Developing High-Performance Streaming Applications in C#

KEVIN MITCHELL, Agilent Technologies

Many algorithms in areas such as digital signal processing can be elegantly expressed as fine-grained streaming computations. Unfortunately mapping such representations to a mainstream programming language in an efficient fashion is often challenging. We describe a streaming framework, S#, that is built on top of C#. We show how streaming algorithms can be expressed concisely in this framework, and how a post-build verifier can be used to enforce program safety. Once a program has been developed and debugged in C# it can be compiled to C++ code for improved efficiency. We can also automatically convert it into a form suitable for execution on an FPGA. We describe this process, and how the components that result can be used as subgraphs within a larger hybrid computation that is spread between the PC and the FPGA.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors

General Terms: Design, Algorithms, Languages, Performance

Additional Key Words and Phrases: Stream processing, C#, decompilation, Vivado HLS

1. INTRODUCTION

Some algorithms are most naturally expressed as stream-processing operations, with data flowing through a graph of processing elements. For instance digital signal processing, networking and encryption algorithms are often of this form. We can distinguish between fine and coarse-grained stream graphs. Fine-grained graphs, where the data flowing through the graph consists of individual values, such as integers or complex numbers, are often the most natural, particularly in many DSP algorithms. But they can be challenging to implement efficiently, particularly if we need to target a range of different architectures, such as multicore PCs, GPUs and FPGAs.

Numerous languages exist for expressing such algorithms in a natural fashion, for example StreamIt [Thies et al. 2002], Brook [Buck et al. 2004], StreamC/KernelC [Kapasi et al. 2003] and Cg [Mark et al. 2003]. However, in many real applications the stream processing component forms only part of a larger application, often just a very small part. Most dataflow languages are too specialized to implement the whole application. We therefore often require a hybrid solution where different parts of the application use different languages. The S# project, the subject of this paper, takes a different tack. Our goal is to allow fine-grained streaming algorithms to be described in a natural fashion within the context of an existing mainstream programming language, C# [Microsoft 2006]. Furthermore, by using the techniques described in this paper, we can generate code that is competitive with other solutions on a range of target architec-

---

Author's address: K. Mitchell, Measurement Research Laboratory, Agilent Technologies UK Limited, Edinburgh EH12 9DJ, UK. Email: kevin_mitchell@acm.org

tures, including FPGAs. We take as our inspiration the language StreamIt. However, instead of developing a new language we implement the structured graph primitives of StreamIt as classes within a C# library. We will give a brief overview of this library before describing how we compile it to efficient code on a range of platforms. In particular we describe how such programs can be compiled into code suitable for execution on an FPGA, with data being streamed to and from an enclosing C# application.

2. S#

S# is a library built on top of C#, together with a small number of support tools. The initial inspiration for the project was the StreamIt language, developed by MIT a decade ago [Thies 2009][Gordon 2010]. StreamIt is an example of a synchronous dataflow (SDF) language [Lee and Messerschmitt 1987], a restricted form of process network [Kahn 1974] in which nodes execute atomic steps, and the numbers of data items produced and consumed during each step are constant and known at compile time. StreamIt also supports a peeking operation to allow code to look ahead in the input stream without consuming the data, and the extent of the peeking is also known at compile time.

When replacing a domain-specific language by a library in an existing language you often have to trade-off conciseness and expressibility. However, C# is an ideal candidate for such a role given the wide range of syntactic mechanisms available to us, such as named parameters, properties, lambdas, and generics. By exploiting all these capabilities we have produced a system that is similar in conciseness and readability to programs written in the StreamIt language.

2.1. Structured Graphs

Dataflow approaches typically fall into two camps, graphical and textual. In a graphical approach, such as SystemVue [Agilent 2013] or Simulink [MathWorks 2013], it is typically left to the user to build graphs with a manageable structure. The user adds graph components from a palette and is then largely free to wire them up in any way they wish, subject to typing constraints. We could mimic such an approach in a textual language as well, manually linking the individual components together. The development of structured programming techniques and language constructs largely removed the need for goto statements in code, leading to clearer and more manageable programs. The same idea is applicable to dataflow algorithms. In many cases we can build our graphs using a small set of graph constructors. Starting with atomic graphs, known as *filters* in StreamIt terminology, we can construct larger graphs using pipelines, split/joins and feedback loops, as shown in Figure 1. Each filter takes a single stream as input, and produces a single stream of results. Each of the composite graphs also has this property, allowing us to construct large graph hierarchies from a small set of primitives.

2.2. Examples

Each of the constituents of a stream graph, the filters and the composite graph constructors, are mapped to classes in S#. The framework is statically typed with each class inheriting from a common base class `StreamGraph<I,O>`, where `I` is the type of the data arriving on the input stream and `O` the type of data produced for the output stream. For conciseness we omit details of the library, instead just presenting a couple of small examples.

