Kiwi: Synthesis of FPGA Circuits from Parallel Programs

David Greaves
Computer Laboratory
University of Cambridge
Cambridge CB3 0FD
United Kingdom
David.Greaves@cl.cam.ac.uk

Satnam Singh
Microsoft Research Cambridge
Cambridge CB3 0FB
United Kingdom
satnams@microsoft.com

Abstract

We describe the Kiwi parallel programming library and its associated synthesis system which is used to transform C# parallel programs into circuits for realization on FPGAs. The Kiwi system is targeted at making reconfigurable computing technology accessible to software engineers that are willing to express their computations as parallel programs. Although there has been much work on compiling sequential C-like programs to hardware by automatically 'discovering' parallelism, we work by exploiting the parallel architecture communicated by the designer through the choice of parallel and concurrent programming language constructs. Specifically, we describe a system that takes .NET *assembly language with suitable custom attributes as input and produces Verilog output which is mapped to FPGAs. We can then choose to apply analysis and verification techniques to either the high-level representation in C# or other* .NET *languages or to the generated RTL netlists. A distinctive aspect of our approach is the exploitation of existing language constructs for concurrent programming and synchronization which contrasts with other schemes which introduce specialized concurrency control constructs to extend a sequential language.*

1 Introduction

Reconfigurable computing technology is well placed to play an important role in the heart of future heterogeneous computing systems. Soon the role of a mainstream software engineer will involve designing programs for processors that contain not only regular CPUs as we know them today but also more specialized processing units e.g. the evolution of today's GPUs as general purpose computing engines and a version of today's FPGAs as a Lego-like 2D parallel computing resource. This presents a valuable opportunity for efficiently expressing many compute intensive calculations using more parallel computing resources. However, heterogeneous systems also pose a huge programming challenge because, up to now, different kinds of compute resources have been modeled and programmed in very different ways. Successful heterogeneous systems need new unified programming techniques if they are to be widely exploited by mainstream developers. Our contribution in this paper is a step in that direction because we try to model parallel computations with parallel programs written in a mainstream language and then effectively transform these descriptions into circuits for realization on FPGAs. Such an approach helps to make reconfigurable computing technology accessible to software engineers who are willing to write parallel programs. The trend towards multi-core processors means that software engineers will be increasingly writing parallel programs even for execution on regular processors.

A significant amount of valuable work has already been directed at the problem of transforming sequential imperative software descriptions into good-quality digital hardware and these techniques are especially good at control-orientated tasks which can be implemented with finite-state machines. Our approach builds upon this work by proposing the use of parallel software descriptions which capture more information from the designer about the parallel architecture of a given problem that can then be exploited by our tools to generate good-quality hardware for a wider class of descriptions.

A novel contribution of this work is a demonstration of how systems-level concurrency abstractions, like events, monitors and threads, can be mapped onto appropriate hardware implementations. Furthermore, our system can process bounded recursive methods and object-orientated constructs (including object pointers).

The output of our system has been tested with the Xilinx XST synthesis flow and we report the experience of producing circuits that perform low-level bus control and parallel implementations of test pattern images for a DVI controller. These circuits were implemented on a Virtex-5 FPGA on the ML-505 development board. Currently our system is in the early stages of design and development and we are working on supporting larger examples and higher level concurrency models.

Throughout this paper when when we refer to an ‘assembly’ language file we specifically mean the textual representation of the byte code file produced by our compilation flow rather than an a .NET assembly which is an altogether different entity.

Although we present work in the context of the .NET system the techniques are applicable to other platforms like the Java Virtual Machine (JVM). The experimental work described in this paper was undertaken on Windows machines and also on Linux machines running the Mono system.

2 Background

A lot of previous research has been directed at transforming sequential C-like programs into digital circuits and this work is strongly related to work on automatic parallelization. Indeed, it is instructive to notice that C-to-gates synthesis and automatic parallelization are (at some important level of abstraction) the same activity although research in these two areas has often occurred without advances in one community being taken up by the other community.

The idea of using a programming language for digital design has been around for at least two decades [4]. Previous work has looked at how code motions could be exploited as parallelization transformation technique [9].

