Designing Application Specific Circuits with Concurrent C# 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—**This paper presents an investigation into the possibility of using a regular concurrent programming language for modeling and implementing digital circuits. Some of the reasons for using an existing language include the ability to use existing compilers and analysis tools for circuit design and verification. Another important reason is the ever increasing need to model complete systems that comprise interacting software and hardware in a single framework which facilitates easier migration of sub-components between hardware and software implementations compared to multi-model approaches. To this end we present the design of the Kiwi system which models digital circuits with concurrent programs using a standard library in C# for multi-threaded programming. Kiwi models can be executed using a regular C# compiler. Also, the compiled bytecode can be automatically converted into circuits using our Kiwi hardware synthesis system.**

I. Introduction

To what extent can we use a regular programming language to model and implement digital circuits? This is an interesting question because the ability to exploit an existing language for hardware design confers upon us many advantages including the use of regular, off-the-shelf compilers; the use of existing training material e.g. books and tutorials; the use of sophisticated tools for profiling, debugging and verification; and the possibility of supporting a hardware/software co-design framework based on a single language that facilitates shifting the hardware/software boundary. A key aspect of modeling digital circuits is the ability to express parallel behaviour through appropriate language constructs or other abstractions provided by a library interface. Although many languages have direct support for expressing parallel behaviour through the use of concurrent multi-threaded language constructs or library modules these mechanisms are rarely used because they are considered too heavyweight. This is because concurrent programs that use abstractions like locks and monitors provide an economic way of writing coarse-grain programs but they are hard to use for the efficient execution of very fine grain parallel systems like digital circuits.

Some systems do use very lightweight concurrency mechanism to facilitate circuit modeling e.g. on Windows there is an implementation of SystemC that uses very lightweight fibers which are user scheduled onto operating system threads. However, the user-level concurrency abstraction provided by SystemC is still very closely related to the process, shared-variable and sensitivity list model found in VHDL and Verilog. We argue that aspects of a circuit's parallel behaviour should be *directly* modeled by language and system-level threads which leads to clearer descriptions and exploits existing features and tools to support concurrent programming.

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 concurrent multi-threaded 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 aspects of their parallel behavior. As compiler technology matures, we might ultimately expect the same parallel expression of an algorithm to be efficiently converted both for execution on a multi-core processor and on FPGA, but in our current work, we expect the programmer to insert more parallelism than might

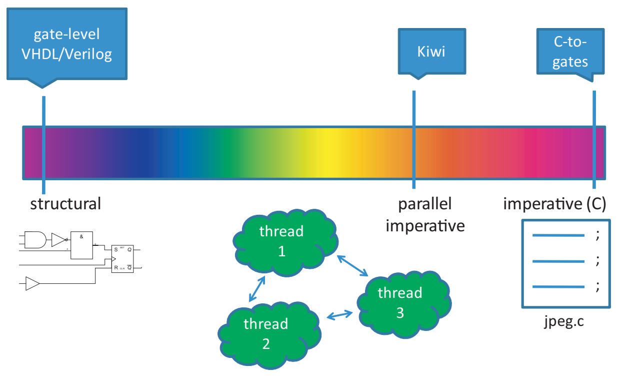


Figure 1. Kiwi relative to other approaches.

be best for today's multi-core processors. Running the same program on a multi-core processor can then be thought of as a multi-threaded simulation of the target 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 program running on a regular processor. So we make FPGA-based co-processing more accessible to domain experts in other areas (scientific computing, biology etc.) that have a secondary skill in programming/hardware to allow them to exploit FPGA-based co-processors without detailed knowledge of the FPGA hardware design or synthesis flow.

Fine-grained parallelism running on multi-core processors (CMPs) suffers synchronization overheads compared with the synchronous, globally clocked FPGA. Conventional approaches have tended to address this by reducing the frequency of interaction between tasks or using SIMD instructions. Where task run time and communication patterns are predictable at compile time the FPGA solution should ultimately prevail since no run-time flow control overheads exist.

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

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.

