Distributing C# Methods and Threads over Ethernet-connected FPGAs using Kiwi

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*—The Kiwi system achieves co-design by allowing nominated regions of C# programs to be targeted at FPGAs while the remainder executes on unmodified .NET and Mono virtual machines. Using C# attributes, certain methods are identified for separate compilation and collections of methods are mapped to multiple FPGAs or to workstations connected to a common Ethernet switch. Individual methods become RPC-callable entities for the top-level C# thread running on one workstation, while server threads may run continually on other workstations or FPGAs. We illustrate the concept using minimal modifications to an Adobe Photoshop plug-in where the processing for each colour channel is farmed over the Ethernet to one or three remote entities which may each be either an FPGA or a workstation.**

I. Introduction

Emerging infrastructure for cloud computing will need to exploit special purpose hardware accelerators including FPGA and GPU in addition to von Neumann resources [1]. These specialized processing elements are necessary for reducing latency and energy consumption to meet our requirements for data processing which cannot be serviced by multicore processors. The general potential is illustrated by success in some domain-specific examples, such as code cracking, DNA sequencing and automated trading (e.g. the MoldUDP protocol for automated trading implemented on FPGAs that issue buy and sell requests [2]) In this paper we explore splitting C# programs into separate components and hosting them on a cluster of workstations and FPGAs interconnected by an Ethernet LAN. Modern FPGAs have (multiple) on-board Ethernet MAC blocks, so connecting them to the LAN is not a problem. Compared with C-to-gates technology, which focuses on highly-efficient implementation of limited statically-allocated subsets of single-threaded C/C++ programs, we look at multi-threaded programs coded in C#. Starting with multi-threaded programs gives us a greater seam of potential parallelism to tap. This was the motivation of the original Kiwi project [3]. However, it is also a source of spatial parallelism, where different parts of an algorithm are placed on different execution platforms.

Apart from coding style and language subset issues, which have been well explored in the past, another main obstacle to using FPGAs for general computations is the long compile times incurred by the vendor synthesis, mapping, placement and routing tools. Although some work has been devoted to less fine-grained reconfigurable architectures [4], [5], which should speed up place and route times, other techniques deserve study. We take it for granted that software toolchains support modular reuse of separately-compiled libraries (DLLs) that may be locally linked or remotely invoked by RPC (e.g. using Apache Tomcat) and then run in parallel. Equivalently, modern FPGA tools now enable designs to be combined together after place and route for complete or partial reconfiguration of the FPGA array. For instance, the Xilinx 'bitgen' program enables initialization files for memories to be inserted into the bitstream immediately prior to FPGA download. The Xilinx PlanAhead tool allows us to design a sub-component and constrain it to a rectangular sub-region of the FPGA. This operation is much quicker than performing a full place and route of the whole FPGA (tens of seconds instead of tens of minutes). For even more rapid, early devlopment, the whole application can be developed and debugged as a Mono or .NET application on a single workstation without any hardware compilation.

Our ultimate objective is to demonstrate automated bitstream-level assembly of such components for FPGAs. As a stepping stone towards that goal in this paper we show how .NET C# programs split over several DLLs can be separately compiled using the KiwiC compiler. These assemblies are then ready for distribution over a mesh of workstations and FPGAs. Each hosted application sports a standard net-level interface that should be amenable to abutment-style wiring, as is required if no routing is to be done post placement. We give details of how data is transferred between components within an FPGA and over the LAN. Specifically, in this paper, we present a method to partition an application over some number of Ethernet-connected nodes and also how to host multiple application components on a single FPGA.

Figure 1 illustrates our general setup, where C# programs are running on a mixture of FPGAs, Linux and Windows workstations, interconnected by a LAN. One of the workstations is a client for the others which resemble the server cloud.

Automatic static allocation of tasks to processing elements

978-1-4577-0118-4/11/$26.00 ©2011 IEEE

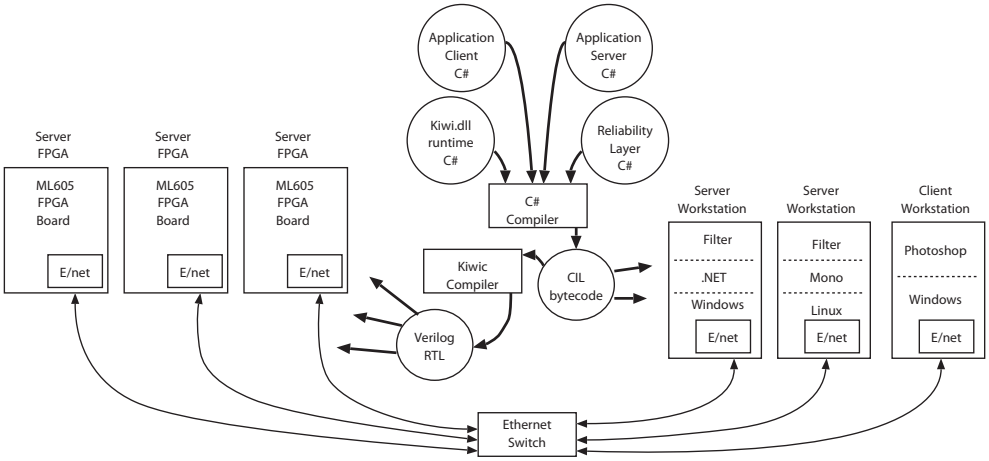

Figure 1. Experiment 1: Running a Photoshop plug-in on a mixture of FPGA and workstation servers.