Examples of C-to-gates systems include Catapult-C [13] from Mentor Graphics, SystemC synthesis with Synopsys CoCentric [1], Handel-C [8], the DWARV [14] C-to-VHDL system from Delft University of Technology, single-assignment C (SA-C) [11], ROCCC [2], SPARK [5], CleanC from IMEC [7] and Streams-C [3].

Some of these languages have incorporated constructs to describe aspects of concurrent behavior e.g. the **par** blocks of Handel-C. The Handel-C code fragment below illustrates how the **par** construct is used to identify a block of code which is understood to be in parallel with other code (the outer **par** on line 1) and a parallel for loop (the **par** at line 4).

```
1  par
2    { a[0] = A; b[0] = B;
3      c[0] = a[0]b[0] == 0 ? 0 : b[0] ;
4      par (i = 1; i < W; i++)
5      { a[i] = a[i−1] >> 1 ;
6        b[i] = b[i−1] << 1 ;
7        c[i] = c[i−1] + (a[i][0] == 0 ? 0 : b[i]);
8      }
9      ∗C = c[W−1];
10   }
```

A notable recent example of exploiting high level parallel descriptions for hardware design is the Bluespec SystemVerilog language [12] which provides a rule-based mechanism for circuit description which is very amenable to formal analysis.

Our approach involves providing hardware semantics for existing low-level concurrency constructs for a language that already supports concurrent programming and then to define features such as the Handel-C **par** blocks out of these basic building blocks in a modular manner. By expressing concurrent computations in terms of regular concurrency constructs we hope to make our synthesis technology accessible to mainstream programmers. Although SystemC descriptions may be very efficiently synthesized they still require the designer to think like a digital circuit engineer. Our approach allows software engineers to remain in the software real to help them move computationally demanding tasks from executing on processors to implementation on FPGAs.

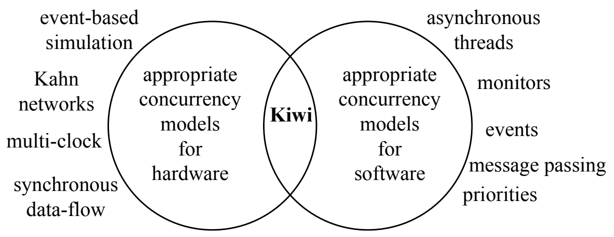

Figure 1: Concurrency models and constructs

3 Parallel Circuit Descriptions

We provide a conventional concurrency library, called Kiwi, that is exposed to the user and which has two implementations:

- A software implementation which is defined purely in terms of the supporting .NET concurrency mechanisms (events, monitors, threads).
- A corresponding hardware semantics which is used to drive the .NET IL to Verilog flow to generate circuits.

A Kiwi program should always be a sensible concurrent program but it may also be a sensible concurrent circuit. The design of the Kiwi library tries to capture a common ground between the concurrency models constructs used for hardware and software (see Figure 1). Our aim to is try and identify concurrency models and constructs which have a sensible meaning both for programs and circuits and this may involve restricting the way they are used in order to support our synthesis approach.

A major paradigm in parallel programming is thread forking, with the user writing something like:

```
ConsumerClass consumer =
        new ConsumerClass(...);

Thread thread1 =
        new Thread(new ThreadStart(consumer.process));
thread1.Start();
```

Within the Kiwi hardware library, the .NET library functions that achieve this are implemented either by compilation in the same way as user code or using special action. Special action is triggered when the newobj ThreadStart is elaborated: the entry point for the remote thread is added to a list that was first created by the user from a command line list of entry points. On the other hand, the call to Threading::Start that enables the thread to run is implemented entirely C# (and hence compiled to hardware) simply as an update to a fresh gating variable that the actual thread waits on before starting its normal behavior.