II. 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 [1]. Previous work has looked at how code motions could be exploited as parallelization transformation technique [2].

Examples of C-to-gates systems include Catapult-C [3] from Mentor Graphics, SystemC synthesis with Synopsys CoCentric [4], Handel-C [5], the DWARV [6] C-to-VHDL system from Delft University of Technology, single-assignment C (SA-C) [7], ROCCC [8], SPARK [9], and Streams-C [10].

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).

```
par
  { a[0] = A; b[0] = B;
    c[0] = a[0][0] == 0 ? 0 : b[0] ;
    par (i = 1; i < W; i++)
    { a[i] = a[i−1] >> 1 ;
      b[i] = b[i−1] << 1 ;
      c[i] = c[i−1] + (a[i][0] == 0 ? 0 : b[i]);
    }
    *C = c[W−1];
  }
```

IBM's *Liquid Metal* co-synthesis system [11] requires the programmer to use specific enumeration types with write once restrictions and it automatically synthesizes the software and hardware components and the interfaces between them. In our approach, certain methods or classes from the source code are manually marked with a C# Kiwi.Hardware() attribute to denote that they should be implemented in FPGA. Automatic generation of interfaces can then follow the same pattern, but we allow free use of all .NET value types and these can be annotated with further attributes to trim implementation register widths. We report compile-time errors if a custom-width hardware register may wrap in a different way from the .NET value type. Like the

'SIR' intermediate language of Liquid Metal, we generate a structure that is the union datapath of all of the VLIW-style operations performed by a thread.

The SpC compiler [12] uses '*points-to*' analysis to partition addressable objects into separate memories. We do the same, with the proviso that the heap has the self-same structure on each iteration of a non-unwound loop. SpC also implements hoisting of loads to reduce control flow hazards. Jonathan Babb's group at MIT has developed an interesting system for synthesizing sequential C and FORTRAN programs into circuits by using the notions of *small memories* and *virtual wires* [13]. These approaches are complementary to the Kiwi front end processing and can be orthogonally incorporated in the scheduling and mapping stages of the back end processing to achieve the same expected benefits. Similarly to how we make use of an existing compiler framework based on .NET and its associated compiler support, the MIT work exploits the rich SUIF framework.

A notable recent example of exploiting high-level parallel descriptions for hardware design is the Bluespec SystemVerilog language [14] 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 defines 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 synthesisable SystemC descriptions may lead to very efficient circuits, they still require the designer to think like a digital circuit engineer: the designer must explicitly implement all of the handshaking wires between components or else keep a mental model of when each shared variable is read or written. 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.

III. 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

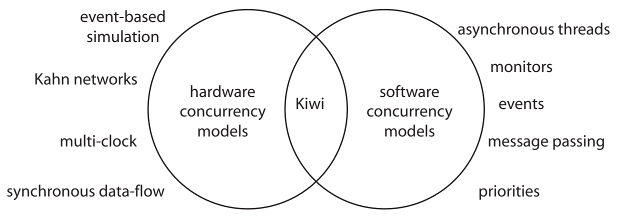


Figure 2. Concurrency models and constructs.

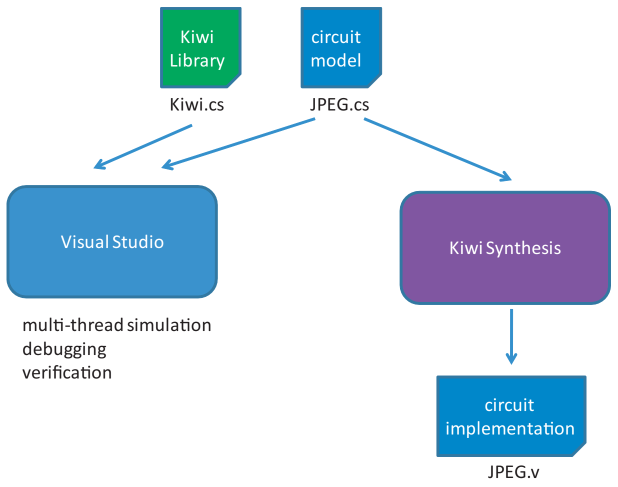


Figure 3. Kiwi descriptions as programs and circuits.

