Exploiting System-Level Concurrency Abstractions for Hardware Descriptions

DAVID J. GREAVES
University of Cambridge

SATNAM SINGH
Microsoft Research

February 2009

Technical Report
MSR-TR-2009-48

This technical report explores the idea of using an existing concurrent programming language and its associated tools for the compilation and debugging for *modeling* parallel computations which can be implemented on FPGAs to yield systems that significantly outperform their sequential software counterparts on conventional processors. An important application of such an approach is to make FPGA-based co-processors more accessible to software developers and other scientist because it removes the need to describe and implement parallel algorithms in terms of conventional hardware descriptions languages like Verilog and VHDL. Previous work has focused on automatically translating sequential programs into hardware which is a problem which is equivalent to automatic software parallelization. There is no known satisfactory solution for this problem. Other researchers have developed new languages or made modifications to existing languages to add special features for expressing concurrency to help model parallelism in hardware. A distinguishing aspect of our work is that we restrict ourselves to the use of an existing language and its concurrency mechanisms and libraries. By doing so we make it possible for developers to use existing compilers, debuggers and analysis tools to help develop and debug their designs. Furthermore, developers do not need to learn a new language and can rely on mature tools which are well documented. Another advantage of our approach is that it gives the developer greater control over the quality of results because the synthesized parallel architecture and communication infrastructure is directly related to the original parallel description. This allows the developer to make space/time trade-offs with greater control compared to techniques which rely on more indirect methods for influencing the structure of the output e.g. the use of pragmas.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

1 Introduction

Future microprocessors will be composed of a heterogeneous mix of processing elements which not only resemble today's processor cores but may also include specialized processors that are the evolution of today's graphics processors, special hardware for performing important functions e.g. Bayesian inference engines as well as a generalized 2D-parallel processing fabric which will be the evolution of today's FPGAs. Given that FPGA-like devices will become part of mainstream computing platforms, the task of programming such devices now becomes a challenge (and opportunity) for mainstream programmers. It would be highly desirable to find a way for software engineers to specify computationally intensive problems in an accessible way which can then be executed on parallel FPGA hardware and in particular it is desirable for programmers to think in terms of computing science abstractions rather than hardware design abstractions. Much work has already been done in the area of compiling C-like programs into gates. In this paper we describe an approach which takes the C-to-gates approach one step further by showing how it is possible to take parallel programs written in a standard modern programming language and automatically compile them to efficient circuits. The reason for starting from parallel programs rather than sequential code is to allow the programmer greater control over the architecture and performance of the generated circuit compared to techniques that start from a sequential description. The user can influence the degree of parallelism and the nature of communication in the generated circuit by creating the appropriate number of threads (each of which is mapped to a distinct group of gates) and by explicitly instantiating inter-thread communication mechanisms (e.g. channels which get mapped into FIFOs in hardware).

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). Figure 1 illustrates how our approach identifies a new part of the design spectrum by focusing on an area which is much more abstract than structural design but still leaves enough control to the programmer via threading compared to synthesis from purely sequential descriptions. It is our hope that such technology will make FPGA-based co-processor more accessible to non-FPGA or hardware experts.

The approach described in this paper uses programming language concurrency mechanisms to *model* the architecture of circuits by expressing important

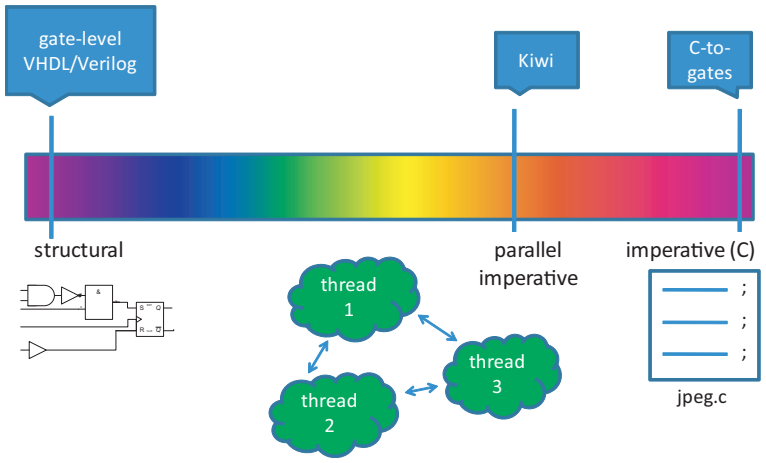


Figure 1: Kiwi relative to other approaches

aspects of their parallel behavior. However, we wish to emphasize that these parallel descriptions are usually not tenable parallel software programs that can be executed on a multi-core processor. So we are not proposing a technique for compiling the same code into efficient software and efficient hardware. The benefit of our approach is to allow scientists to express parallel computations in a programming language environment with the associated tools for debugging and verification and then automatically produce circuits which perform faster than the corresponding sequential program running on a regular processor.

In this paper we describe the architecture of our Kiwi synthesis system and present results obtained from a prototype implementation that generates Verilog circuits which are processes by Xilinx implementation tools to produce FPGA programming bit-streams.

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 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

There has been significant interest in the area of compiling circuit descriptions that look like programs automatically into circuits. Most approaches take an imperative, C-like language as a starting point and then try to work out how to

efficiently represent an equivalent sequential computation in terms of a circuit with an appropriate level of parallelism and efficient communication between sub-blocks.

The task of taking a sequential program and then automatically transforming it into an efficient circuit 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. Both procedures are ultimately limited by the level of achievable parallelism in a program which, in turn, is limited by a number of well-known programming artifacts, such as the decidability of conditional branches and array pointer comparisons.

The idea of using a programming language for digital design has been around for at least two decades [5]. Previous work has looked at how code motions could be exploited as parallelization transformation technique [10].

Examples of C-to-gates systems include Catapult-C [15] from Mentor Graphics, SystemC synthesis with Synopsys CoCentric [2], Handel-C [9], the DWARV [16] C-to-VHDL system from Delft University of Technology, single-assignment C (SA-C) [12], ROCCC [3], SPARK [6], CleanC from IMEC [8] and Streams-C [4].