In line 1 of Figure 2 we declare a new C# class representing a filter that reads a stream of floats from the input and produces a stream of floats on the output. The filter is called repeatedly to process the input stream, each time executing the “work”

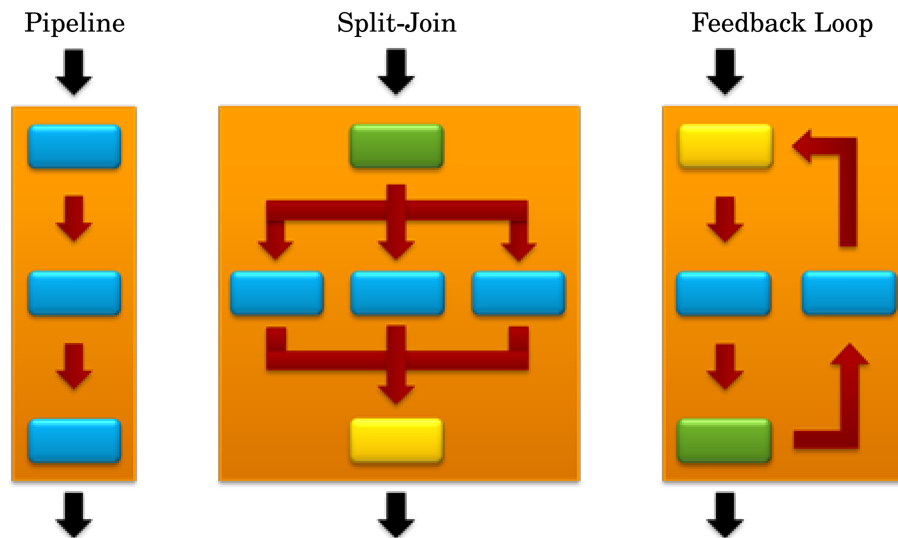

Fig. 1. Structured stream graphs.

```
1  public class FIR : Filter<float, float> {
2    private readonly float[] taps;
3    public FIR(float[] taps)
4      : base(peek: taps.Length, pop: 1, push: 1) {
5      this.taps = taps;
6    }
7    protected override void Work() {
8      float sum = 0.0f;
9      for (int i = 0; i < taps.Length; i++)
10       sum += Peek(i) * taps[i];
11     Pop();
12     Push(sum);
13   }
14 }
```

Fig. 2. A simple filter

function defined on lines 7..13. S# is a synchronous dataflow language[1] and so we must declare the rate at which data is consumed and produced each time the work function is executed. This information is declared on line 4. Finally, within the work function we use the predefined functions `Peek`, `Pop` and `Push` to access the input stream and push results on the output stream.

Our second example, shown in Figure 3, illustrates how to build a composite graph. We use inheritance to define the type of the composite graph, in this case a Split-Join component. Just as with the filter, we must declare the types of the values traversing the input and output streams. A split-join graph requires a splitter, one or more subgraphs, and a joiner. Line 3 defines the splitter in this example to be a duplicate splitter; every value received by the split-join will be passed to every child subgraph. Line 8 in the example defines the joiner to be a round-robin joiner. It will output the

[1] We do allow some rates to vary, but this is the exception rather than the norm.

value from the first child, then the second, in a round-robin fashion. Finally we must add the children that make up the branches of the split-join component. In this example the number of children is dependent on an argument to the constructor, and so we add the children using a for loop.

```
1  public class PolyphaseInterpolator : SplitJoin<float, float> {
2    public class PolyphaseInterpolator(int factor, float[] taps) {
3      Splitter = new Duplicate<float>();
4      for (int i = 0; i < factor; i++) {
5        float[] phaseTaps = . . . ;
6        Add(new FIR(phaseTaps));
7      }
8      Joiner = new RoundRobin<float>();
9    }
10 }
```

Fig. 3. A simple split-join graph

We can define classes for pipelines and feedback loops in a similar fashion. For large components, particularly where they are potentially reusable in other examples, it makes sense to define them as new classes. However, in some cases it can be more concise to define anonymous components. For example, we can instantiate an instance of the `Filter` class directly, passing a lambda expression to use for the Work function.

2.3. Support Classes

Our intention is for the design of an S# program to be largely independent of any target platform. However, there are some situations where we relax this ideal. For example, in C# if we need to store an integer we would have a choice of types, but each one is constrained to be a power-of-two bytes in length. FPGAs have no such constraints, and so if we simply mapped a C# integer to a 32-bit bus on an FPGA this would often waste resources. To avoid this, S# provides tools that allow us to construct a family of fixed-width integer and fixed-point classes in C#. These types emulate the appropriate width arithmetic when executed in C#, but are compiled to fixed-width buses and registers when compiled to an FPGA.