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:

```
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.

```
public class channel<T>
{ T datum;
  bool empty = true;
  public void write(T v)
  { lock(this)
    { while (!empty)
        Monitor.Wait(this) ;
      datum = v ;
      empty = false ;
      Monitor.PulseAll(this);
    }
  }

  public T read()
  { T r ;
    lock (this)
    { while (empty)
        Monitor.Wait(this);
      empty = true;
      r = datum;
      Monitor.PulseAll(this);
    }
    return r;
  }
}
```

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.

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.

IV. Synthesis Flow

In our synthesis flow C# source code passes through three general stages of processing and several intermediate forms before being emitted as synthesisable 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 start by using either the Microsoft or the Mono C# compiler to convert the source code to CIL code. Although

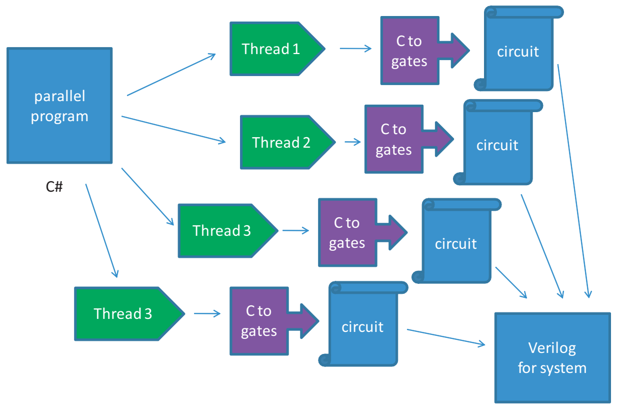


Figure 4. Synthesis of threads to circuits

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 structure of the heap, in terms of number of objects and their type, must be constant at each iteration of any loop not unwound 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 synthesisable 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 to use a fixed number of storage variables. It totally removes the CIL stack.

We say that the elaboration process '*subsumes*' a number of variables present in the input source code, meaning that they do not need to be represented in the executing RTL. In CIL, a variable is either a static or dynamic object field, a top-level method formal, a local variable, or a stack location. Subsumed variables include object pointers and array handles that are initialized once and not subsequently changed, as well as loop variables where the loops are fully unwound and removed during elaboration. Other variables might also be removed in a later stage of processing if they have no effect or influence on the outputs of the RTL. Most object pointers and array handles are subsumed, but certain of them are represented in the final RTL. These are ones that range over a fixed, finite set of objects, created during elaboration, but which get pointed at from different places: e.g. as messages are passed between objects. Since these pointers range over a known, fixed population, they can be recoded into a packed form using an appropriate number of bits. Our current approach does not support fresh object allocation or garbage generation during run-time RTL execution, although, in the future, using more static analysis, we hope to be able to compile any program that uses a bounded number of objects.

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 two-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 stage 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 and heap at every basic block boundary. Such symbolic evaluation is straightforward since every operator and method call is strongly typed. In our implementation of this approach, 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 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, 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.

The second stage of processing for a thread performs loop unwinding and VM code generation. Certain loops must be unwound since they contain calls to new that are not matched with a dispose on each iteration and, as mentioned, this is not supported at runtime. Forking new threads often happens inside loops and these loops too must be unwound at compile time if the resulting design is to be finite, since each new thread turns into new hardware. Other loops may be selectively unwound to alter the trade off between silicon use and clock cycle use. The user can control clock cycle use and loop unwinding by inserting calls to the dummy functions Kiwi.Pause() and Kiwi.NoUnroll() in the bodies of the loop. In the future, we want to reduce dependence on such user-inserted directives and instead allow such trade offs to be controller by higher-layer metrics.

Future work will allow partial unwinding of loops (e.g. doing four or eight source-level iterations in parallel at run time) and hoisting of loop exit predicates. Loop order inversions can also be implemented to assist with 'blocking' leading to spatio/temporal reuse on large arrays held in cached DRAM.

As mentioned, C# does not have an RTL-like `generate` statement and so the same looping constructs are used both for structural generation and runtime process loops. The separation of function is achieved by compiling the program using as much compile-time constant propagation as possible and following conditional branches whose branch conditions are fully determined. Where the boundary conditions of a loop are determined only by constants, and the loop does not contain a call to Kiwi.Pause() or Kiwi.NoUnroll() then it will be unwound. The heap storage allocator is modelled in full with every call to newobj or newarr allocating a nominal, numeric address to each nominal object and all of its contents. These nominal addresses appear in the VM code generated for each pointer assignment or as arguments to calls to side-effecting hook in the run-time system.

Loops that consume run-time clock cycles, because they contain, directly or indirectly, a Kiwi.Pause() call, certainly cannot be unwound at compile time. In order to determine such a loop, each thread is compiled with the concept of a current *region* which is an integer. The operation of the elaborator when it encounters a call to Kiwi.Pause() is both to emit the call to an internal pause routine to the output VM code and also to note that the thread has entered a new region by incrementing the region counter. The operation of the elaborator when it encounters a back jump (i.e. to a point it has already covered), where the destination point had a different value of region, is to emit a VM jump statement rather than following the flow of control.

Where an assignment to a variable that is not written by another thread is of a constant value, for which purpose nominal addresses and constant operations on them are considered constant, the value is kept by the elaborator as part of its current environment for that thread, as a variable to value mapping, and substituted out in any subsequent references to that variable by the thread. Where a thread makes a back jump, where the environment on the previous pass contained a different assumption over the value of a variable, the whole elaboration of that thread is rolled back to an earlier point in its progress, where the work can be done again, augmented with a note not to store that variable in the environment at that point in the code.

In a third stage of processing, a points-to analysis finds disjoint regions in the nominal address space and then implements each region as a register, register file, RAM or DRAM segment according to its size. A cone-of-influence analysis deletes any code that has no effect on output values. Each VM is then mapped to hardware components using conventional C-to-gates synthesis techniques that balance loads on structural resources, such as memory ports.

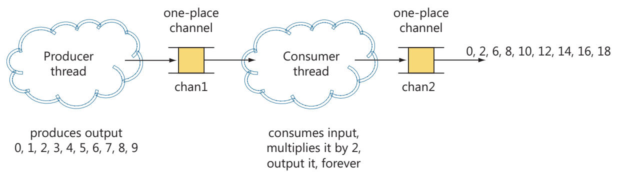


Figure 5. A producer/consumer scenario.

V. 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 5.

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.