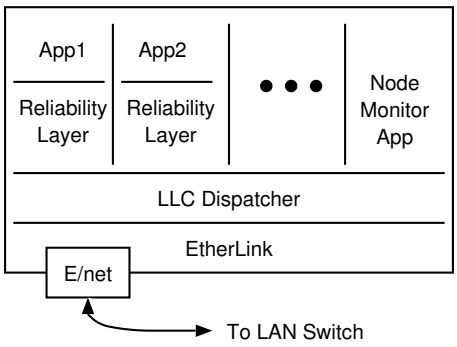

Figure 2. FPGA architecture for Experiment 2, offering multiple, separately-compiled services.

in the absence of runtime profiles is unlikely to achieve good results. We have not yet investigated parsing the output of profiling tools as part of our flow, so we use manual partitioning at the moment. The partitioning is manifested as a 'toplevel.v' structural RTL file that instantiates the separately-compiled hardware components as well as a Makefile that invokes the C# and KiwiC compilers and the Xilinx synthesis, map, placement and bitstream tools. Selection of DLLs for running on workstations is performed by conditionally copying them into folders or adjusting the MONO_PATH and/or PATH environment variables. An ultimate aim is that our 'toplevel.v' file is invariant over a large number partitioning decisions, or at least one of several pre-compiled top levels can be automatically selected.

An important aspect of our approach is that .NET DLLs do not need to be recompiled between running on the FPGA and workstations. This enables us to claim that we are providing cloud-like execution resources where the author of the code was not aware of his execution platform. This claim is slightly tenuous since some current examples contain specific Kiwi attributes or calls (or even updates to LED indicators) that a general author would not include.

Another significant aspect is that the complete application can be run as .NET binary code on a single workstation for development and debugging, before farming it out to the cloud.

II. Kiwi Introduction

The basic principles behind the design of the Kiwi system are:

- The use of an existing language rather than the invention of a new language: this is achieved by taking as input .NET bytecode rather than the source text of a specific programming language;
- The modeling of hardware architecture by using concurrency to describe the behaviour of system components: this is achieved by using the .NET threading library called System.Threading and in particular thread synchronize and communicate with thread-safe channels (which are implemented as FIFOs).

By not extending an existing language the programmer is free to use a regular compiler to compile programs built against a Kiwi library and then execute them on a computer to achieve a simulation of the system description. Furthermore, standard static analysis tools (e.g. deadlock checkers) can be applied to the regular .NET bytecode.

An artificial but complete example of a circuit that iteratively computes the factorial function in Kiwi and C# is shown below. This program contains a class called Factorial which has a custom attribute [Kiwi.Hardware()] which indicates that this class contains a method that should be converted into hardware through Kiwi synthesis to yield a Verilog file.

Somehow the inputs and outputs of a generated circuit module need to be identified. In Kiwi this is done by decorating static fields with one of several special custom attributes that specify ports, the polarity of a port, a name

for it if something other than the .NET name is required and for integer types the number of bits required to represent an integer value can be specified. In this case there is an unsigned 8-bit integer input n and two outputs: a 126-bit unsigned integer fac which will eventually represented the factorial of the input n and a single bit output done which will go high when the factorial has been computed.

```
using System;

class Factorial
{
  [Kiwi.Hardware()]

  [Kiwi.InputWordPort(7, 0)]
  static uint n ; // Input to factorial circuit

  [Kiwi.OutputWordPort(15, 0)]
  static uint fac = 1; // Result of factorial circuit

  [Kiwi.OutputBitPort]
  static bool done = false; // Signal indicating when result is ready

  static void FactorialCircuit ()
  {
      uint i = n; // On reset capture the input n
      while (!done)
      {
          if (i > 1)

              fac = fac*i;
              i--;
              if (i == 1)
              {
                  Console.WriteLine("Factorial is {0}", fac);
                  done = true;
              }
          Kiwi.Pause();
      }
  }

  public static int Main() // Test-bench
  {
    n = 5;
    FactorialCircuit();
    return 0;
  }

  public static void HWEntryPoint() // Top level of hardware circuit
  {
      FactorialCircuit();
  }

}
```

The method FactorialCircuit computes the factorial circuit and works iteratively performing a multiplication in each clock cycle until the base case is reached. The explicit use of Kiwi.Pause() indicates a synchronization with an implicit clock.