Generic arithmetic is another area requiring specialized support in S#. Consider the FIR filter defined in Figure 2. It manipulates floats, and so if we wanted a FIR filter that processed a stream of doubles, or ints, we would need to define some additional classes. This is clearly not ideal, and generic types were partly introduced in C# to avoid such code duplication. We could exploit the availability of generic types by defining a class

```
class FIR : Filter<I,O> where I : struct where O : struct { ... }
```

S# passes data by value, and so the type parameters `I` and `O` must be constrained to be value types, such as ints, floats and structs. Unfortunately this change would result in compilation errors as operations like the multiplication on line 10 of Figure 2 cannot be applied to values of generic types. To avoid such problems we use the `Operator` class [Gravell 2009], replacing expressions like `Peek(i) * taps[i]` by

```
Operator.Multiply(Peek(i), taps[i])
```

At runtime such expressions are compiled on the fly to efficient code. And when the code is compiled to C++, or a hardware description language, then the expressions are

optimized to the appropriate multiplication instructions as at this point the types of the generic arguments are known.

2.4. Parameters

One other feature deserves mention at this point, as it impacts on the compilation process described later. Frequently we want to be able to change certain characteristics of our filters after a graph has been constructed. For example, in our FIR filter we might wish to change the filter coefficients. To support such behavior we allow the user to annotate fields within the filter with the `[Parameter]` attribute. The S# runtime provides method to find such fields within a graph, and get/set the values of these fields.

3. EXECUTION

Having built a stream graph we now have to execute it. S# defines a `StreamProcessor` class that takes stream graphs, analyses them and then executes the result. Now, ignoring safety for the moment, a simple-minded execution strategy would be to execute each graph worker, the filters, splitters and joiners, in a separate thread, connected via asynchronous buffers. Unfortunately, due to the fine-grained nature of the stream graphs, the communication overheads of reading and writing to these buffers would tend to dominate the execution speed. Furthermore, we would not be exploiting an important source of information, the declared filter rates. Consider, for the moment, a graph where all the rates are fixed. For such a graph we can construct an execution schedule [Karczmarek 2002], an ordered list of firings of nodes in the graph. Every firing of a node consumes some data from input channel(s) and pushes data onto the output channel(s). A schedule is well-formed if the amount of data buffered up between any two nodes does not change from before to after the execution of the schedule.

In reality multiple schedules can be constructed, each one trading off schedule/code size for buffer sizes/latency. And for some graphs, for example where the rates in a split-join are inconsistent, then no schedule is possible. Such graphs are rejected by S#. For well-formed graphs we can compute accurate sizes for each of the buffers connecting the workers. Furthermore, by executing the workers in the order dictated by the schedule, we can guarantee no buffer underflows or overflows can occur, and so we can avoid performing such checks when executing the graph. In practice we need two schedules, an initialization schedule that is executed just once, and a steady-state schedule which is executed indefinitely from then on. The purpose of the initialization schedule is to “prime the pumps”, ensuring each buffer has enough data in it at the start of the steady-state schedule to support subsequent peeking operations.

Whilst SDF graphs are convenient as they support an efficient execution model, they are at times too restrictive. S#, just like StreamIt, also allows some input and output rates to be variable but bounded. Judicious use of such rates allows us to target a wider range of applications without needing to introduce “dummy” values, with all the inefficiencies and inelegance associated with them. A graph containing links with variable rates is partitioning into a collection of SDF subgraphs, connected by asynchronous links. Each subgraph is then executed as a separate thread running a static schedule.

4. VERIFICATION

Embedding a domain-specific language (DSL) within an existing language has a lot of advantages. For example, in the case of S# it allows us to use the full power of the host language C# when building our stream graphs. However, in some places, for example within the body of a filter, this power can be undesirable. Without additional constraints there is nothing preventing a user from manipulating a global variable from within the body of a filter, for example. And as the user has no control over the

execution order of the filters, other than those implicitly defined by the data stream dependencies, the results in such cases are undefined. We would also like to map our stream graphs to other architectures, such as FPGAs, where activities such as allocating data on the heap, using exceptions, and invoking explicit synchronization primitives will have no efficient counterpart. So for both safety and platform independence we would like to constrain the power of the host language where it is used inside the definition of a filter.

There are various approaches we could follow to impose such constraints. Perhaps the most obvious is to use a preprocessor. This would read in ”S#” programs and then pass them to the C# compiler if they satisfied the constraints. Unfortunately the design of C# makes it difficult to process files in isolation, and so a preprocessor would require some knowledge of the build process to have the complete picture. Furthermore, anything that runs prior to the compilation step has to cope with invalid programs.