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][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   }
```

Jonathan Babb's group at MIT have developed an interesting system for synthesizing sequential C and FORTRAN programs into circuit by using the notions of *small memories* and *virtual wires* [1]. Just as we make use of an existing compiler framework based on .NET and its associated compiler support infrastructure the MIT work exploits the rich SUIF framework. We believe both of these approaches are complementary to the synthesis flow that we have developed and there is no reason why both virtual wires and small memories could be incorporated into our system if they are required to reduce resource usage or improve performance.

A notable recent example of exploiting high level parallel descriptions for hardware design is the Bluespec SystemVerilog language [13] 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 standard 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 realm, to help them move computationally demanding tasks from executing on processors to implementation on FPGAs.

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.

The design of the Kiwi library tries to capture a common ground between the concurrency models and constructs used for hardware and software (see Figure 2). Our aim to is try to 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. However, although we use software concurrency mechanisms to model the parallel computations performed by hardware we do not expect these parallel programs to execute efficiently on multi-processor computers. This is because we will often express very fine grain parallelism which can be implemented effectively in circuits but which is not economic when mapped to threads of a conventional operation system. The dual design-flow nature of the Kiwi system is illustrated in Figure 3.

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

```
1  ConsumerClass consumer = new ConsumerClass(...);
2
3  Thread thread1 = new Thread(new ThreadStart(consumer.process));
4  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

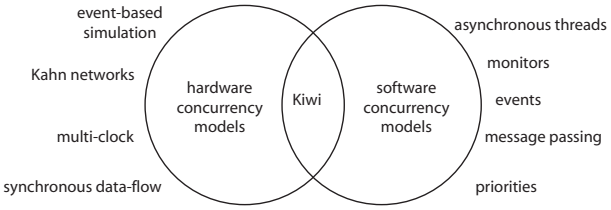


Figure 2: Concurrency models and constructs

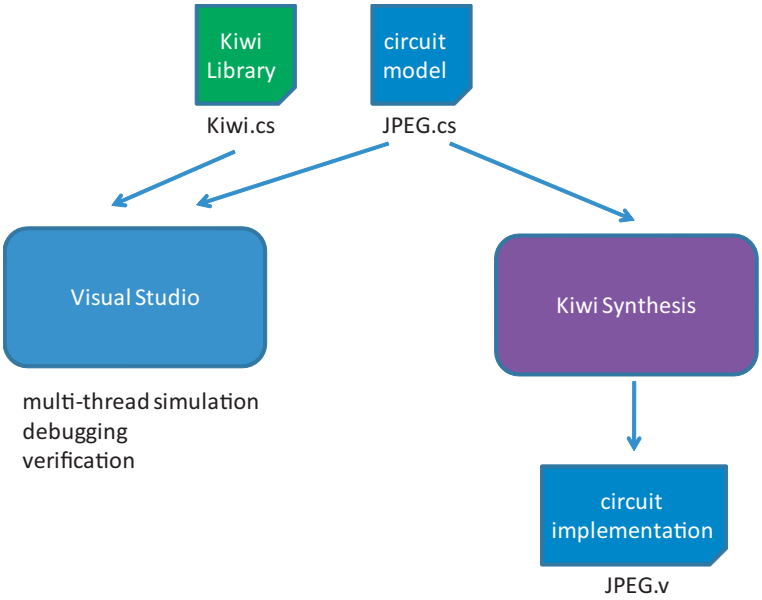


Figure 3: Kiwi descriptions as programs and circuits

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)
```

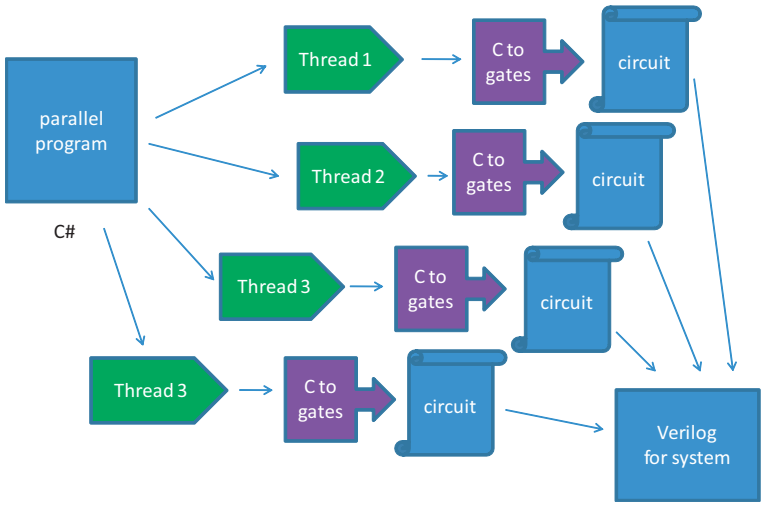


Figure 4: Synthesis of threads to circuits

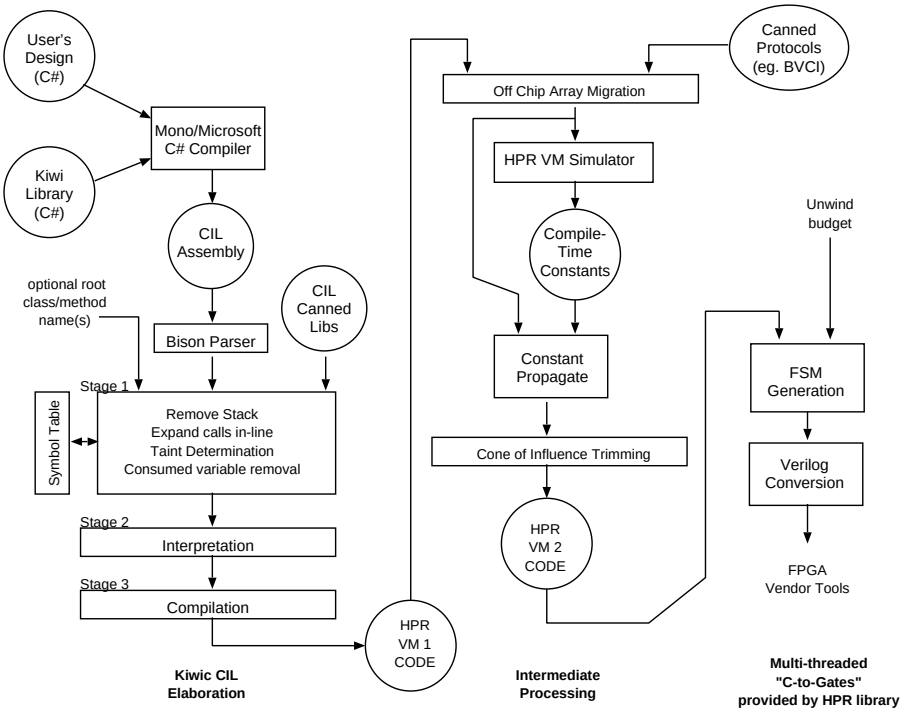
```
{ while (hpr_testandset(mutex, 1))
    hpr_pause();
}
void Exit(object mutex)
{ hpr_testandset(mutex, 0);
}
```

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.

One way to logically view the system is shown in Figure 4, which shows the original parallel program being decomposed into a static collection of threads each of which is subjected to a synthesis pass described in the following sections. The separately produced sub-circuits are then composed into a single circuit with the inter-thread communication implemented with appropriate hardware structures.


User's Design (C#)
Kiwi Library (C#)
Mono/Microsoft C# Compiler
CIL Assembly
optional root class/method name(s)
CIL Canned Libs
Bison Parser
Stage 1
Symbol Table
Remove Stack
Expand calls in-line
Taint Determination
Consumed variable removal
Stage 2
Interpretation
Stage 3
Compilation
HPR VM 1 CODE
Kiwic CIL Elaboration
Canned Protocols (eg. BVCI)
Off Chip Array Migration
HPR VM Simulator
Compile-Time Constants
Constant Propagate
Cone of Influence Trimming
HPR VM 2 CODE
Intermediate Processing
Unwind budget
FSM Generation
Verilog Conversion
FPGA Vendor Tools
Multi-threaded "C-to-Gates" provided by HPR library

4 Synthesis Flow

Our flow is shown in Figure 4. The C# source code passes through three general stages of processing and several intermediate forms before being emitted as synthesizable Verilog RTL. The first intermediate form is CIL (common intermediate language) and the subsequent forms are an internal virtual machine (VM) code. A bison parser is used to convert the textual CIL form into an abstract syntax tree (AST) as an SML data structure and the rest of the flow is implemented in Moscow ML. We now describe each stage in detail.

4.1 .NET Assembly Language Elaboration

We start by using either the Microsoft or the Mono C# compiler to convert the source code to CIL code. Although these two tools occasionally diverge in the way they handle certain details, such as the way arrays are initialized and the layout of basic blocks, they have so-far been fully interchangeable without affecting experimental results.

For illustration, we show some CIL code below. Key aspects of the CIL 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)
```