The main program acts like a test bench for this program. This program can be compiled and executed and will produce the output Factorial is 120. The method HWEntryPoint is a special method that is understood to be the hardware entry pint and in this case it just calls the static method for computing factorial. When this program is submitted to the Kiwi system it synthesizes the FactorialCircuit method into a Verilog module which can then be implemented for our ML605 FPGA boards using the Xilinx ISE tools (or other vendor tools).

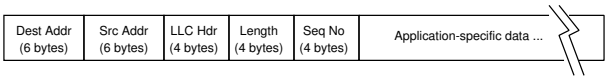

Figure 3. Ethernet MAC frame format as typically used in our experiments.

III. Exp. 1: Single Application per FPGA

In order to change the protocol stack implementation between workstation and FPGA execution platforms, we place the different layers in different DLLs and have multiple alternatives at various layer. A client in one layer makes method calls on the service access points provided by the lower layer. We then choose a particular combination when targeting a particular platform. Linking of separately-compiled sections on a workstation is performed by the .NET/Mono class path loader. Linking of separately-compiled sections on the FPGA is performed by joining hardware nets with wires. An alternative approach would have been to exploit an existing inter-process communication framework like Windows Communication Foundation (WCF) however we choose our lighter-weight variant to avoid taking a dependency on too much .NET infrastructure and to make our system work just as effectively on Linux as on Windows.

We first describe an experiment where a single application server (an Adobe Photoshop plug-in) was developed in C# and run both on the same workstation under .NET as the Photoshop application and on locally-attached FPGA cards. In section IV we describe a second experiment where multiple application servers can be placed on each FPGA at once.

Figure 3 shows the typical experimental frame format we used in these experiments. The format is defined entirely by the C# code and is conveyed through the workstation operating system using the raw-frames API or compiled into gates by KiwiC for the FPGA platforms. Rather than using UDP or TCP we used a strictly word-oriented format that was easier to develop. In terms of address resolution, all of the current implementations send replies to the last MAC address that sent a message, which works fine when each server is peered with just one client workstation.

In Experiment 1, three DLL files were compiled by KiwiC to Verilog RTL and combined with 'toplevel.v' to complete the FPGA circuit. They were KiwiNetworkDevice.dll, Reliability.dll and PhotoFilter.dll (containing PhotoFilterChannel and ThreeChannels). Hence one application server was hosted on one FPGA. Two KiwiC compilations were in fact used, since the application and its reliability/presentation layer were compiled together rather than as separate RTL sections. This was done for convenience in this instance,

but generally compiling DLL's together results in inter-procedural optimisations arising from a combined datapath and sequencer. These optimisations are not generally discovered by the state machine optimisers in the FPGA tools when operating on individual controllers in isolation. The reliability layer places a sequence number and length field in each frame of the packet stream for each application. The current implementation only checks and logs errors in a global variable, rather than requesting a retransmit.

The Virtex 6 series of FPGAs contain a Tri-Mode Ethernet MAC block which is easy to connect to at the net-level using the LocalLink protocol.

Figure 4 illustrates the transmit-side, low-level LocalLink code that deals with the net-level interface to the Ethernet MAC. This DLL is only used on the FPGA platform and, for brevity, we do not list the workstation alternative. Since the net-level protocol is defined, as is common, in terms of clock-cycles, this is compiled in Kiwi 'hard pause mode' where clock cycles must not be introduced by KiwiC except at the manually-inserted `Pause()` calls. This allows us to specify cycle accurate behaviour. It can be seen that the transmitter uses source and destination addresses formed by swapping over those from the last-received frame.

Customers send payload data to the KiwiNetworkDevice static class instance by calling its static method `WriteInt` defined in Figure 5. This is marked with a Kiwi attribute `Remote` so that a net-level interface to it is generated when the C# compiled DLL is further compiled by KiwiC, to give output as partially shown in Figure 6. The 'Remote' mechanism enables this procedure to be called by a separately-compiled section of hardware. Buses of the appropriate width are constructed for each argument (and would be for the return value if it were not `void` in this example) along with `req` and `ack` signals that execute a four-phase handshake. In general, any number of methods can be attributed in this way, each resulting in an additional set of connections to the RTL component. For the KiwiNetworkDevice module, four methods were marked in total, the other three being

```
public static void DiscardRxFrame();
public static uint RxBytes();
public static uint ReadInt();
```

For all application modules, we shall use a standard net-level interface to ultimately support abutment-based wiring, but the system classes do not require this as they are specifically instantiated in 'toplevel.v'.