Another important paradigm in parallel composition is the *channel*. The implementation uses blocking read and write primitives to convey a potentially composite item, of generic type T, atomically. These channels are designed to allow one circuit to produce a result which is consumed by another circuit and in hardware they can be compiled into single place buffers which are placed between a single producer circuit and a single consumer circuit.

```
1  public class channel<T>
2  { T datum;
3    bool empty = true;
4    public void write(T v)
5    { lock(this)
6      { while (!empty)
7          Monitor.Wait(this) ;
8        datum = v ;
9        empty = false ;
10       Monitor.PulseAll(this);
11     }
12   }
13
14   public T read()
15   { T r ;
16     lock (this)
17     { while (empty)
18         Monitor.Wait(this);
19       empty = true;
20       r = datum;
21       Monitor.PulseAll(this);
22     }
23     return r;
24   }
25 }
```

The **lock** statements on lines 5 and 16 are translated by the C# compiler to calls to Monitor.Enter and Monitor.Exit with the body of the code inside a try block whose finally part contains the Exit call. This construct can be used to model a rendezvous between a specific producer and consumer pair.

There are numerous levels at which we might introduce primitives when implementing parts of the Kiwi library for hardware synthesis. An entire function can be recognized and translated to the primitives of the underlying virtual machine. Alternatively, the C# code from the software implementation can be partially translated. In our current implementation of channels, calls to Monitor.Enter and Monitor.Exit were replaced with the following C# code (containing only native functions understood by the core compiler)

```
void Enter(object mutex)
{ while (hpr_testandset(mutex, 1))
    hpr_pause();
}
void Exit(object mutex)
{ hpr_testandset(mutex, 0);
}
```

Monitor.Wait was replaced with

```
void Wait(object mutex)
{ hpr_testandset(mutex, 0);
  hpr_pause();
  while (hpr_testandset(mutex, 1))
    hpr_pause();
}
```

and Monitor.Strobe was treated as a NOP (no operation), because the underlying hardware implementation is intrinsically parallel and can busy wait without cost.

4 Synthesis Flow

In our flow, shown in Figure 2, the .NET Assembly Language is parsed to an AST and then a CIL (common intermediate language) elaborate stage converts the AST to the internal form known as an HPR machine. This was used because a library of code from the University of Cambridge can operate on this format. The HPR machine contains imperative code sections and assertions. The library enables HPR machines to be emitted in various hardware and software forms and converted between them. The imperative code sections can be in series or parallel with each other, using Occam-like SER and PAR blocks.

For illustration, we show some IL code below. Key aspects of the IL code include the use of a stack rather than registers (e.g. mul pops two elements off the stack, multiplies them and pushes the result onto the stack); local variables stored in mutable state (e.g. ldloc.1 pushes the value at local memory location 1 onto the stack); control flow through conditional and unconditional branches; and direct support for overloaded method calls.

```
IL_0019: ldc.i4.1
IL_001a: stloc.0
IL_001b: br IL_005b
IL_0020: ldc.i4.1
IL_0021: stloc.1
IL_0022: br IL_0042
IL_0027: ldloc.0
IL_0028: ldloc.1
IL_0029: mul
IL_002a: box [mscorlib]System.Int32
IL_002f: ldstr "␣"
IL_0034: call string string::Concat(object, object)
```

The IL elaboration stage subsumes a number of variables present in the input source code, including all object pointers.

The resulting machine is simulated with all inputs set to don't know. Any variables which are assigned a constant value and not further assigned in the body of the program (i.e. that part which is not deterministic given uncertain inputs) are determined as compile-time constants and subsumed by the constant propagation function shown in Figure 2. Constructor code must not depend on run-time inputs.

The resulting HPR machine is an array of imperative code for each thread, indexed by a program counter for that thread. There is no stack or dynamic storage allocation. The statements are: assign, conditional branch, exit and calls to certain built-in functions, including hpr_testandset(), hpr_printf() and hpr_barrier(). The expressions occurring in branch conditions, r.h.s. of assignment and function call arguments still use all of the arithmetic and logic operators found in the .NET input form. In addition, limited string handling, including a string concat() function are handled, so that console output from the .NET input is preserved as console output in the generated forms (e.g. $display() in Verilog RTL).