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.

Hardware description languages such as VHDL and Verilog 2000 contain constructs for generating structure at compile time. These two languages specifically use the keyword '`generate`' for this, and certain variables are specifically associated with the generate statements. On the other hand, C# programs do not necessarily possess a clear delineation between structural-generation and run-time evaluation. Another major difference between C# and RTL is the lack of dynamic-storage allocation in synthesizable RTL. Therefore, our first stage of processing, referred to as Assembly Language Elaboration, decides what to do at compile time and what to leave to run time, as well as reducing the program using a fixed number of storage variables. It totally removes the CIL stack.

We say that the elaboration process '*consumes*' a number of variables present in the input source code, including variables used only for structural generation and all object pointers and array handles. In CIL, 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 consume it in the elaboration using heuristic rules.

Actually, there is a potentially valid reason for preserving certain object and array handles through to run-time, which is where these are cycled over a finite pool of objects and arrays. However, this feature is missing in our current implementation.

The first step of processing of the AST is to form an hierarchic symbol dictionary containing the classes, methods, fields, and custom attributes. Other declarations, such as processor type, are ignored.

We have two ways of deciding which methods to convert to hardware. In the first method, a command line flag to the compiler, called `-root`, enables the user to select a number of methods or classes for compilation. The argument is a list of hierarchic names, separated by semicolons. The second method consists of a 'Kiwi.Hardware' attribute that is placed on certain classes or methods by the user to nominate them from compilation. Either way, the tool is presented with one or more thread starting points for hardware compilation. Additionally, every class in CIL has a class constructor method, that is considered to be an entry point if that class is nominated for compilation by either way. Other items present in the .NET input code are ignored, unless called from a root thread.

All procedure calls made by a thread are 'in-lined' in the elaborate stage by macro-style expansion of the CIL subroutine call instructions. This is possible because we maintain sufficient type information about what is stored in what variable to select between different overloaded implementations of methods. Each thread is symbolically evaluated using a three-stage mechanism. The first stage is a pre-processing run on each method body when the thread first enters it. It does not expand the called function bodies, whereas the second and third stages performs function body expansion.

The first stage operations on a method body eliminate the CIL stack. Symbolic tracking of expression types and code reachability is used to determine the concrete type stored in every variable and the layout of the stack at every basic block boundary. Such symbolic evaluation which is straightforward since every operator and method call is strongly typed. In our implementation of this ap-

proach, which of several overloaded method bodies is called cannot currently be controlled by run-time data, but this limitation can be removed in the future. At the entrance and exit to each basic block, load and store instructions are respectively inserted, to load and store the contents of the stack at the block boundaries into statically scoped surrogate variables, created for this purpose. The surrogate variables are frequently consumed, 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, to reduce run-time register generation the same surrogate variable instances are shared across all instances of a stack frame at the same depth of recursion. Since we have full knowledge of when a variable is potentially live, alternative methods for variable sharing could be explored in the future, such as re-using variables between stack frames that cannot be concurrently active, but registers are not at a premium in modern target technologies, such as FPGA and ASIC, and such an approach would most-likely result in slower designs owing to the multiplexors needed. The same algorithm is used for local variable allocation.

Another role played by the first stage is run-time value taint determination. Run-time input values are considered to be tainted and the algorithm propagates the taint through every operator and function call, thereby ending up with a map of which variables may possibly contain a run-time value. Those which do cannot be consumed.

The second stage and third stages of processing for a threads progress within each method respectively perform interpretation and compilation. The model is that all threads do all of their structural generation, if any, before performing any of their run-time behavior, if any. An algorithm determines a dynamic switchover point for the thread between interpretation and compilation. It maintains a fallback position for each method that is initialized to the entry point. In the interpretation phase, the CIL code is directly simulated, with concrete values being stored for each variable that is assigned in a slot in the symbol table and with no VM code being emitted. When the simulated thread encounters a basic block boundary, the fallback position is set to that point and the state of all the variables is also noted. Some threads reach the end of the method body in this way and others stop earlier because they encounter a run-time tainted value or a function or operator that cannot be simulated. In either situation, the pass switches to its second phase, by emitting a number of VM assignment statements that 'copy out' the simulated state to the run-time virtual machine, followed by a VM goto instruction to the fallback basic block exit point. For certain threads, the exit point is the thread exit point, and so there is no more to do, and for other threads, the third phase then proceeds. The third phase is a conventional compilation that converts CIL code to VM code. The parts of the method body that need converting are determined by a reachable program-counter value scan seeded from the fallback point.