```
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.

The pause statements in the producer, consumer or output loop can be omitted, thereby not dictating a particular mapping of the behavior to hardware clock cycles, although clock cycles will still be consumed by a thread if it blocks in one of its Read or Write calls to a Kiwi.Channel. The thread schedule that is chosen by the compiler can alter whether the data is passed using combinational logic between a pair of stages or whether it is registered and consumes a clock cycle at that point. For instance, if a data generator is statically scheduled before an always-ready sink, then no handshake wires are needed in the generated hardware at that point and the decision between combinational and pipelined data transfer can be made based on other grounds, such as conventional high-level synthesis metrics that globally balance pipeline stages or balance loading on structural resources, such as memory ports.

VI. 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];

    // Clear to window of x values to all zero.
    for (int w = 0; w < size; w++)
        window[w] = 0;

    // For each sample...
    for (int i = 0; i < input.Length; i++)
    {
        // Shift in the new x value
        for (int j = size − 1; j > 0; j−−)
            window[j] = window[j − 1];
        window[0] = input[i];

        // Compute the result value
        int sum = 0;
        for (int z = 0; z < size; z++)
            sum += weights[z] * window[z];
        result[i] = sum;
    }
    return result;
}
```

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 6.

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.

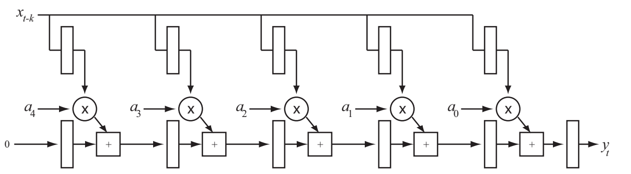


Figure 6. A transposed multi-tap filter.

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.