Another approach to enforcing localized language constraints would be to modify a compiler to add support for S#. Unfortunately we cannot modify the Microsoft C# compiler, and many developers would be reluctant to use an open-source alternative such as Mono [Xamarin 2013]. There is a third alternative. The approach adopted by S# is to run a post-compilation verification step. We wait until Visual Studio, or MonoDevelop, has constructed the assembly or executable, guaranteeing the program is well-formed. We then run a verifier on the result. This tool loads the assembly, and then uses .NET's reflection mechanism to locate all the filters defined in the assembly. It is the code within these filters that we wish to constrain. At this point in the analysis the code is represented by sequences of Common Intermediate Language (CIL) instructions [Microsoft 2012]. We could traverse these instructions checking for constraint violations. However, there is a better way. As part of the ILSpy application [ICSharpCode 2013] the Mono group has developed a set of decompilation libraries. These can take CIL instruction sequences and convert them into an equivalent high-level representation of the code, for example in C#. We use these libraries to construct a C# view of the filter code, and any functions it calls. We can then traverse this high-level representation to check that the code satisfies our constraints. If the assembly passes these checks it is tagged; the S# runtime will only run verified assemblies. This approach to enforcing library-specific language subsets works well, and is potentially applicable to a wider range of applications than just S#.

5. COMPILATION

We now have an execution strategy that is safe and reasonably efficient. But there is still room for improvement. As described so far, we process a stream of input data by continually executing a schedule. And to execute a schedule we iterate through it calling each worker in turn. This introduces two overheads. First, we must make a function call to execute each worker, and this can be a non-negligible overhead for small workers. Furthermore, the C# compiler is unable to perform any cross-filter optimizations. For example, in our schedule filter A might always be followed by filter B. Filter A might compute a value, double it, and push the result onto the output buffer. Filter B might fetch a value from the input buffer, halve it, and then perform some additional computations. There's clearly scope for optimizing such behavior, but such opportunities are not visible to the C# compiler.

As described in Section 4, decompilation techniques can be used to build a high-level representation of the code for a method. Using such an approach we can take a schedule and concatenate (a specialized version of) the code for each worker in the order it appears in the schedule to produce a new method. During this process we can optimize the buffer representations, for example using simple scalars where appropriate. The result is a data structure representing the new method; we still have to compile

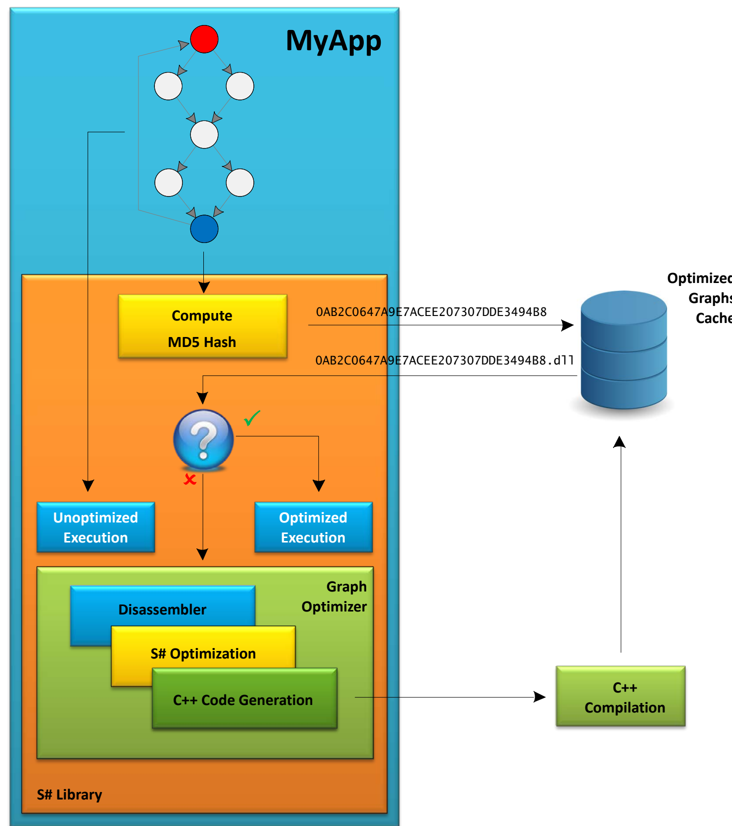

Fig. 4. The graph cache

it before we can use it. And we have a choice as to which compiler to use. We could simply output the result as C# code and use a C# compiler. However, for increased efficiency, we currently rewrite the data as C++ code and use a C++ compiler. In general this step would be non-trivial. However, given the restrictions placed on the code that can appear within a filter's work function the translation in this instance is largely straight-forward.

Compiling the code resulting from a static schedule is reasonably fast, even when using an external compiler, but it is not instantaneous. Furthermore, it may not be worth compiling the entire graph, only one or more compute-intensive subgraphs. In many programs the overall graph structure may change during the execution of a program, or between executions. But it is often possible to identify stable subgraphs, graphs that are reused as components over and over again. In such cases the main graph is built by assembling a set of such subgraphs in different configurations over time. We therefore introduce the notion of a graph component that encapsulates this notion of stability. If a graph is likely to be used frequently then it is worth expending the time and effort to compile it. We therefore use the construction of a graph component to trigger the compilation process described earlier. To avoid repeatedly recompiling the same graph component we use a caching mechanism. This process is illustrated in Figure 4. To convert a graph to a hash we use a two-step approach. We first serialize the graph into a byte sequence, and then compute an MD5 hash [Rivest 1992] from this sequence. We

use a custom serializer which allows us to control which fields to serialize, and therefore which ones should affect the resulting hash value and therefore when two graphs should be viewed as equivalent.