The VM code runs on a so-called HPR virtual machine. This was used because a library of code from the University of Cambridge was available that includes many useful functions, including compilation of the VM code into synthesizable Verilog. An HPR machine contains internal and externally-visible

variables, imperative code sections and assertions. The variable declarations carry tag/value attributes that are interpreted by the subsequent VM compiler and are used for specifying things like signedness, wrapping and off-chip attributes (described below). One form of imperative code section consists of an array of instructions indexed by program counter variables and, by default, there is one program counter for each array. Associated with each program counter there is an option for a clock and reset net specification, although the way these are used to relate to stepping the program is not specified at this point: it is determined later on when the VM code is converted to hardware. All program counters start execution in parallel from location zero of their respective array. The VM instructions are: assign, conditional branch, fork, join, exit and calls to certain built-in functions, including hpr_testandset(), hpr_printf() and hpr_barrier(). The expressions occurring in the instructions, such as branch conditions, array subscripts, r.h.s. of assignment and function call arguments can use all common integer arithmetic and logic functions, including all of the integer 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).

The elaborate stage creates an HPR machine for each root thread. The externally-visible variable list for the HPR machine is formed from the parameter list of the methods nominated as roots and from static user variables that have been marked with Kiwi attributes. A return value from a root method is assigned to an externally-visible variable. The externally-visible variables become the I/O terminals of the generated RTL section. The internal variables are the remainder of the variables, including stack surrogate variables, the contents of heap-allocated arrays and object fields.

The C# compiler assumes that a number of libraries are present in the runtime system. These include functions for initializing arrays, accessing multidimensional arrays, forking threads and performing string operations. We have to provide implementations of all of these as 'canned' libraries. We did this by compiling suitable C# fragments to SML data structures and then pasting these into the **kiwic** source code. As explained elsewhere, certain of the canned libraries map through to hpr_xxx primitives whereas other trigger specific behavior during CIL elaboration. For instance, new user threads are enabled by trapping the 'Thread.Start()' library call.

We have defined attributes for marking certain C# methods as assertions to be included in the HPR machine's assertion list, but these mechanisms are beyond the scope of this paper.

Using C# attributes applied to classes and fields the user can influence the hardware that is generated. He can control the width of registers, the names of clock domains, which signals are input/output connections, how memories are implemented and various other details (which are being described in a user manual).

A synchronous circuit designed with kiwi requires a clock and reset input. A default clock domain exists and the default net names `clock` and `reset` are

automatically generated. To override the default names, or when more than one clock domain is used, the '`ClockDom`' attribute is used to mark up a root method, giving the clock and reset nets to be used for activity generated by that method.

```
[Kiwi.ClockDom("clknet1", "resetnet1")]
public static void Work1()
{ while(true) { ... } }
```

A root method may have at most one clock domain annotation but unannotated methods can be called from various clock domains. These annotations are passed on as tags to the HPR imperative code array.

Integer variables of width 1, 8, 16, 32 and 64 bits are native in C# and CIL but hardware designers frequently use other widths. We support declaration of registers with width up to 64 bits that are not a native width using an '`HwWidth`' attribute. For example, a five-bit register is defined as follows.

```
[Kiwi.HwWidth(5)] static byte fivebits;
```

When running the generated C# naively as a software program (as opposed to compiling to hardware), the width attribute is ignored and wrapping behavior is governed by the underlying type, which in the example is a byte. The HPR machine supports variable declarations that have both an enumeration range that controls when they will actually wrap and a secondary range that the subsequent VM compiler just uses for checking. The VM compiler performs a conservative data-flow analysis for the reachable ranges of all variables and flags a compile-time error if there is any chance that the variable will wrap differently in hardware from software.

Object-oriented software sends threads between compilation units to perform actions. Synthesizable Verilog and VHDL do not allow threads to be passed between separately compiled circuits: instead, additional I/O ports must be added to each circuit and then wired together at the top level. Accordingly, we mark up methods that are to be called from separate compilations with a remote attribute.

```
[Kiwi.Remote("parallel:four−phase")]
public return_type entry_point(int a1, bool a2, ...)
{ ... }
```

When an implemented or up-called method is marked as '`Remote`', a protocol is given and **kiwic** generates additional I/O terminals on the generated RTL that implement a stub for the call. The currently supported protocol for remote calling is asynchronous, using a four-phase handshake and a wide bus that carries all of the arguments in parallel. Another bus, of the reverse direction, conveys the result where non-void. The remote-calling facilities were easy to implement: the user's code is placed inside an infinite loop with top and tail code added to synchronize with the external control signals and handle data transfer.

4.2 Intermediate Processing

The output from CIL elaboration is a multi-threaded virtual machine. In our intermediate processing stage, we preserve this structure while making a number of rewrites to simplify the code, share resources and provide off-chip arrays.

By default, arrays allocated by the C# code pass through our tool chain and convert directly to simple Verilog array definitions. These typically compile to on-chip RAMs using today's FPGA tools, but there is also frequently a need to map larger arrays into off-chip SRAM or DRAM banks. Any memory subsystem is limited in terms of the number of ports it has and the simultaneous number and type of transactions possible at any one port, so there can be a structural hazard if too many reads or writes need to be active in one RTL clock cycle. We define static and dynamic structural hazards based on the notion that we do not generally know at compile time what the relative alignment of separate threads might be at run time. We define a static structural hazard to be when the code for a single thread attempts too many operations at once and a dynamic structural hazard to be when two different threads both try to use the same resource at once, exceeding its ports. In general, other component that must be shared, such as expensive ALUs, can also generate structural hazards. We explain our approach to overcoming these hazards and giving control over resource sharing. Static hazards become resolved at compile time, owing to the symbolic evaluation applied to the mutex variables in the HPR library, whereas for dynamic hazards, the operations on the mutex variables cannot generally be evaluated at compile time and so these variables appear in the generated RTL. At the level of the RTL tool-chain, all '*threads*' are combined into one Verilog '`always`' block for each clock domain, since it is not allowed within the definition of synthesizable RTL for any register to be written by more than one thread. Therefore, the difference between our static and dynamic resolution is not apparent at the level of the RTL tool-chain.

To use off-chip RAMs, we provide the kiwi 'OutboardArray' attribute to cause an array declaration and access operations to be replaced with a set of external connections to a memory subsystem outside of the current compilation. The user then wires up an external SRAM, DRAM or cache port. In the current system, a number of outboard ports may be declared, each of which supports read and write operations and has a data width and address range. The port name and offset within the port for mapping a particular array are given in the kiwi attribute next to the array declaration in the C# program. The net-level protocol to use for each port and number of operations in progress it will support are given on the command **kiwic** line. The example below maps a pair of user arrays into the same external memory using the same port at user-defined offsets. We are still finding out how to make it as easy as possible to map the user's arrays into RAMs. Within the current mechanism we can at least flag whether or not the mappings overlap. However, a more-automatic mechanism is going to be preferable in the long run.

```
[Kiwi.OutboardArray("portx", 0)]
static short [] PA = new short[32768];
```

```
[Kiwi.OutboardArray("portx", 32768)]
static short [] PB = new short[32768];
```

Off-chip arrays are implemented by rewriting all the VM instructions that access the array. A write access is made by an assignment statement. This is replaced with a call to a canned macro library code that implements a write transaction on the external port for the required protocol

$A[e'] := e$ is replaced with `hpr_array_write`(A, e', e)

and read accesses occurring in any expression are replaced with the names of freshly-created holding registers that are loaded by calls inserted before the instruction

$v := A[B[e]]$ is replaced with
`hpr_array_read`$(A, e, h1)$; `hpr_array_read`$(A, h1, h2)$; $v := h2$

where $h1$ and $h2$ are the holding registers. The inserted read calls are sorted into an order whereby no holding register is used before it is written. The canned library code is then expanded in place.