4.1 .NET **Assembly Language Elaboration**

From the .NET AST, an hierarchic dictionary is created containing the classes, methods, fields, and custom attributes. Other declarations, such as processor type, are

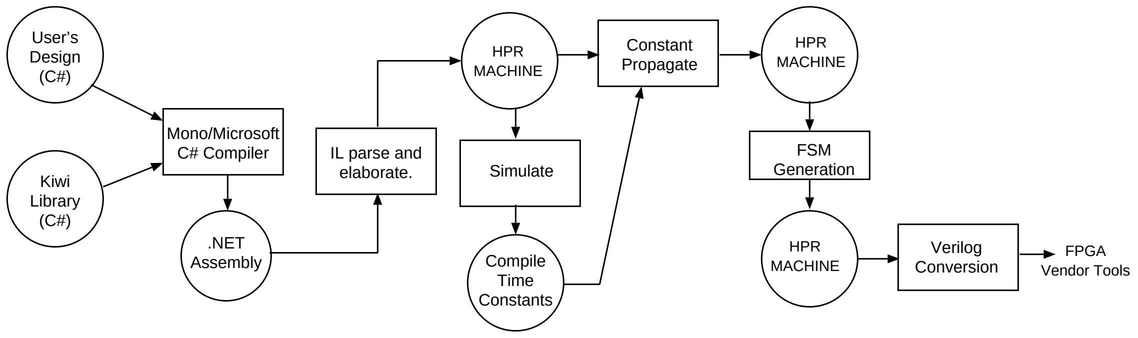

Figure 2: Overall System Flow.

ignored.

A variable is either a static or dynamic object field, a top-level method formal, a local variable, or a stack location. For each variable we decide whether to *subsume* it in the elaboration. If not subsumed, it appears in the VM code that is fed to the next stage (where it may then get subsumed for other reasons). Variables that are subsumed during elaboration always include object and array handles.

We perform a symbolic execution of each thread at the .NET basic block level and emit VM code for each block. .NET label names that are branch destinations define the basic block boundaries and these appear verbatim in the emitted VM code.

Although .NET byte-code defines a stack machine, no stack operations are emitted from the .NET processing stage. Stack operations within a basic block are symbolically expanded and values on the stack at basic block boundaries are stored and loaded into statically scoped surrogate variables, create for this purpose, on exit and entry to each basic block. The surrogate variables are frequently subsumed, but can appear in the VM code and hence, from time-to-time, in the output RTL. Where a method is expanded in line multiple times, the same local and surrogate variable instances are shared across all frames at the same depth of recursion.

A -root command line flag enables the user to select a number of methods or classes for compilation. The argument is a list of hierarchic names, separated by semicolons. Other items present in the .NET input code are ignored, unless called from the root items.

Where a class is selected as the root, its contents are converted to an RTL module with I/O terminals consisting of various resets and clocks that are marked up in the C# source using attributes defined in the Kiwi library. Where a method is given as a root component, its parameters are added to the formal parameter list of the RTL module created. Where the method code has a preamble before entering an infinite loop, the actions of the preamble are treated in the same way as constructors of a class, viz. interpreted at compile-time to give initial or reset values to variables. Where a top-level method exits and returns a non-void value, an extra parameter is added to the RTL module formal parameter list. Additional fields may be declared as I/O terminals using attribute mark up within the C# source code:

```
[OutputBitPort(''scl'')]
static bool scl;
[InputBitPort(''sda_in'')]
static bool sda_in;
```

RTL languages, such as Verilog and VHDL, do not support the common software paradigm of method entry points and upcalls. Threads are not allowed to cross between separately-compiled modules. To provide C# users with procedure call between separately-compiled sections, while keeping our output RTL synthesisable to FPGA, we must provide further connections to each RTL module that implement RPC-like stubs for remote callers.

Certain restrictions exist on the C# that the user can write. Currently, in terms of expressions, only integer arithmetic and limited string handling are supported, but floating point could be added without re-designing anything, as could other sorts of run-time data. More importantly, we are generating statically allocated output code, therefore:

1. arrays must be dimensioned at compile time
2. the number of objects on the heap is determined at compile time,
3. recursive function calling must bottom out at compile time and so the depth cannot be run-time data dependent.

The start-up path is split off from the main process loop by running an interpreter on the code up to the first blocking primitive or to the top of the first loop that cannot be unwound (for reasons of it being infinite or the number of trips depending on run time data values). The main process loop of each thread is converted to an LTS by defining a sequencer. The sequencer is an automata with a state defined for each blocking primitive and for each .net program label that is the destination for more than one flow of control. The I/O operations and operations on local variables performed by the program are all recorded against the appropriate arc of the sequencer.

