.7 (running on a P6 200 under Windows NT for us, on an UltraSPARC for Synopsys) for a Xilinx 4025e-2 [11]. We compared the stated maximum worst-case delay and LUT utilization between the two designs, as well as the time to create the XNF netlist. Not included in the times are the compile time and runtime required to create the RAW benchmark's verilog instances, or the compile time necessary to create the java generators.[12]

The poor performance of our system while creating an instance of the `life` benchmark points out the deficiencies in our system with respect to random logic, which expands out to numerous 2 input boolean operations which stress our traversal and maintenance routines. Since we need to visit and maintain considerable information about every object in our system, designs which consist of numerous small objects will require more time to create netlists then designs consisting of a few large monolithic blocks. The reason

		execution time (M:SS)	
benchmark	*parameters*	*generator*	*verilog*
`bubblesort`	16 8-bit cells	0:11	2:31
`jacobi`	4×4, 16-bits	0:14	0:39
`life`	32×1, 32 iterations	0:56	0:52
`mergesort`	8 16-bit cells	0:13	2:22
`nqueens`	4 rows, 4 cycles	0:15	0:22

Table 2: Parameters and tool runtime for the RAW benchmarks

	design size		*max delay*	
benchmark	*generator*	*verilog*	*generator*	*verilog*
`bubblesort`	226 CLBs	227 CLBs	52 ns	46 ns
`jacobi`	289 CLBs	247 CLBs	26 ns	26 ns
`life`	63 CLBs	110 CLBs	15 ns	16 ns
`mergesort`	429 CLBs	206 CLBs	35 ns	39 ns
`nqueens`	339 CLBs	46 CLBs	21 ns	22 ns

Table 3: RAW benchmark final designs

[9] Essentially an ad-hoc generator

[10] Early experimentation with JDK1.2beta2 with a just in time bytecode compiler showed speedups of 50% over the same generator running in the 1.1.4 bytecode interpreter.

[11] All instances of the benchmarks required roughly 5 to 15 minutes for the Xilinx tools to map a design from XNF to a bitfile for the targeted array, with both the generator and HDL version taking roughly equal time.

[12] It requires 90 seconds to build the entire generator framework including the generic runtime, Xilinx library, and all testcases and benchmarks we implemented. Compiling just the Life benchmark requires 8 seconds.

for our design being smaller is that it was somewhat easier to express the calculation in the life cell in a way which ends up more conducive to the logic trimming which occurs at the edge of each cell.

Otherwise, the results are always time competitive although not always area competitive with HDL based systems. This is largely due to lower level optimizations which the HDL systems implement but are not present in our system.

10 Conclusions

As seen in tables 2 and 3, our current generator framework creates designs which are comparable with current mature FPGA toolflows in performance, and often comparable in area, while being significantly faster at mapping many datapath oriented designs, even when running in a bytecode interpreter.

Our generator system offers many features not present in other systems, including the ability to offer integrated simulation which automatically compares a behavioral description with the structural result, partial evaluation and the ability to specify high level optimizations, and the ability to easily integrate further optimizations into our framework. We expect to continue to develop this framework by adding further extensions including placement and verilog output, interfaces to other tools, and focusing on improving runtime performance.

11 Acknowledgments

Many thanks to Randy Huang and Tim Callahan for their expert assistance with the development tools, Jonathan Babb for assistance with the RAW benchmark suite, and Ingo Schaefer of Synopsys for providing us with the RAW benchmark numbers for FPGA Express.

References

[1] Jonathan Babb, Matthew Frank, Victor Lee, Elliot Waingold, Rajeev Barua, Michael Taylor, Jang Kim, Sirkrishna Devabhaktuni, and Anant Agarwal. The RAW Benchmark Suite: Computation Structures for General Purpose Computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 1997.

[2] Patrice Bertin and Herve Touati. "PAM Programming Environments: Practice and Experience". In *IEEE Workshop on FPGAs for Custom Computing Machines*, Napa Valley, CA, April 1994.

[3] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.

[4] Tauati Herve and Mark Shand, "PamDC: a C++ Library for the Simulation and Generation of Xilinx FPGA Designs". March 30, 1997. *http://www.research.digital.com/SRC/pamette/PamDC.pdf.*

[5] Dzung Hoang, "Searching Genetic Databases on Splash 2", in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1993.

[6] Tom Kean, Bernie New, and Bob Slous, "A Fast Constant Coefficient Multiplier for the XC6200". In *Field Programmable Logic 96.*

[7] O Mencer, M Morf, and M Flynn, "PAM-Blox: High performance FPGA Design for Adaptive Computing", in *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machine*, April 1998.

[8] Charle Rupp, D4 User's Guide.

[9] Xilinx Corporation, *The Programmable Logic Data Book.*