With multi-threaded user code, the read and write functions can potentially be re-entrant, but generally external bus protocols can only support a finite number of transactions open at one time. The maximum is typically only one for simple protocols. It is easy to cater for these constraints in our approach since the transaction is inserted into the user's thread, and so the number of outstanding operations is easily bounded by inserting further code that blocks the thread by spinning on a mutex with `hpr_testandset` calls at the entry point to the library code and freeing the mutex on exit. Providing a write-posting mechanism, that does not block the user threads on writes, just requires that the inserted code has its own thread to implement the actual write call.

We have implemented BVCI [14] and four-phase handshake as alternative canned protocols for off-chip arrays and remote procedure call, but, as with the other canned library code, there is no reason why these protocols should not be read in from separate C# description files if desired. In the fullness of time, we expect to support a range of protocols that are compatible with standard on-chip busses.

Off-chip arrays are the obvious example of components that offer structural hazards (i.e. they have limited accessibility in terms of concurrent user threads), but in the future, other resources described in the C# input or in other ways, such as a complex ALU module or a subroutine that should not be inlined in two different threads can also be shared between threads if suitably declared. For a C# subroutine we can provide an attribute that makes it a shared resource, rather than a resource in-lined in each thread. Bluespec has a facility of this nature, called the '*FSM server*' [13]. This will provide a flexible and elegant way for an engineer to choose whether to use time (clock cycles) or space (silicon area) to solve his problem.

After the rewrites that multiplex access to shared resources and off-chip resources, the constructor methods are executed by simulation, with any inputs

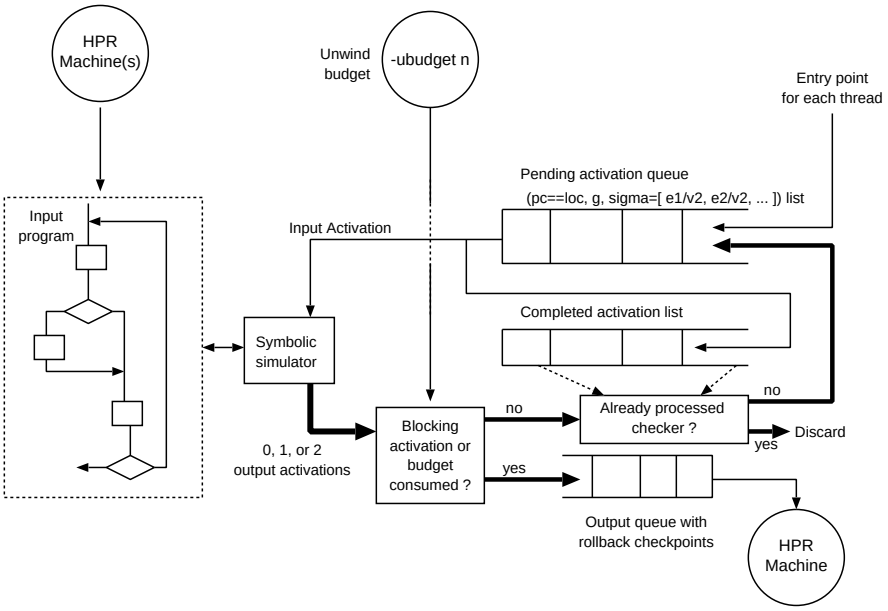


Figure 5: Conversion of control flow graph to FSM.

(free variables) set to don't know. However, constructor methods do not commonly have inputs, apart from constant values passed from parents as formals. 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 consumed by a constant propagation function (Figure 4.) Finally, a cone-of-influence logic trim is performed, whereby assignments to variables that have no ultimate effect on any output or side-effecting system call, such as `writeline`, are removed. Although such trimming is aways ultimately performed by the backend RTL synthesis tools, our purpose here is to reduce the size and vocabulary of the HPR code that is to be converted to RTL.

4.3 FSM Generation

The input to the FSM generation stage is an HPR machine with its executable code held in an array for each root thread and user-forked thread. In this section we explain how the HPR library converts this form of program code to Verilog RTL.

Each instruction is either an assignment, exit statement, built-in primitive or conditional branch. We have not used the fork and join instructions supported by the HPR library since only static thread creation has been supported in the **kiwic** front end, but in the future we will use them to process C# programs that contain dynamic thread creation or joins. 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 from FSM generation is an HPR machine where the imperative code consists of an HPR parallel construct for each clock domain. The parallel construct contains a list of finite-state-machine edges, where edges have three possible forms:

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

where the first form assigns `e` to `v` when `g` holds, the second assigns to a named array in a similar way and the third calls built-in function `f` when `g` holds. All .NET arrays are single-dimensional with multi-dimensional arrays being folded down within the canned libraries.

An additional input, from the command line, is an unwind budget: a maximum number of basic blocks to explore in any loop unwind attempt. 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 5), 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 undone. The pending activation queue is initialized with the entry points for each thread.

Compilation uses a symbolic simulator function, denoted as $SS[\![\ C\ ]\!]_{(n,g,\sigma)}$ that evaluates command C from address n of the code array according to its denotational semantics, as given in Table 1. This uses the symbolic expression evaluator function $[\![\ \ ]\!]_\sigma$ that rewrites the AST for an expression using values from σ and performing evaluation of compile-time manifestly constant expressions. The expression $[e/v]\sigma$ denotes a modified version of the environment where variable v is set to expression e. 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

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

Table 1: Semantic Rules for the HPR Imperative Code

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.

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. The normal-form normalizer uses rules to eliminate common operators, to sort the arguments of commutative operators and adjust the nesting of associative operators. For instance, $a > b$ is always represented as $b < a$, $b + a$ is always held as $a + b$ and $(a + b) + c)$ is held as $a + (b + c)$. 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, resolves 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.) Any boolean variable can be used as a mutex. 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:

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

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. Sometimes, this final value is always the same, allowing the hardware register to be eliminated, with the arbitration completed fully at compile time.