4.2 FSM Generation

The input and output to the FSM generation stage are both HPR machines. Each input machine consists of an array of instructions addressed by a program counter. Each instruction is either an assignment, exit statement, built-in primitive or conditional branch. The expressions occurring in various fields of the instructions may be arbitrarily complicated, containing any of the operators and referentially-transparent library calls present in the input language, but their evaluation must be non-blocking.

The output machine consists of an HPR parallel construct for each clock domain. The parallel construct contains a list of finite-state-machine edges, where edges have two possible forms:

```
(g, v, e)
(g, f, [ args])
```

where the first form assigns e to v when g holds and the second calls built-in function f when g holds.

An additional input, from the command line, is an unwind budget: a number of basic blocks to consider in any loop unwind operation. Where loops are nested or fork in flow of control, the budget is divided amongst the various ways. Alternatively, in the future, the resulting machine can be analyzed in terms of meeting a user's clock cycle target and the unwinding decisions can be adjusted until the clock budget is met.

The central data structure is the pending activation queue (Figure 3), where an activation has form $(p == v, g, \sigma)$ and consists of a program counter (p) and its current value (v), a guard (g) and an environment list (σ) that maps variables that have so far been changed to their new (symbolic) values. The guard is a condition that holds when transfer of control reaches the activation.

Activations that have been processed are recorded in the completed activation queue and their effects are represented as edges written to the output queue. All three queues have checkpoint annotations so that edges generated during a failed attempt at a loop unwind can be rolled-back.

The pending activation queue is initialized with the entry points for each thread. Operation removes one activation and symbolically steps it through a basic block of the program code, after which zero, one or two activations are returned. These are either converted to edges for the output queue or added to the pending activation queue. An exit statement terminates the activation and a basic block terminating in a conditional branch returns two activations. A basic block is also terminated with a single activation at a blocking native call, such as hpr_pause(). When returned from the symbolic simulator, the activation may be flagged as blocking, in which case it goes to the output queue. Otherwise, if the unwind budget is not used up the resulting activation(s) go to the pending queue. If the budget is used up, the system is rewound to the latest point where that activation had made some progress.

The basic rules for assignment and conditional branch, implemented by the symbolic simulator, with guard g and with environment σ, are:

$$\begin{array}{rcl} [\![\, \mathtt{v := e;} \,]\!]_{(n,g,\sigma)} & \rightarrow & [(n+1, g, [[\![\, e \,]\!]_\sigma / v]\sigma)] \\ [\![\, \mathtt{if\ (e)\ goto\ d;} \,]\!]_{(n,g,\sigma)} & \rightarrow & [(d, g \wedge [\![\, e \,]\!]_\sigma, \sigma), \\ & & (n+1, g \wedge {}^{\sim}[\![\, e \,]\!]_\sigma, \sigma)] \end{array}$$

The conditional branch returns a list of two activations.

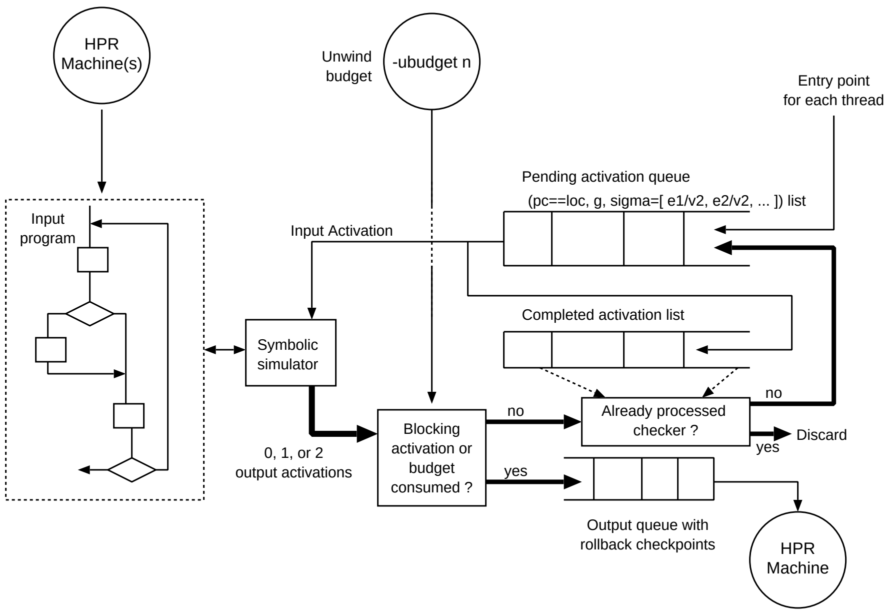
HPR Machine(s)
Unwind budget
-ubudget n
Entry point for each thread
Pending activation queue
(pc==loc, g, sigma=[e1/v2, e2/v2, ...]) list
Input program
Input Activation
Completed activation list
Symbolic simulator
no
Blocking activation or budget consumed ?
no
Already processed checker ?
Discard
yes
0, 1, or 2 output activations
yes
Output queue with rollback checkpoints
HPR Machine