The client thread for these procedures can be compiled in the normal 'soft pause mode' but some manual `Pause` calls are still required (as shown in figure 5). These were inserted because part of the KiwiC compiler that overcomes structural hazards in the generated code is currently broken and the resulting circuit would otherwise attempt four writes on the byte-wide memory in one clock cycle. An alternative solution would have been to write the C# code with a 32-

```
// A static class since only one LAN port in use.
public static class KiwiNetworkDevice
{
 static byte[] rx_buffer = new byte[2048];
 static byte[] tx_buffer = new byte[2048];

 static int tx_PktLength, tx_PktPtr;

 //Use Kiwi attributes to define the net-level connections
 [Kiwi.OutputWordPort("tx_data")]
 static byte tx_data; // Write data to be sent to device
 [Kiwi.OutputBitPort("tx_sof_n")]
 static bool tx_sof_n = !false; // Start of frame indicator
 [Kiwi.OutputBitPort("tx_eof_n")]
 static bool tx_eof_n = !false; // End of frame indicator
 [Kiwi.OutputBitPort("tx_src_rdy_n")]
 static bool tx_src_rdy_n = !false; // Source ready indicator
 [Kiwi.InputBitPort("tx_dst_rdy_n")]
 static bool tx_dst_rdy_n; // Destination ready indicator

 static void SendPacket()
 { Kiwi.PauseControl oldmode = Kiwi.PauseMdSet(Kiwi.hardPause);
   tx_src_rdy_n = !true; // We are not at the start of a frame
   // Now send an Ethernet packet back to where it came from
   // Swap source and destination MAC addresses
   int j = 0;
   tx_sof_n = !false;
   for (j = 6; j < 12; j++) // Emit src address from companion
   {

       tx_data = rx_buffer[j];
       tx_sof_n = j != 6;
       Kiwi.Pause();
   }
   for (j = 0; j < 6; j++) // Now emit src address as dest
   {
       tx_data = rx_buffer[j];
       Kiwi.Pause();
   }

   // Transmit the remanining bytes from transmit buffer
   j = 12;
   while (j < tx_PktLength)
   {
       tx_data = tx_buffer[j];
       if (j == tx_PktLength - 1)
           tx_eof_n = !true;

       j++;
       Kiwi.Pause();
   }
   tx_src_rdy_n = !false;
   tx_eof_n = !false;
   Kiwi.Pause();
   // End of frame, ready for next frame
   Kiwi.PauseMdSet(oldmode);
}
```

Figure 4. Ethernet MAC interface module, transmit LocalLink code, compiled in 'hard pause' mode where clock cycles may only be inserted in correspondance with `Pause` calls in the source code.

```
[Kiwi.Remote("EtherentLocalLink", "parallel:four−phase")]
public static void WriteInt(uint d, KiwiFarmingInterface.Framing kfp)
{
    if (kfp == KiwiFarmingInterface.Framing.Start) tx_PktPtr = 0;
    // Three pauses calls currently: avoids struct haz on mem write
    tx_buffer[tx_PktPtr ++] = (byte)(d >> 24); Kiwi.Pause();
    tx_buffer[tx_PktPtr ++] = (byte)(d >> 16); Kiwi.Pause();
    tx_buffer[tx_PktPtr ++] = (byte)(d >> 8); Kiwi.Pause();
    tx_buffer[tx_PktPtr ++] = (byte)(d >> 0);
    if (kfp == KiwiFarmingInterface.Framing.End)
    {
      tx_PktLength = tx_PktPtr;
      SendPacket();
    }
}
```

Figure 5. Ethernet MAC interface module, transmit-side client entry point.

```
module EtherLink(reset, clk, tx_dst_rdy_n, ...
  input reset;
  input clk;
  input tx_dst_rdy_n;
  output tx_src_rdy_n;
  output tx_eof_n;
  output tx_sof_n;
  output [7:0] tx_data;
  output KiwiNetworkDevice_WriteInt_ack;
  input KiwiNetworkDevice_WriteInt_req;
  input [31:0] KiwiNetworkDevice_WriteInt_d;
  input [31:0] KiwiNetworkDevice_WriteInt_kfp;
  output [31:0] KiwiNetworkDevice_ReadInt_return;
  ...
```

Figure 6. Partial signature of EtherLink (Kiwi-MAC interface module) of compilation unit (Verilog RTL listing).

bit wide memory and to serialise and deserialised on the LocalLink side of the buffers.

For brevity, the mutex to stop concurrent operation of the `WriteInt` and `SendPacket` methods on the same buffer is deleted from the listing, but it follows the same pattern as we shall illustrate for the Dispatcher component.