The run-time semantics for the three types of finite-state machine edge are that all of the edges whose guards hold are executed in parallel on the clock edge, with no assignment visible in any expression until they are all completed. This directly corresponds to Verilog's non-blocking assignments and the signal assignment found in VHDL. Therefore, all three types of finite-state machine edge are readily converted to synthesizable RTL. However, we have found that some, even quite small, examples can exceed the capabilities of certain FPGA tools if rendered directly in RTL. Therefore, a pair of optimizers are used that collate all of the guard expressions for a given assigned variable and simplify them using Espresso in conjunction with a 1-D linear-programming package that implements examples such as the following

$$\begin{array}{lcl} x < 4 \ \&\& \ x < 6 & \rightarrow & x < 4 \\ 4 < x \ \&\& \ x < 6 & \rightarrow & x == 5. \end{array}$$

5 Producer Consumer Example

This section presents a small example of two communicating threads which are synthesized into a circuit. The following section presents a more realistic example of a filter circuit. However, before tackling a more sophisticated example we describe in detail an example built using threads and the one place channels described in the previous sections to build an example of a producer/consumer scenario which is a common idiom for channel based concurrent systems.

The example in this section comprises two threads: a producer thread which generates the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 and then stops; and a consumer thread which continually reads integer values from a channel and outputs their double on an output channel. The two threads are joined by a shared channel as shown in Figure 6.

This circuit is represented by a collection of methods in a class ProducerConsumerExample which are used to spawn off threads plus other declarations to define the ports of the circuit and the channels used for inter-thread communication. For example, here is the portion of the code that specifies the output to be an integer port and which also declares and creates the two channels used for communication between the threads and the main program.

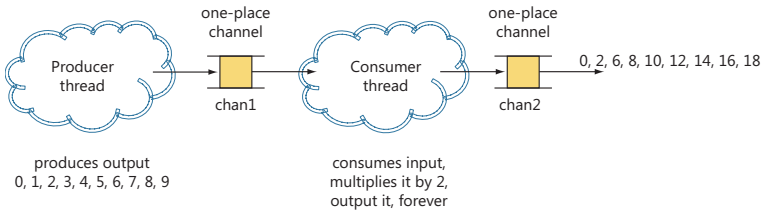


Figure 6: A producer/consumer scenario

```
class ProducerConsumerExample
{
    [Kiwi.OutputIntPort("result")]
    public static int result;

    static Kiwi.Channel<int> chan1 = new Kiwi.Channel<int>();
    static Kiwi.Channel<int> chan2 = new Kiwi.Channel<int>();
```

The two channels that are created are exactly the same one-place channels described in the previous sections. Note that all of the declarations in this class have so far been of static fields.

The producer is described by a thread which is an instantiation of the following static method.

```
public static void Producer()
{
    for (int i = 0; i < 10; i++)
    {
        chan1.Write(i);
        Kiwi.Pause();
    }
}
```

The producer writes out ten values and then stops. The values are written to the shared channel chan1 and the writing of values is sequenced to synchronize with an implicit clock by called Kiwi.Pause();.

The consumer is another static method which runs forever.

```
public static void Consumer()
{
    while (true)
    {
        int i = chan1.Read();
        chan2.Write(2 * i);
        Kiwi.Pause();
    }
}
```

The consumer reads values from the shared chan1 (which is populated by

the producer thread) and then writes the double of the read value to the output channel chan2.

The top level circuit description instantiates the producer and consumer threads and then reads the result values from chan2 which are used to drive the result output.

```
public static void Behaviour()
{
    Thread ProducerThread = new Thread(new ThreadStart (Producer));
    ProducerThread.Start();

    Thread ConsumerThread = new Thread(new ThreadStart(Consumer));
    ConsumerThread.Start();

    while (true)
    {
        Kiwi.Pause();
        result = chan2.Read();
        Console.Write(result + " ");
     }
```

When this program is compiled and run on the command line or in the Visual Studio IDE it produces the expected output values.

```
>ProducerConsumerExample
0 2 4 6 8 10 12 14 16 18 ^C
```

The consumer executes indefinitely so this execution of the program has been terminated with a control-C signal.

6 Filter Example

This section demonstrates how a filter can be designed as a collection of communicating threads. First, we describe a 5-tap filter without using any threads other than the main program. This produces a filter with five multipliers and a combinational adder tree. Later we shall show how a semi-systolic filter can be designed with multiple threads.

The specification of the filtering operation we describe and implement in this section is shown below.

$$y_t = \sum_{k=0}^{N-1} a_k x_{t-k}$$

The code to implement a simple finite impulse response filter as described above is shown below as a static method in C#.

```
public static int[] SequentialFIRFunction (int[] weights, int[] input)
{
    int[] window = new int[size];
    int[] result = new int[input.Length];
```

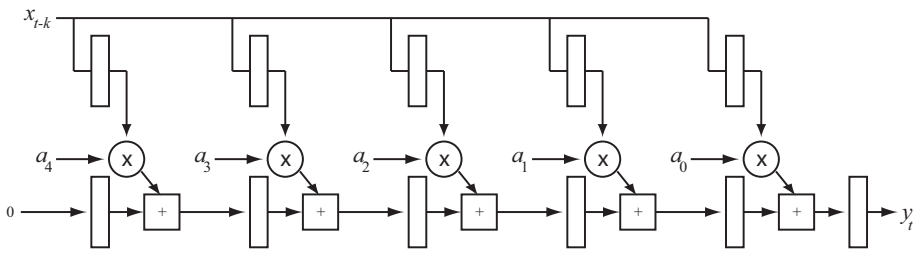


Figure 7: A transposed multi-tap filter

```
5
6       // Clear to window of x values to all zero.
7       for (int w = 0; w < size; w++)
8           window[w] = 0;
9
10      // For each sample...
11      for (int i = 0; i < input.Length; i++)
12      {
13          // Shift in the new x value
14          for (int j = size - 1; j > 0; j--)
15              window[j] = window[j - 1];
16          window[0] = input[i];
17
18          // Compute the result value
19          int sum = 0;
20          for (int z = 0; z < size; z++)
21              sum += weights[z] * window[z];
22          result[i] = sum;
23      }
24
25      return result;
26  }
```

Note that this code has no explicitly sequencing through calls to Kiwi.Pause() and there is no inter-thread communication. This code can be synthesized into a circuit which fairly directly implements the logic above with the loops unrolled to yield five multipliers.

A much better way to make a filter is to use 5-taps with registers between the taps to yield either a semi-systolic or systolic filter which will have a much better throughput than the one produced from the design above and which will also not suffer from a long combinational critical path. Furthermore the filter can be transposed to allow the input samples to be broadcast to each stage. Such a design is illustrated in Figure 7.