Figure 3: Conversion of control flow graph to FSM.

Activations are discarded instead of being added to the pending queue if they have already been successfully processed. Checking this requires comparison of symbolic environments. These are kept in a 'close to normal form' form so that typographical equivalence can be used. A more-powerful proof engine can be used to check equivalence between activations, but there will always be some loops that might be unwound at compile time that are missed (decidability).

Operation continues until the pending activation queue is empty.

The generated machine contains an embedded sequencer for each input thread, with a variable corresponding to the program counter of the thread and states corresponding to those program counter values of the input machine that are retained after unwinding. However, the sequencer is no longer explicit; it is just one of the variables assigned by the FSM edges. When only one state is retained for the thread, the program counter variable is removed and the edges made unconditional.

The output edges must be compatible. Compatible means that that no two activations contain a pair of assignments to the same variable under the same conditions that disagree in value. Duplicate assignments of the same value at the same time are discarded. This checking cannot always be complete where values depend on run-time values, with array subscript comparison being a common source of ambiguity. Where incompatibility is detected, an error is flagged. When not detected, the resulting system can be non-deterministic.

The built-in hpt_testandset() function, operating on a mutex, m, solves non-determinism arising from multiple updates at the same time using an ordering that arises from the order the activations are processed. (Other, fairer forms of arbiter could also be implemented.) Mutexes are statically-allocated boolean values. The acquire operation returns the previous value from the symbolic environment, σ, of the activation, or the mutex itself if it is not present, while updating the environment to set the mutex. A clear operation is implemented as a straightforward reset of the mutex:

$$\begin{array}{lcl} [\![\, \texttt{hpr\_testandset}(\texttt{m}, 1) \,]\!]_\sigma & \rightarrow & (\sigma(m), [1/m]\sigma) \\ [\![\, \texttt{hpr\_testandset}(\texttt{m}, 0) \,]\!]_\sigma & \rightarrow & (0, [0/m]\sigma) \end{array}$$

Multiple set and clear operations can occur within one clock cycle of the generated hardware with only the final value being clocked into the hardware register.

If all variables are kept in flip-flops, it is almost trivial to convert the HPR machine in FSM form to an RTL design. However, we map arrays in the C# source code to arrays to RAMs in hardware and scalar variables to flip-flops. In the future we will extend the Kiwi attributes set so that the user can control which variables are placed in which output RAMS. A static structural hazard occurs when the actions of a single transition require more than one operation out of a RAM, such as reads to a single-ported RAM at more than one location. Other expensive components, that must be shared, such as an ALUs, can also generate structural hazards. A dynamic structural hazard occurs when two different *threads* both try to use the same location at once. Static structural hazards are resolved at compile time, using rewrites that stall one FSM at the expense of another and introducing any necessary holding registers. Dynamic structural hazards can be resolved at run-time using arbitration circuits that delay the advancement of one FSM while a needed resource is in use by another.

5 I2C Controller Example

In this section we demonstrate how a circuit that performs communication over an I2C bus can be expressed using the Kiwi library. The motivation for tackling such an example arises from the fact that the typical coding style for such circuits involves hand coding state machines using nested case statements in VHDL (or equivalent features in Verilog). In particular, the sequencing of operations is tedious and error prone. However we can exploit the built in ‘semi-colons’ of a conventional language to capture sequencing. By allowing the programmer to express control orientated tasks using the convenience of a regular language we believe Kiwi affords the designer a more productivity since the corresponding high level state machine specification can be automatically derived by our system and then efficiently implemented down to the gate level by vendor tools.