```
static void Tap(int a, Kiwi.Channel<int> xIn, Kiwi.Channel<int> yIn,
                Kiwi.Channel<int> yout)
{
  int x;
  int y;
  while(true)
  { y = yIn.Read();
    x = xIn.Read();
    yout.Write(x * a + y);
  }
}
```

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 6 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.

```
static void ParallelFIR(int size, Kiwi.Channel<int> xin,
                        Kiwi.Channel<int> yout)
{
    Kiwi.Channel<int>[] Xchannels = new Kiwi.Channel<int>[size];
    Kiwi.Channel<int>[] Ychannels
      = new Kiwi.Channel<int>[size + 1];

    // Create the channels to link together the taps
    for (int c = 0; c < size; c++)
    {
        Xchannels[c] = new Kiwi.Channel<int>();
        Ychannels[c] = new Kiwi.Channel<int>();
        Ychannels[c].Write(0); // Pre-populate y-registers with zeros
    }
    Ychannels[size] = new Kiwi.Channel<int>();

    // Connect up the taps for a transposed filter
    for (int i = 0; i < size; i++)
    {
        int j = i;
        Thread tapThread = new Thread(delegate()
                    { Tap(j, weights[j], Xchannels[j], Ychannels[j],
                          Ychannels[j+1]); });
        tapThread.Start();
    }

    // Broadcast the input
    Thread broadcast = new Thread(delegate()
                { BroadcastInput(xin, Xchannels); });
    broadcast.Start();

    // Insert an infinite sequence of 0s into the first Y channel stage
    Thread zeroYs = new Thread(delegate()
                { ZeroFirstY(Ychannels[0]); });
    zeroYs.Start();

    // Drive yout
    int yresult;
    while (true)
    {
        yresult = Ychannels[size].Read();
        yout.Write(yresult);
    }
}
```

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.

```
class ParallelConvolver
{
    const int size = 5;
    static int[] weights = new int[size] {2, 5, 6, 3, 1} ;

    [Kiwi.InputIntPort("sample")]
    public static int sample;

    [Kiwi.OutputIntPort("result")]
    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:

```
[Kiwi.OutputWordPort("result", 31, 0)]
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 filter sub-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;
    }
}
```

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 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 7.1 nanoseconds on an XC5VLX50T-1 part and can operate at 141MHz. The handshaking protocol that was generated used four cycles to process each 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 would make the FPGA synthesis tools map the remaining filter tap stage into a DSP48E block and this would have been present if our part of the flow had chosen a different static schedule for the user's threads.) 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.

VII. Conclusions

We have shown that system-level concurrency constructs can be synthesized into circuits. This capability can be exploited 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 constructs expressed in terms of these constructs e.g. join patterns, multi-way rendezvous and data-parallel programs.

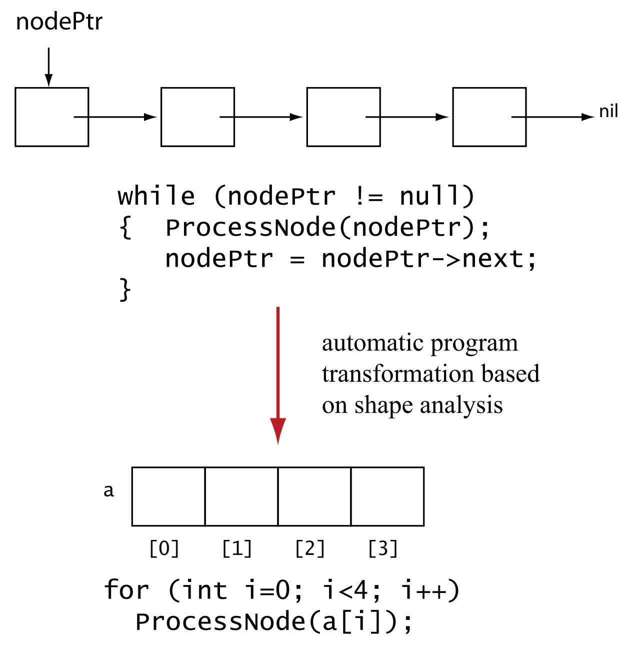
We see FPGAs as a viable target platform for the future of scientific programming and systems like Kiwi are needed to make FPGAs accessible to the wider community. Rapid development and debugging is facilitated by the use of standard languages and the ability to 'run' the design on a standard processor before being processed by FPGA place and route tools. (FPGA tools may themselves be faster in the future if FPGA architecture moves from bit-level to word-level internal implementation.)