Some graphs are closed, in the sense that the first element of the graph is a source filter requiring no values from the input stream, and the last element is a sink filter producing no values for the output stream. If the entire graph is compiled into C++ code then we can simply execute the resulting DLL without linking it to a surrounding graph context. More commonly we only compile subgraphs, and so we must establish links from the C# host graph to the subgraph. This involves constructing marshaling and unmarshaling code to convert a stream of C# data values to and from the equivalent C++ values. The compilation process generates the appropriate marshalers and unmarshalers in the C++ code, and the S# runtime dynamically creates matching code for the C# environment. This allows us to embed compiled subgraphs within C# graphs with data flowing back and forth in a transparent fashion. The code for the enclosing graph is independent of whether the subgraph is implemented in C#, C++, or is running on an FPGA.

6. COMPILING TO AN FPGA

Compiling to a multicore platform can be challenging. We typically need to adjust the parallelism to match the number of cores available, using fusing and fissing [Gordon 2010]. Or else we must map the computations to a task-based approach [Reinders 2007], which introduces its own set of inefficiencies.

In contrast, fine-grained streaming algorithms are a particularly good fit to FPGAs. The S# system provides an alternative backend that can be used to target such systems. This allows us to develop and debug streaming applications in C#, either on Windows or under Linux using Mono, where the full power of modern development environments is available to us. We can then choose to compile all or parts of the graph into either C++ or firmware using the same code base.[2] The runtime automatically takes care of loading the compiled components and streaming the data between them.

The starting point for the FPGA backend is similar to the C++ one. We partition the graph into subgraphs that can be statically scheduled, and use decompilation to construct high-level representations of each worker. From such representations we could generate code for a hardware description language, such as VHDL or Verilog. However, S# adopts a simpler approach, targeting the Xilinx®Vivado High-Level Synthesis (HLS) tool [Xilinx 2013]. This tool supports ANSI-C (with C99), C++ and SystemC. Vivado HLS also provides a C++ template class, `hls::stream<>`, for modeling streaming data structures. We use this feature to connect the synchronous blocks, in a similar fashion to our use of asynchronous buffers in the C++ backend. In the RTL streams are implemented as either a FIFO or full handshake interface port. We use FIFOs for the interior streams, and handshaking ports for the external interface to the graph.

6.1. Graph partitioning

Whilst Vivado HLS can take C++ code as input, simply reusing our existing C++ backend as a starting point will not lead to efficient utilization of the FPGA. For example, consider the case where we start with a fixed-rate graph, and a static schedule for this graph. A high-level view of the execution step would then look like

[2] In a few cases we may need to annotate the code with some pragmas to help guide the compilation process.

```
bool inSteady = false;

void process(hls::stream<float>& input, hls::stream<float>& output) {
  if (!inSteady) {
    // code for initialization schedule
    inSteady = true;
  } else {
    // code for steady schedule
  }
}
```

Unfortunately the synthesis tool needs to compute the number of clock cycles required for each block of code, and in the case of a conditional this will be the maximum of the cycles required for each branch. Thus if the initialization branch requires more cycles than the steady-state branch then the effect will be to slow down the steady-state branch, even though the initialization branch is executed just once. Such behavior is common. Consider a graph containing the FIR filter defined earlier. In the initialization phase we must execute the upstream workers repeatedly to generate enough data in the FIR filter's input buffer to satisfy subsequent peeking operations. But in the steady state the upstream workers might only need to fire once in each execution of the schedule. Thus the number of register reads and writes can often be greater in an initialization schedule than a steady-state schedule. In the case of the C++ backend we typically partition the graph into a small number of SDF subgraphs, assuming we are targeting a relatively small number of cores, as this minimizes buffering overheads. In contrast, for the FPGA backend it is more efficient to construct a larger number of simpler subgraphs to ensure the number of cycles required to execute the initialization schedule for each graph is guaranteed to be less than the steady-state execution. Or at the very least can be easily split into a number of stages, each one requiring no more cycles than the steady-state execution. Our approach is to partition the graph so we can split the initialization phase of each partition into N steps, each one requiring a single clock cycle. The generated code then looks like

```
int stage = 0;

void process(hls::stream<float>& input, hls::stream<float>& output) {
  if (stage == N) {
    //  code for steady schedule
  } else {
    switch (stage) {
       case 0:  // code for initialization stage 0
       case 1:  // code for initialization stage 1
       ...
    }
    stage++;
  }
}
```

6.2. Initiation intervals

The initiation interval is the number of cycles between the start of one function or loop and the next. If it is not specified for a block then the High-Level Synthesis tool will seek to minimize the initiation interval and start operation as soon as data is available. One of the goals of S# is to allow essentially the same code base to be reused across a range of platforms. However, at times the system needs help where there are tradeoffs between speed and resource utilization, and where the intentions and priorities of the user cannot be inferred from the code. Our FIR filter is a good example of this situation. When compiling such a graph component down to gates we have a choice.