The example we use is a circuit that writes a series of values into the register of a DVI control chip on the Xilinx ML-505 board. These registers are written using an I2C interface between the FPGA and the DVI chip. The code below demonstrates what is wrong with a typical VHDL implementation for performing I2C control, which also representative of many other kinds of control code:

```
case state is
  when initial
    => case phase is
         when 0 => scl <= '1' ; sda_out <= '1' ;
         when 1 => null ;
         when 2 => sda_out <= '0' ; -- Start condition
         when 3 => scl <= '0' ;
                   index := 6 ;
                   state := deviceID_state ;
         end case ;
  when deviceID_state
     => case phase is
          when 0 => sda_out <= deviceID (index) ;
```

Such nested **case** statements are typical of situations like this where the designer ends up hand coding a nested finite state machine to capture a sequence of operations i.e. we are missing the semi-colons of an imperative language. Using Kiwi, we can more directly represent the sequencing operations e.g. in the following deserialization code.

```
public static void SendDeviceID()
{ int deviceID = 0x76;
  for (int i = 7; i > 0; i--)
  { scl = false;
    sda_out = (deviceID & 64) != 0;
    Kiwi.Pause(); // Set it i-th bit of the device ID
    scl = true; Kiwi.Pause(); // Pulse SCL
    scl = false; deviceID = deviceID << 1;
    Kiwi.Pause();
  }
}
```

The call to Kiwi.Pause() in turn makes a call to a AutoResetEvent object by waiting for an event that corresponds to the rising edge of a clock signal.

This is is processed by our system which generates Verilog RTL code. Although not designed to be human readable, the RTL can then be processed by FPGA vendor tools to generate a programming bitstream.

The generated Verilog was fed to the Xilinx ISE 9.2.03i tools and a programming netlist was produced for the Virtex-5 XC5VLX50T part on the ML-505 development board. The source program performed a series of write operations to the registers of the Chrontel DVI chip over the I2C bus. The design was then used to successfully configure the Chrontel DVI chip. The generated design used 126 slice registers and 377 slice LUTS with a critical path of 5.8ns. A hand written version will be smaller because these results do not yet reflect the use of integer sub-ranges so some registers are represented with 32-bits rather than just a hand full of bits.

6 Future Work

The techniques presented in this paper lay the groundwork for expressing higher-level concurrency and parallelism idioms in terms of lower level concurrency constructs in the form of events, monitors and threads.

Aggressive loop unwinding increases the complexity of the actions on each clock step in exchange for reducing the number of clock cycles used. Currently an unwind budget is given as a command line option but it may be worth exploring higher-level ways of guiding such space/-time trade-offs.

In software designs, threads pass between separately compiled sections and update the variables in the section they are in. This is not supported in synthesisable RTL, so instead updates to a variable from a separately compiled section must be via a special update interface with associated handshaking for write and test and set. It would be interesting to explore others mechanisms for separate compilation and composability.

One initial source of inefficient circuits was the use of `int` types in C# which resulted in circuits with 32-bit ports after synthesis. Our fix for this problem involves attaching yet another custom attribute that specifies the range for integer values which can then be used by our system to generate bit-vectors of the appropriate size in Verilog. Another approach would have been to follow the example of System-C and provide a new type that encapsulates the idea of an integer range but we felt that this would be change that permeates the whole program in a negative way.

Our hypothesis for our future work is that because we have a good translation for the low-level concurrency constructs into hardware then we should be able to translate the higher-level idioms by simply implementing them in the usual way. An interesting comparison would be examine the output of our system when used to compile join patterns and then compare them to existing work on compiling join patterns in software using Hardware Join Java [6].

Another direction to take our work is to generate code for other kinds of parallel computing resources like GPUs. It is not clear if we can continue to use the same concurrency abstractions that we have developed for Kiwi or if we need to add further domain specific constructs and custom attributes.

It may appear that our approach requires static allocation although strictly speaking our system analyzes instances of dynamic allocation (as identified by the `new` keyword) and tries to subsume them as static allocations. Future work could involve dealing with a broader class of dynamic allocations in order to make the programming model less restrictive.