The designs presented in this paper were developed using an off-the-shelf software integrated development environment (Visual Studio 2008) 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 existing research on the automatic synthesis of sequential programs and 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 to reconfigurable computing technology for mainstream programmers. The advent of many-core processors will require programmers to write parallel programs anyway, so it is appropriate to investigate 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.

In this paper, we separately compiled each user thread, generating hardware that can model all possible execution interleavings, but in future work we will analyse inter-thread trigger/guard dependencies and accordingly restrict the allowable interleavings so that fewer gates are used for handshaking.



```
while (nodePtr != null)
{  ProcessNode(nodePtr);
   nodePtr = nodePtr->next;
}
```

```
for (int i=0; i<4; i++)
  ProcessNode(a[i]);
```

Figure 7. Possible conversion of a linked list program to use arrays when maximum list length could be determined using static analysis.

Although we currently support limited forms of dynamic storage allocation, where the heap has the same structure on each run-time iteration of a loop, we aim to deploy recent results in shape analysis [15] and separation logic [16] to automatically transform more-general programs into their statically-allocated and array-based equivalents. We will build upon existing work by the authors in the context of C [17], [18] and extended it to a subset of CLR bytecode. For example, Figure 7 demonstrates how we might 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. 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] R. K. Gupta and S. Y. Liao, "Using a programming language for digital system design," *IEEE Design and Test of Computers*, vol. 14, Apr. 1997.

[2] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," *The 19th Annual International Symposium on Computer Architecture*, May 1992.

[3] A. Takach, B. Bower, and T. Bollaert, "C based hardware design for wireless applications," *Design, Automation and Test in Europe*, 2005.

[4] F. Bruschi and F. Ferrandi, "Synthesis of complex control structures from behavioral systemc models," *Design, Automation and Test in Europe*, 2003.

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

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

[7] 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*, vol. 36, no. 8, 2003.

[8] 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*, Mar. 2006.

[9] S. Gupta, N. D. Dutt, R. K. Gupta, and A. Nicolau, "SPARK: A high-level synthesis framework for applying parallelizing compiler transformations," *International Conference on VLSI Design*, Jan. 2003.

[10] 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.

[11] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah, "Liquid metal: Object-oriented programming across the hardware/software boundary," in *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 76–103.

[12] L. Séméria and G. De Micheli, "Spc: synthesis of pointers in c: application of pointer analysis to the behavioral synthesis from c," in *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*. New York, NY, USA: ACM, 1998, pp. 340–346.

[13] J. Babb, M. Rinard, C. A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe, "Parallelizing applications into silicon," *7th IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.

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

[15] S. Magill, M. Tsai, P. Lee, and Y. Tsay, "THOR: A tool for reasoning about shape and arithmetic," *Computer Adided Verification (CAV)*, 2008.

[16] M. Raza, C. Calcagno, and P. Gardner, "Automatic parallelization with separation logic," *European Symposium on Programming (ESOP)*, 2009.

[17] B. Cook, A. Gupta, S. Magill, A. Rybalchenko, J. Simsa, S. Singh, and V. Vafeiadis, "Finding heap-bounds for hardware synthesis," *Formal Methods for Computer Aided Design (FMCAD)*, 2009.

[18] J. Simsa and S. Singh, "Designing hardware with dynamic memory abstraction," *ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA)*, 2010.