At one extreme we might attempt to execute the work function in a single clock cycle. For small examples this can be possible, but may require a lot of resources from the FPGA. At the other extreme we can minimize the resource requirements by splitting the computation over multiple cycles and reusing the same multiplier in each cycle, for example. There are also positions between these two extremes. A compiler has no way of knowing which of these options is appropriate for the current application. We might argue that the system should choose the fastest option possible whilst allowing the overall design to fit within the FPGA. Unfortunately at the time such decisions need to be made it is hard to estimate the resources required by the whole design with any degree of accuracy. Furthermore, if there are multiple subcomponents that can each be implemented with different tradeoffs then without outside guidance we would have to treat them all with the same priority.

A better alternative is to allow the user to explicitly specify their intentions in such cases. Each S# graph node maintains a set of properties that can be set by the user and can influence the compilation behavior. The Initiation Interval (II) property allows the user to specify the minimum initiation interval to use when compiling the node with the FPGA backend. Consider the compilation of the FIR filter. By default, if no property is attached to an instance of this filter then the II is set to one, that is the system will attempt to schedule the block in one clock cycle. However, by specifying larger values for the II then we instruct the compiler we wish to trade of speed for resources.

Most S# graphs use components with fixed rates. Such SDF components are essential for efficient execution. However, taken to extremes this is too restrictive. For example, in the short-read matching example described in Section 7 one of the components produces no output at all most of the time. But occasionally it may produce up to two outputs. So in the worst case the compiler may need to set the II value to two as we cannot push two values in one clock cycle. However, in such cases it can be preferable to output one value, and then output the remaining values on successive clock cycles. Once the outputs have been flushed we return to processing data from the input FIFO. The S# compiler automatically performs such optimizations when appropriate.

6.3. Peeking

S# supports a peeking operation, allowing an application to look ahead in the input stream. This helps reduce the number of filters that need to maintain state. For example, the equivalent FIR filter without peeking would need to maintain a FIFO history of read values, and such state would make it difficult to replicate the filter, for example to increase potential parallelism. To support peeking we need to construct a peek buffer that is populated with data from the input stream during the initialization phase. During the steady-state phase we push N new values into the peek buffer, where N is the declared pop rate, and then access values within the peek buffer when executing the work function. An obvious representation choice for the peek buffer would be a shift register. This allows us to access multiple elements in the buffer at the same time in one clock cycle. However, for large peek buffers this takes up a lot of resources. In such cases there is an alternative representation, an `ap_shift_reg`, which is implemented by a Xilinx SRL resource. This is much more compact, but we can only access one element per clock cycle, in addition to shifting a new value. So if we peek at lots of elements then it can be slow as we will require multiple clock cycles to execute the work function. The S# compiler analyses the peeking patterns within the code, and chooses the `ap_shift_reg` representation when it doesn't increase the initiation interval, and uses a shift-register implementation otherwise.

6.4. Marshaling

We use AXI4 Streams [Xilinx 2012] as the input and output stream types. These process data in 64-bit chunks and so we need to handle the marshaling of data between C# and the FPGA. The situation is similar to the case with the C++ backend. However, in this case there is no requirement for byte alignment of the values. For example, in a gene sequence matching example we might represent a DNA base by an **enum**, typically requiring 32-bits in C#. But this data will be marshaled into 2 bits in the C# code interfacing to the FPGA and so the unmarshaler in such a case would read one 64-bit chunk and then output 32 bases. Similarly fixed-point and arbitrary precision types get marshaled into their natural bit widths. The data also has to be marshaled and split into 64-bit chunks at the exit of the FPGA. Figure 5 illustrates the top-level structure of an S# component compiled for an FPGA.

6.5. Interfacing to the host computer

The output from a tool like Vivado HLS is just an IP block. We need still need some additional "glue" to connect it to the host PC. We could use a framework such as Xillybus [Xillybus 2013] to bridge the gap. This solution is ideal if we are not too concerned about filling the FPGA, or we can adapt our algorithms, for example through replication, to fit the size of the FPGA. For other cases the situation is not as simple. The application may require multiple compiled subgraphs, each one substantially smaller than the capacity of the FPGA. We could take the approach of OpenCL on FPGAs [Altera 2013] where all the kernels are compiled into a single bit file. But this is very restrictive, and a better solution is to use partial reconfiguration [Dye 2012]. We partition the FPGA into N regions, $N \geq 1$, and allow each region to be reconfigured independently. A switching fabric allows the CPU to stream data to and from each region independently. A group within Agilent's research labs has developed such a framework, and we use this system to support S#. The framework uses a PLDA EZDMA IP core with a custom scatter/gather controller for increased performance. The overall architecture is illustrated in Figure 6.