A significant and perhaps optimistic assumption in our approach is that programmers can write parallel software and it is not clear that thread-level parallelism as supported by current mainstream languages is suitable for our objectives [10]. Although we have shown how to map specific uses of systems level concurrency constructs to hardware a more realistic system would provide levels of abstractions that make it easier to specify concurrency and parallelism e.g. nested data parallel arrays and their associated operations.

7 Conclusions

We have demonstrated that it is possible to write effective hardware descriptions as regular parallel programs and then compile them to circuits for realization on FPGAs. Furthermore, we have shown that we can transform programs written using conventional concurrency constructs and synchronization primitives into hardware. Specifically, we have provided translations for events, monitors, the **lock** synchronization mechanism and threads under specific usage idioms. By providing support for these core constructs we can then automatically translate higher-level constructs expressed in terms of these constructs e.g. join patterns, multi-way rendezvous and data-parallel programs.

The designs presented in this paper were developed using an off the shelf software integrated development environment (Visual Studio 2005) and it was particularly productive to be able to use existing debuggers and code analysis tools. By leaveraging an existing design flow and existing language with extension mechanisms like custom attributes we were able to avoid some of the issues that face other approaches which are sometimes limited by their development tools.

Our approach complements the existing research on the automatic synthesis of sequential programs (e.g. ROCCC and SPARK) as well as work on synthesizing sequential programs extended with domain specific concurrency constructs (e.g. Handel-C). By identifying a valuable point in the design space i.e. parallel programs written using conventional concurrency constructs in an existing language and framework we hope to provide a more accessible route reconfigurable computing technology for mainstream programmers. The advent of many-core processors will require programmers to write parallel programs anyway, so it is interesting to consider whether these parallel programs can also model other kinds of parallel processing structures like FPGAs and GPUs.

Our initial experimental work suggests that this is a viable approach which can be nicely coupled with vendor-based synthesis tools to provide a powerful way to express digital circuits as parallel programs.

References

[1] Francesco Bruschi and Fabrizio Ferrandi. Synthesis of complex control structures from behavioral systemc models. *Design, Automation and Test in Europe*, 2003.

[2] B. A. Buyukkurt, Z. Guo, and W. Najjar. Impact of loop unrolling on throughput, area and clock frequency in ROCCC: C to VHDL compiler for FPGAs. *Int. Workshop On Applied Reconfigurable Computing*, March 2006.

[3] M. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high hevel language. *8th IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[4] Rajesh K. Gupta and Stan Y. Liao. Using a programming language for digital system design. *IEEE Design and Test of Computers*, 14, April 1997.

[5] Sumit Gupta, Nikil D. Dutt, Rajesh K. Gupta, and Alex Nicolau. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. *International Conference on VLSI Design*, January 2003.

[6] John Hopf, G. Stewart Itzstein, and David Kearney. Hardware Join Java: A high level language for reconfigurable hardware development. *IEEE International Conference on Filed Programmable Technology*, 2002.

[7] IMEC. CleanC analysis tools. *Web page* http://www.imec.be/CleanC/, 2008.

[8] Celoxica Inc. Handel-C language overview. *Web page* http://www.celoxica.com, 2004.

[9] Monia S. Lam and Robert P. Wilson. Limits of control flow on parallelism. *The 19th Annual International Symposium on Computer Architecture*, May 1992.

[10] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5), 2006.

[11] W. A. Najjar, A. P. W. Bohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. *IEEE Computer*, 36(8), 2003.

[12] Rishiyur Nikhil. Bluespec SystemVerilog: Efficient, correct RTL from high-level specifications. *Formal Methods and Models for Co-Design (MEMOCODE)*, 2004.

[13] Andres Takach, Bryan Bower, and Thomas Bollaert. C based hardware design for wireless applications. *Design, Automation and Test in Europe*, 2005.

[14] Y. D. Yankova, G.K. Kuzmanov, K.L.M. Bertels, G. N. Gaydadjiev, Y. Lu, and S. Vassiliadis. DWARV: Delftworkbench automated reconfigurable VHDL generator. *17th International Conference on Field Programmable Logic and Applications*, August 2007.