The first design decision we make is to represent each tap of the transposed filter with one thread. This will not result in an efficient software implementation but this decision does allow us to express the idea that we want to build a

filter using N parallel stages which then does result in fast parallel hardware.

A static method can be defined which can be instantiated several times to create multiple tab threads. Each tap-thread is passed in its weight, a channel to read its x sample value from, a channel to read the sum of the previous multiply-add operations and a channel to write out the result. Each tap tread contains an infinite loop which repeatedly consumes values from the input channels and writes results to the output channel. Synchronization occurs implicitly through the use of the read and write methods of the channel class.

```
1 static void Tap(int a, Kiwi.Channel<int> xIn, Kiwi.Channel<int> yIn,
2                 Kiwi.Channel<int> yout)
3 {
4   int x;
5   int y;
6   while(true)
7   { y = yIn.Read();
8     x = xIn.Read();
9     yout.Write(x * a + y);
10  }
11 }
```

In the description shown above the reads from the yIn and xIn channels may occur sequentially. We could have explicitly specified that the reads are concurrent by spawning off a thread for one of the reads and then joining on it and this will schedule the read operations within the same clock cycle. However, this is rather clumsy in C# and this is case where having a language level par block is useful (e.g. as is done in Handel-C). However, we believe this problem can be alleviated through the use of a join pattern which expresses the notion of reading from multiple channels atomically. It is possible to implement join patterns as a library in C# without changing the compiler or runtime.

The filter architecture shown in Figure 7 can now be modeled by instantiating the tap thread multiple times with the appropriate channels between the threads and the addition of some extra threads to provide the zero input along to the y chain of channels.

```
1 static void ParallelFIR(int size, Kiwi.Channel<int> xin, Kiwi.Channel<int> yout)
2 {
3     Kiwi.Channel<int>[] Xchannels = new Kiwi.Channel<int>[size];
4     Kiwi.Channel<int>[] Ychannels = new Kiwi.Channel<int>[size + 1];
5
6     // Create the channels to link together the taps
7     for (int c = 0; c < size; c++)
8     {
9         Xchannels[c] = new Kiwi.Channel<int>();
10        Ychannels[c] = new Kiwi.Channel<int>();
11        Ychannels[c].Write(0); // Pre-populate y-channel registers with zeros
12    }
13    Ychannels[size] = new Kiwi.Channel<int>();
14
15    // Connect up the taps for a transposed filter
```

```
16      for (int i = 0; i < size; i++)
17      {
18          int j = i;
19          Thread tapThread = new Thread(delegate()
20                          { Tap(j, weights[j], Xchannels[j], Ychannels[j], Ychannels[j+1]); });
21          tapThread.Start();
22      }
23
24      // Broadcast the input
25      Thread broadcast = new Thread(delegate() { BroadcastInput(xin, Xchannels); });
26      broadcast.Start();
27
28      // Insert an infinite sequence of zeros into the first Y channel stage
29      Thread zeroYs = new Thread(delegate() { ZeroFirstY(Ychannels[0]); });
30      zeroYs.Start();
31
32      // Drive yout
33      int yresult;
34      while (true)
35      {
36          yresult = Ychannels[size].Read();
37          yout.Write(yresult);
38      }
39  }
```

The top-level inputs and outputs of the circuit are represented by integer ports. The class that defines the transposed convolver starts with the port declarations and a definition of the weights.

```
1  class ParallelConvolver
2  {
3      const int size = 5;
4      static int[] weights = new int[size] {2, 5, 6, 3, 1} ;
5
6      [Kiwi.InputIntPort("sample")]
7      public static int sample;
8
9      [Kiwi.OutputIntPort("result")]
10     public static int result;
```

We may also have explicit control over the bit-vector representation of an output port e.g to create a 32-bit bit-vector in the generated Verilog instead of an integer port we could write:

```
1  [Kiwi.OutputWordPort("result", 31, 0)]
2  public static int result;
```

Finally the top level definition of the filter is a static method that consumes sample values every tick from the input and pumps them into the filter and which also consumes a value from the filter and writes it to the output port. This is the method that is nominated as the 'root' method to the Kiwi tools for the generation of a Verilog netlist.

```
static void FIRtop()
{
    // Create channels to allow the main program to communicate with the circuit
    Kiwi.Channel<int> xin = new Kiwi.Channel<int>();
    Kiwi.Channel<int> yout = new Kiwi.Channel<int>();

    // Create a thread to filter a single channel.
    Thread filterChannel = new Thread(delegate() { ParallelFIR(xin, yout); });

    // Perform the parallel filtering.
    filterChannel.Start();

    while (true)
    {
        xin.Write(sample);
        Kiwi.Pause();
        result = yout.Read() / sumOfWeights;
    }

}
```

For the purposes of simulation with the Microsoft Visual Studio IDE we can define the main method to feed in some data values and write out the results.

```
public static void GenerateInput(Kiwi.Channel<int> xin, int[] inputs)
{
    for (int k = 0; k < inputs.Length; k++)
       xin.Write(inputs[k]);
}

static void Main(string[] args)
{
    int[] inputs = new int[16] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 };

    // Create channels to allow the main program to communicate with the circuit
    Kiwi.Channel<int> xin = new Kiwi.Channel<int>();
    Kiwi.Channel<int> yout = new Kiwi.Channel<int>();

    // Create a thread to filter a single channel.
    Thread filterChannel = new Thread(delegate() { ParallelFIR(xin, yout); });

    // Perform the parallel filtering.
    filterChannel.Start();

    // Generate the inputs
    Thread generateInput = new Thread(delegate() { GenerateInput (xin, inputs); });
    generateInput.Start();

    // Write out the first 10 results.
    for (int i = 0; i < 10; i++)
    {
```

```
28          result = yout.Read();
29          Console.Write("{0} ", result);
30      }
31      Console.WriteLine();
32  }
```

This description can be executed directly inside Visual Studio and a new top-level program can be written to write out the results. Alternatively, the same code can be processed by our Kiwi system to generate the corresponding semi-systolic circuit. We believe the ability to specify circuit structure through the explicit use of threads in an existing language and library for controlling circuit architecture is a novel and useful feature.

The sequential filter code was used for the kernel of a program for convolving Windows BMP images and we instrumented its performance. On a dual-core Pentium Q6700 system running at 2.67GHz with 3GB of memory the sequential code could process 6,562,500 pixels per second. We measured only the time taken for the kernel operation on the image in memory and not the time taken to read or write images to the disk.