We refer to each reconfigurable region as a sandbox. Bit streams are not relocatable, and so we must compile each graph component potentially multiple times, once for each sandbox we wish to target. We package up all the bit files for these sandboxes into a single dynamic-link library (DLL), with the bitfiles represented as resources. In most cases such a DLL will be loaded into a C# program by the S# runtime. However, we could also use it in environments where S# does not exist. In such cases we can view S# as another high-level synthesis tool. The tool chain, from the S# code to the compiled DLL containing the bit files, is completely automated. This allows software developers unfamiliar with FPGA tools to accelerate their streaming applications on such platforms without requiring access to a firmware specialist in many cases.

Section 2.4 briefly discussed the parameter mechanism supported by S#. When a graph containing parameters is compiled to an FPGA we must route any changes to such parameters into code that performs the corresponding modification in the FPGA. In addition to the input and output streams our graphs take an additional register argument, implemented as an AXI4 Lite Slave interface. S# collects together all the parameters defined in the graph being compiled, and builds a data structure containing a field for each parameter. The additional register argument passed to our top-level function in Figure 5 is a reference to this structure. The Vivado HLS tool generates an address map for such arguments. This defines an address for each component of the register structure. The map is stored in the DLL containing the bit files and API calls provided by the interfacing framework are used by the S# runtime to translate parameter updates to instructions that update register addresses. On the FPGA side we

```
struct AXIBUS {
  ap_uint<64> tdata;
  ap_uint<8> tuser;
  ap_uint<4> tdest;
  ap_uint<8> tid;
  ap_uint<1> tlast;
};

void graph(hls::stream<Base>& input, hls::stream<Match>& output, Regs& regs) {
  // Code to process user graph
}

void unmarshall(hls::stream<AXIBUS>& input, hls::stream<Base>& output) {
  #pragma HLS pipeline II=32
  AXIBUS bus = input.read();
  ap_uint<64> data = bus.tdata;
  for (unsigned int i = 0; i < 32; i++) {
    #pragma HLS unroll
    ap_uint<2> v = data.range(2*i+1,2*i);
    output.write(Base(v));
  }
}

void marshall(hls::stream<Match>& input, hls::stream<AXIBUS>& output) {
  #pragma HLS pipeline II=1
  ap_uint<64> buffer;
  Match v = input.read();
  buffer.range(31,0) = *(unsigned int*)&v.offset;
  buffer.range(37,32) = v.cost;
  AXIBUS bus;
  bus.tdata = buffer;
  output.write(bus);
}

void top(hls::stream<AXIBUS>& input, hls::stream<AXIBUS>& output, Regs& regs) {
  #pragma HLS resource core=AXI4Stream variable=input metadata="-bus_bundle IS"
  #pragma HLS resource core=AXI4Stream variable=output metadata="-bus_bundle OS"
  #pragma HLS resource core=AXI4LiteS variable=regs metadata="-bus_bundle REG"
  #pragma HLS dataflow
  #pragma HLS pipeline II=1
        ...
  static hls::stream<Base> graphInput;
  static hls::stream<Match> graphOutput;
  unmarshall(input, graphInput);
  graph(graphInput, graphOutput, regs);
  marshall(graphOutput, output);
}
```

Fig. 5. Top-level structure of a Vivado HLS S# component

monitor changes to the register fields, and update the state of the FPGA when a field changes.

7. PERFORMANCE EVALUATION

When evaluating the performance of a system like S# there are a number of different criteria we might consider. The S# code required to describe a graph is broadly similar in size to the StreamIt version. The code for filters is slightly more verbose, but we can often write more concise code when building the graph structure itself as we have access to the more powerful language features of C#. We therefore refer the reader interested in the conciseness and expressibility of the language to the StreamIt litera-

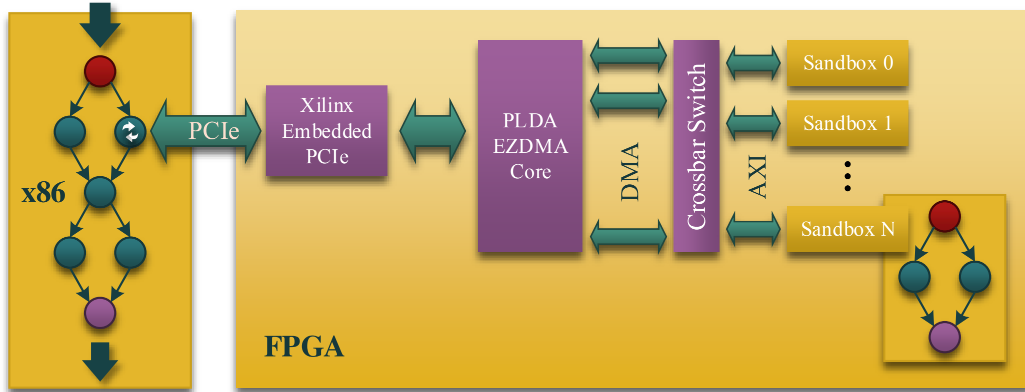

Fig. 6. The S# FPGA architecture

ture, for example [Thies 2009]. In this section we will focus on the performance of the system on a real problem.