Figure 7 shows the example application for Experiment 1—a one dimensional convolver. This follows the design pattern common to all our applications, of using its own thread (started by a separate code, not shown, when running on the local workstation and started by the KiwiC compiler in response to the `Hardware()` attribute when running on the FPGA). It requests work by making a blocking read into `ArrayRead`. Note the call to `Kiwi.Pause()` in the inner loop, which is a suggestion (in 'soft pause mode') to KiwiC to consume one clock cycle per iteration. Also, note that we have simplified the code in the listing to do each channel in turn, whereas separate threads for each channel within the application are used in the fuller version. Another possibility, that is easy to code, is to just handle one channel on each FPGA and to use three FPGA cards in parallel to serve the workstation. A third possibility is that three instances of this complete, single-threaded application are run in parallel on one FPGA using the mechanisms of

```
class PhotoFilterChannel
{ int[] coefs = new int [9] {1, −2, 3, −4, 5, −4, 3, −2, 1 };
  int[] data = new int [9];
  int ptr, max;
  public int convolve(int din)
  {
    ptr = (ptr == coefs.Length−1) ? 0: ptr+1;
    if (ptr > max) max = ptr;
    data[ptr] = din;
    int sum = 0;
    for (int xx =0; xx<coefs.Length; xx++)
    {
      int yy = (ptr−xx + coefs.Length) % coefs.Length;
      if (xx <= max && yy<=max) sum += data[xx] * coefs[yy];
      Kiwi.Pause();
    }
    return sum;
  }
  public void Reset()
  {
    ptr = 0; max = 0;
  }
}

public static class ThreeChannels // Top−level for the application.
{ [Kiwi.OutputBitPort()]
  // We connect an oscilloscope to these for observation
  static bool rx_led, tx_led, work_led, poll_led;

  static PhotoFilterChannel yy_ch = new PhotoFilterChannel();
  static PhotoFilterChannel uu_ch = new PhotoFilterChannel();
  static PhotoFilterChannel vv_ch = new PhotoFilterChannel();

  static ReliableLayer FarmPort = new ReliableLayer();
  static int[] workbuf = new int[512];

  [Kiwi.Hardware()]
  public static void Main()
  { int k = 0;
    yy_ch.Reset();
    work_led = false; poll_led = true; tx_led = false;
    while(true)
     { rx_led = true;
       uint len = FarmPort.ArrayRead(workbuf);
       rx_led = false;
       yy_ch.Reset(); vv_ch.Reset(); uu_ch.Reset();

       work_led = true;
       for (int i=0; i<len; i+=3) // Work loop
       { workbuf[i+0] = yy_ch.convolve((int)workbuf[i+0]);
         workbuf[i+1] = uu_ch.convolve((int)workbuf[i+1]);
         workbuf[i+2] = vv_ch.convolve((int)workbuf[i+2]);
         poll_led = !poll_led;
       }
       work_led = false; tx_led = true; // Send result data out.
       FarmPort.ArrayWrite(workbuf, len);
       tx_led = false;
     }
  }
}
```

Figure 7. Photoshop Plugin Application (version with three filter channels as one application, simplified single-threaded version).

```
public void ArrayWrite(uint [] buffer, uint len)
{ // Add protocol id + flags
  KiwiNetworkDevice.WriteInt(0x45C03200, Framing.Start);
  KiwiNetworkDevice.WriteInt(tx_seqno, Framing.Mid);
  KiwiNetworkDevice.WriteInt(len, Framing.Mid);
  for (uint pp = 0; pp<len; pp++)
      KiwiNetworkDevice.WriteInt(buffer[pp], Framing.Mid);
  KiwiNetworkDevice.WriteInt(0x45C03201, Framing.End);
  // Protocol id + flags with end of message flag.
  tx_seqno = tx_seqno + 1;
}
```

Figure 8. The `ArrayWrite` method from the reliability/presentation layer code when directly operating on the KiwiNetworkDevice service-access point as in Experiment 1. (In Experiment 2 it is modified to invoke the dispatcher SAP).

Experiment 2 to dispatch work in parallel to each instance.

The methods `ArrayRead` and `ArrayWrite` are provided by a reliability and presentation layer, whose simplified code is shown in Figure 8.

Compared to following a heavy-weight approach like adopting custom attributes for implementing WCF for cross-process communication or the use of MPI or direct use of sockets we believe the level of abstraction provided by the highly specialized hardware and software custom attributes strikes a good balance between low-level detail and high-level intent.

IV. Exp. 2: Dynamic Dispatch

There are several potential levels where dynamic dispatch can be applied:

1) dynamic sharing on a per-FPGA basis (with one customer for one pre-loaded FPGA at any one time),
2) implementing more than one function on the FPGA, if the functions are small, and expecting to get better load-balancing from the improved sharing potential,
3) dynamic loading of FPGA with combinations of applications that need to be run.

Numbers 1 and 2 are addressed in this work whereas 3 is for further study. No 1 is provided by the packet routing within the Ethernet LAN.

For Experiment 2 we specifically chose option no 2 from the above list. We placed four application servers and a Dispatcher and a KiwiNetworkDevice on one FPGA. Each application had its own instance of the reliability layer. The applications were three instances of the one-channel PhotoFilter and one MonitorApp. The latter is a fairly simple application server that provides status and error reports by returning the value of public static variables in other classes (e.g. total number of packets processed and error counts). Each application connects to a port on the dispatcher and the first word of the Ethernet payload is used for selecting the application number. This is a crude approximation of the port-demultiplexing implemented by TCP and UDP. In

```
class Dispatcher
{ static Mutex tx_mutex;

 // Transmit interface − this is a simple exclusion zone:
 // only one application can send at a time
 public static void ClientWriteInt(uint data,
    KiwiFarmingInterface.Framing kf, uint port)
 {
   if (kf == KiwiFarmingInterface.Framing.Start)
   { // If start of write − need to write LLC header
     Monitor.Wait(tx_mutex); // Wait here until we gain the lock.
     KiwiNetworkDevice.WriteInt(llc_header_const<<16 | port,
                                KiwiFarmingInterface.Framing.Start);
     KiwiNetworkDevice.WriteInt(data,
KiwiFarmingInterface.Framing.Mid);
   }
   else if (kf==KiwiFarmingInterface.Framing.End)
   { // If end of write − need to release lock
     KiwiNetworkDevice.WriteInt(data, kf);
     Monitor.PulseAll(tx_mutex);
   }
   else KiwiNetworkDevice.WriteInt(data, kf);
 }
```

Figure 9. LLC Dispatcher module, transmit-side multiplexor with exclusion Mutex.

future we may use 802 LLC and other standard protocols at this layer.