The parallel software version of the kernel which used a separate filter thread for each of three three color channels operated at 10,467 pixels per second which gives an indication of how poorly very fine grain parallelism maps onto a conventional multi-core architecture. The FPGA version has a critical path of 7.093ns on a XC5VLX50T-1 part can operate at 141MHz. The handshaking protocol that we synthesize means that it takes four cycles to process a sample so this circuit operates at 35,000,000 pixels per second. The generated Verilog produces a circuit which is mapped into 359 slice LUTs and 4 DSP48E blocks (we believe the insertion of another register could help the synthesis tools map the remaining filter tap stage into a DSP48E block). A similar filter generated using Xilinx's core generator which makes aggressive use of DSP48E blocks and pipelining operate at around 400MHz. We generated a similar transposed systolic filter using Core Generator

On the BEE3 RAMP board the DRAM memory controller delivers 288-bits for each read operation so we can process 12 8-bit pixels in each clock ticks. This increases the perform to 429,000,000 pixels per second if we instantiate 12 banks of filters (with 3 filters per bank for each color channel). There is significant room for improvement e.g. by optimizing the implementation of the handshaking protocol (or totally removing it through aggressive analysis) and by further pipelining.

In conclusion our prototype system can produce a convolver from a parallel program which operations 3,000 times faster (on the ML-505 board) or 40,000 times faster on the BEE3 system than the corresponding sequential program. However, compared to an optimized filter from Xilinx's Core Generator our system is ten times slower. This supports our thesis that our approach can help to significantly speed up certain kinds of computations compared to their sequential software counterparts however we do not aim to match the speed of hand crafted designs.

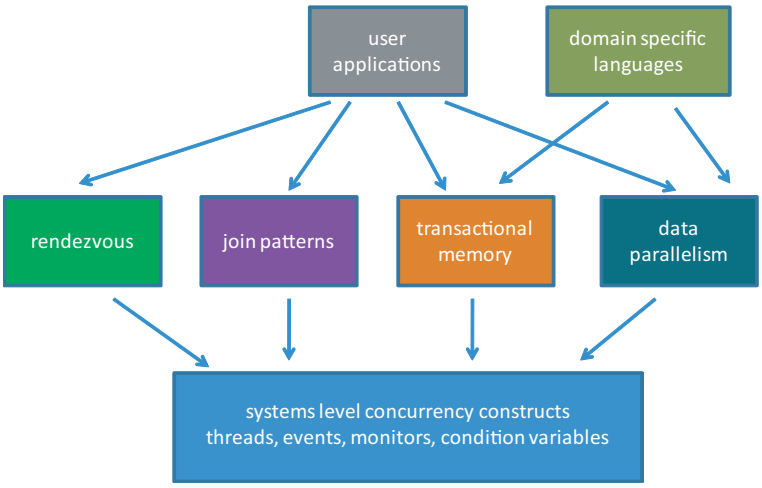


Figure 8: Higher level concurrency abstractions

7 Future Work

This paper describes how certain uses of system-level concurrency constructs may be compiled into circuits. In the next steps of the project we aim to implement higher-level concurrency abstractions in terms of the low-level constructs that we have already implemented. The objective is for user applications to use these higher-level abstractions rather than the low-level mechanisms that we have described in the previous sections. Figure 8 also shows that we hope to layer domain-specific languages on top of higher-level concurrency abstractions like software transactional memory, join patterns and data parallelism.

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 we are exploring higher-level ways of guiding such space/time trade-offs, including allowing the user to nominate objects and methods that are to be shared between threads rather than having fresh allocations for each thread.

In software designs, threads pass between separately compiled sections and update the variables in the section they are in. This is not supported in synthesizable RTL, so instead updates to a variable from a separately-compiled section must be via a special update interface with associated handshaking protocol. This neatly mirrors contemporary programming style in OO languages such as C#, where direct access to an object's internal state is avoided with preference for a variety of accessor methods that may read or update more than one variable. 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 a custom attributes that specify the bit-width or integer sub-range for integer values that can then be used by our system to generate bit-vectors of the appropriate size in Verilog. Integer sub-ranges can be used as assertions about the reachable state space of the design, thereby removing the need to accurately preserve the behavior of the program outside its naturally-reachable state space and also providing a source of dont-cares for hardware optimization. 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 a 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 to 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 [7].

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. For example, Figure 9 demonstrates how we may apply shape analysis and separation logic to automatically transform a program that uses a linked list into a program that uses a statically allocated array.

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 [11]. 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.

8 Conclusions

Although it may not seem possible at first sight we have shown that system level concurrency constructs can be synthesized into circuits and this can be used as the basis of an approach for compiling parallel programs into circuits. 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

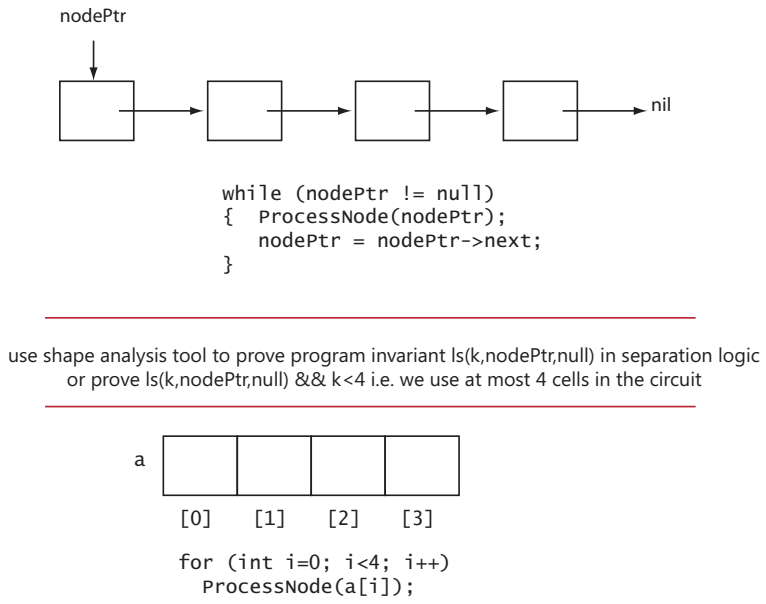


Figure 9: Using shape analysis to convert a linked list program into an array program

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 leveraging 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.

Further work which exploits recent results in shape analysis and separation logic give us the possibility of taking programs that use dynamic memory allocation and then automatically transform these programs into their array equivalents. Such a technique would greatly extend the utility of an approach that aims to take regular parallel programs written by software engineers and convert them into efficient circuits.

References

[1] Janthan Babb, Martin Rinard, Csaba Andras Moritz, Walter Lee, Matthew Frank, Rajeev Barua, and Saman Amarasinghe. Parallelizing applications into silicon. *7th IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

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

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

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

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

[12] 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.

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

[14] OCPIP. Open Core Protocol Specification Release 1.0. *Web page http://www.ocpip.org*, 2001.

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

[16] 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.