Concurrent tasks can easily be run on a given FPGA, provided it has sufficient area, without crosstalk except at the network interface and DRAM ports. The examples in this paper did not use off-chip DRAM. Since each application has its own hardware thread or threads, and makes a blocking call into the dispatcher to receive its next work item, the concurrency relies on thread-safe re-entrant hardware being generated. Each application server may have any number of internal threads in general, but in this experiment we replaced the triple-threaded, three-channel PhotoFilter with three single-threaded, single-channel instances.

The LLC dispatcher module implements a logical-link layer packet demultiplexing using a table of registered hardware entities. Figure 9 shows the `ClientWriteInt` method which is re-entrant, being potentially called simultaneously by several loaded applications. Each client must gain exclusive access to the transmit method of the NetworkDevice module so a mutex is used directly. We assume clients obey a simple protocol based on the `Framing` enumeration type, where their first call sets this to `Start`, requiring the mutex to be claimed and also causing the LLC header to be inserted, and where their last call sets this to `End` causing release of the mutex and also signaling to the NetworkDevice that the packet can be sent.

The received packet handling is slightly more complex: each client application initially performs a 'listen' by sending a thread into the `ClientReadInt(port)` blocking method and supplying their port number as an argument. Each client has its own condition variable, `rx_ready`, in the `Farmable` record, that it blocks on. The Dispatcher

```
...
static int PortsInUse = 0;

public class Farmable
{
  public volatile bool rx_ready;
  public void IndicateRX() { rx_ready = true; }
  ...
}
static Farmable [] PortBindings = new Farmable [Ports];

public static int Register(string id)
{ // NB: entirely executed at compile time under KiwiC.
  PortBindings[PortsInUse] = new Farmable(id);
  return PortsInUse++;
}

static void ReceiverThread()
{
    // LLC-like header scan
    rxpktLen = KiwiNetworkDevice.ReadInt();
    WriteLine("Dispatcher - pkt len word {0}", rxpktLen);
    uint llc_header;
    do llc_header = KiwiNetworkDevice.ReadInt();
    // scan for LLC protocol id and flags.
    while (llc_header>>16 != llc_header_const)
    uint port = (uint)(llc_header & 0xFF);
    if (port >= PortsInUse)
    { WriteLine("Discarded frame ({1} words)"
        " rx'd on port {0}", port, rxpktLen, PortsInUse);
      KiwiNetworkDevice.DiscardRxFrame();
    }
    else
    { WriteLine("Forwarding frame {1}"
        " rx'd on port {0}.", port, rxpktLen);
      lock (rx_mutex)
      { PortBindings[port].rx_ready = true;
        // Set a ready flag and wait for client application
        while (PortBindings[port].rx_ready)
            { Kiwi.NoUnroll(); Monitor.Wait(rx_mutex); }
        Monitor.PulseAll(rx_mutex);
      }
    }
 }

// Receive client interface service access point
public static uint ClientReadInt(int port)
{ // This entry point blocks its thread until the dispatcher
   // thread receives a frame for this client.
   WriteLine("Listen from client blocked waiting.");
   lock (rx_mutex) // block client thread spinning here
   { while (!PortBindings[port].rx_ready)
          { Kiwi.NoUnroll(); Monitor.Wait(rx_mutex); }
   }
   uint rv = KiwiNetworkDevice.ReadInt();
   if (--rxpktLen == 0)
      lock (rx_mutex)
      {
         PortBindings[port].rx_ready = false;
         Monitor.PulseAll(rx_mutex);
      }
   return rv;
}
...
```

Figure 10. LLC Dispatcher module, listing continued, receive-side demultiplexor with condition variables and Mutex. Note that calls to 'new' are fully elaborated at compile time: a KiwiC rule is that the heap must have the same structure at each iteration of a non-unwound loop. (`WriteLine` calls are converted to Verilog `$display` statements but discarded at FPGA synthesis.)

receiver has its own thread, started by the initialisation code at the bottom of the `Dispatcher` class (not shown) that itself enters the NetworkDevice blocking receive method until the first packet is received. When this returns, it sets the condition variable of the desired recipient and spins until it is cleared again. The recipient, meanwhile, will copy the LLC payload from the NetworkDevice before clearing the flag.

The KiwiC compiler cannot handle separate compilation when threads in different compilations make operations on a common mutex. This would require synthesis of a hardware arbiter with an undefined number of input request signals. On the other hand, as mentioned in the introduction, we eventually aim to overcome FPGA place and route delays, and we certainly want to dynamically alter the mix of applications instantiated on a given FPGA. Therefore we cannot, in general, compile all of the applications together with the Dispatcher (although this is supported by KiwiC and is as illustrated by the reported metrics in §V). Our approach is to place the central arbiter mechanisms in a .DLL as is usual to make the program run on mono/.net, but also to write a number, n, of stub clients of these mechanisms as separate DLL's only used for hardware compilation. These stubs are combined with the main arbiter DLL in a single KiwiC compilation that results in an n-way arbiter being generated with separate net-level ports being exposed, one for each customer application. The same stub is used when compiling its related application under KiwiC to generate the calling-side interface nets. When run as a software program, the stubs have no special significance: instead they provide a slight inefficiency, introducing an extra layer of procedure calling between the client and the dispatcher. This inefficiency could be removed by a C# compiler that in-lines leaf calls to static methods compiled at the same time, or alternatively, sometimes it could be useful to put some minimal functionality in these stubs, such as presentation-layer format conversion. An example stub is shown in Figure 11 and the overall compilation flow is shown in Figure 12. The semi-manually created `top-level.v` and other KiwiC-generated RTL files, such as `EtherLink.v`, must also be included. Currently we used a single run of the Xilinx tools (and/or Synplify) to generate each bitstream but in future work we will combine separate placements.

V. Results

Table I indicates the size of the RTL files generated by the KiwiC compiler for each compiled section in terms of the number of lines of Verilog, flip-flops and memory location (RAM array) bits. Because the structural-hazard code in the KiwiC was not working, all of the RAM is distributed RAM instead of BlockRAM and the clock frequency was lower than expected.

```
public class App14Stub
{
   const int MyPortNo = 14;
   [Kiwi.Remote("StubPorts", "parallel:four-phase")]
   public static void WriteInt(uint d, KiwiFarmingInterface.Framing kfp)
   {
      Dispatcher.ClientWriteInt(d, kfp, MyPortNo);
   }

   [Kiwi.Remote("StubPorts", "parallel:four-phase")]
   public static uint ReadInt()
   {
     return Dispatcher.ClientReadInt(MyPortNo);
   }
}
```

Figure 11. An example application stub.

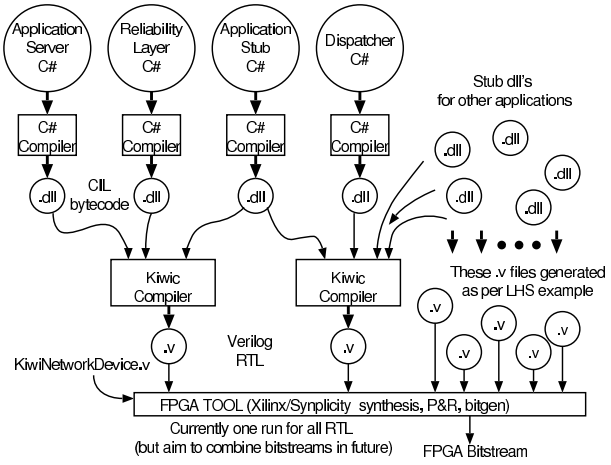

Figure 12. Exp. 2 Tool Flow - Each application is compiled separately with its presentation/reliability layer and stub and then each stub is also compiled with the central dispatcher.

Module	Used in Exp	RTL lines	Scalar bits	Total bits
EtherLink.v	1+2	502	320	33121
3ch PhotoFilter.v	1	1127	128	16800
toplevel1.v	1	88	0	0
Dispatcher.v	2	2059	256	2720
MonitorApp.v	2	452	32	64
1ch PhotoFilter.v	2	1127	128	5712
toplevel2.v	2	208	0	0

Table I
CODE SIZE OF EACH SEPARATE-COMPILATION ENTITY (EXCLUDING ETHERNET MAC). THE RELIABILITY LAYER DLL WAS COMBINED UNDER KIWIC WITH EACH APPLICATION SO DOES NOT APPEAR SEPARATELY.

Test	Place+Route CPU time	LUTS	DSP blks	Utilisation percent	Clock freq
Ex. 1	21 mins	45025	3	7.4	31 Mhz
Ex. 2	50 mins	89046	5	14	20 MHz

Table II
FPGA XC6VLX240 P&R LUT UTILISATION (EXCLUDING ETHERNET MAC), PLACE+ROUTE TIME (SYNPLIFY PREMIER, 5332 BOGOMIPS X86_64) AND MAXIMUM CLOCK FREQUENCY.

The ML605 card uses an XC6VLX240 device. Table II reports the utilisation and clock frequencies together with the place and route time. The execution time for the KiwiC and C# compilers was at least two orders of magnitude lower than the FPGA tool time.

VI. RELATED WORK

Automatic synthesis of channel handshaking signals between compilation units is not new in hardware synthesis. Like KiwiC, compilers for the Handle-C [6] and Bluespec [7] languages generate a handshake signal in each direction between components with these nets commonly disappearing when components are part of the same compilation run or when attributed as 'always-ready' or similar. Cardoso [5] describes a C compiler for a reconfigurable execution platform that has course grain for rapid place and route, similar to [4]. Swapping modules at run-time on a Virtex-4 FPGA was presented in [8] and FPGA companies are increasing support for dynamic reconfiguration, but there are many restrictions remaining, such as a tile bitstream being tied to a fixed absolute X-Y co-ordinate within the device with no API for moving it. An alternative approach would be to follow the example of Wires on Demand [9] which performs light-weight run-time placement and routing. We avoid this approach because run-time placement and routing is still a very fragile technology and we target scenarios where off-line construction of a programming bit-stream is acceptable.

The closest related work to the Kiwi project is the Liquid Metal project at IBM and the associated Lime language [10]. The Liquid Metal project takes the decision to design a new hardware description language by adding concurrency and hardware constructs to a subset of Java. This permits much more direct descriptions of system behaviour although it also requires the development of a special compiler and standard Java program analysis tools will no longer function on the extended subset. In comparison, we suffer the constraints of adding extra hardware and parallel behaviour information through custom attributes but in return we do not need to develop special compilers and we benefit from existing static analysis tools e.g. for deadlock detection. Furthermore, since our approach takes as input .NET bytecode we are not tied to a specific language and the Kiwi system can process descriptions written in other languages like F#, VisualBASIC and Ada or any other language with a compiler that targets .NET bytecode.

VII. CONCLUSIONS

We have developed a preliminary experimental infrastructure where

1) DLLs generated from C# and other .NET languages can be developed and debugged using standard software tools on an unmodified workstation,

2) users are encouraged to use multi-threaded C# to express parallelism that can be exploited during execution in a distributed system,
3) a run-time infrastructure (also in C#) that provides a 'Farmable' interface, where application servers written to that interface can be placed on the local workstation, remote workstations or in FPGA,
4) a hardware infrastructure where separately-compiled DLLs can be allocated to FPGA platforms using Makefiles and the like,
5) a vision for assembling the pre-compiled application servers on to the FPGA after the time-consuming place and route step.

We are advocating the use of a hardware, net-level API for binding components that has a software dual in the .NET system. The interface between the application and its server is defined by our Farmable abstract interface which translates into a concrete electronic API consisting of those nets. The marshalling code that packs and unpacks the frame is part of the application code and so the interface remains the same even if the arguments to the user's distributed functions vary.

At a future stage, we would like to move to some automated farming scenario, perhaps based on load balancing. We can envision dynamically loading FPGAs with pre-compiled bit streams and dynamically generating the bit streams in the way we currently concatenate the RTL files from separate KiwiC compilations.

Our methodology also supports access to the FPGA DRAM, but again with static allocation of compiled DLLs to regions of address space. Automating the memory map allocation (as in LEAP [11]) and providing run-time allocation/sharing is future work.

It may be argued that automatically generating n-way arbiters on demand inside the KiwiC compiler would be better than writing a specific Dispatcher C# class and compiling it with the required number of customers that register. The contrary argument is that the Dispatcher class needs writing only once, but being in C# it is easy to modfiy if required and compiling it with the required number of customers adds no complexity beyond what is already needed to automate the top-level assembly and wiring configuration for the FPGA (as done with Makefiles, Perl/Python etc.).

REFERENCES

[1] A. Madhavapeddy and S. Singh, "Reconfigurable data processing for clouds," in *Field-Programmable Custom Computing Machines (FCCM), 2011 19th IEEE Annual International Symposium on FPGAs for Custom Computing Machines*, April 2011.

[2] N. T. Note, "MoldUDP-64 specification," *Docstore/Nasdaq*, 2006.

[3] D. Greaves and S. Singh, "Designing-application specific circuits with concurrent c# programs," in *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, 2010, pp. 21 –30.

[4] D. Grant, G. Smecher, G. Lemieux, and R. Francis, "Rapid synthesis and simulation of computational circuits in an mppa," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, 2009, pp. 151 –158.

[5] J. Cardoso and M. Weinhardt, "From C programs to the configure-execute model," in *Design, Automation and Test in Europe Conference and Exhibition, 2003*, 2003, pp. 576 – 581.

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

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

[8] S. Koh and O. Diessel, "Communications infrastructure generation for modular fpga reconfiguration," in *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, 2006, pp. 321 –324.

[9] P. M. Athanas, J. Bowen, T. Dunham, C. Patterson, J. Rice, M. Shelburne, J. Surís, M. B. Bucciero, and J. Graf, "Wires on demand: Run-time communication synthesis for reconfigurable computing," in *FPL*, 2007, pp. 513–516.

[10] 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.

[11] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer, "LEAP scratchpads: automatic memory and cache management for reconfigurable logic," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA'11. New York, NY, USA: ACM, 2011, pp. 25–28. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950421