The short-read gene sequence alignment problem involves matching a large number of short DNA sequences, each perhaps a few hundred bases in length, against a reference sequence containing potentially millions of bases. The matching is approximate, with a cost for each insertion, deletion and replacement. The task can be viewed as a dynamic programming problem, as described in [Smith and Waterman 1981]. Such problems map well to a streaming framework. This example also illustrates why judicious use of variable rates can be useful. We are typically interested in locations in the reference sequence that can be made to fit a short-read sequence by editing operations whose cost is below some threshold. Thus whilst at the start of the processing pipeline the rate at which the workers run is directly linked to the rate at which data is injected into the pipeline, in later stages of the graph the data rates may be much lower. The execution strategy is simple. We stream in a short-read sequence to configure the matching pipeline, and then stream the reference sequence through it to find all the matches. We then repeat this process with the next short-read sequence, followed by the reference sequence, until we have processed all the short-read sequences. Such problems are also simple to extend to multiple matchers running in parallel. Suppose we want to use N matchers. We simply combine them in a split-join graph with a duplicating splitter. We now feed in N short-reads followed by the reference sequence. Each branch picks the short-read it should match against, ignoring the others. The joiner is a little more complicated. It reads all the matches from the first matcher until the end of the reference sequence is detected. It then proceeds to the next branch in a round-robin fashion.

The ability to scale the solution, by adding additional matchers, makes it easy to choose an implementation that fills our FPGA resources. For example, if we know our short-read sequences are no more than 32 bases in length then we can fit 24 parallel matchers on a HiTech Global board containing a Virtex 6. Figure 7 illustrates the range of speeds that can be achieved on such a problem using the different implementation strategies supported by S#. In each case we are using exactly the same code base, but augmented with pragmas to help the compiler as discussed earlier. The three columns on the right illustrate the performance obtained when the example is run in parallel on a 24-core Z820 workstation. The right-most column demonstrates the price we pay for emulating fixed-point arithmetic in C# on a PC. The algorithm is numerically intensive, and so the emulation costs dominate the performance. Traversing to

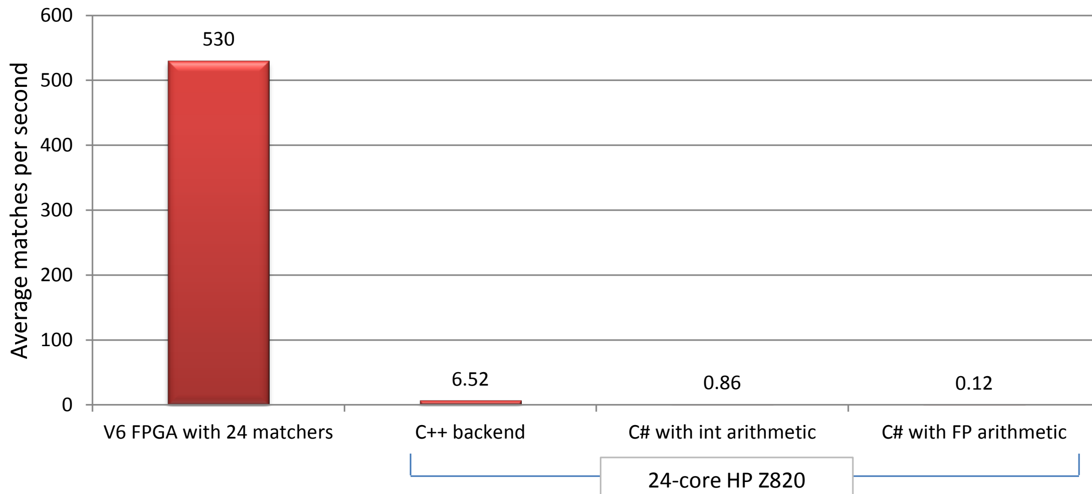

Fig. 7. S# short-read matching performance comparison for different platforms

the left, the next column shows the performance we get by using a one-line change to replace the fixed-point type by an int. The next column shows the performance that can be achieved using the C++ backend. At present such a figure under-represents the potential of this backend as little effort has been made to optimize the code generator for this case. Finally, the left-most column shows the performance that can be achieved using the FPGA backend. In each case the subgraphs are embedded in an identical C# graph that supplies the data and consumes the results.

8. CONCLUSIONS AND FUTURE WORK

Whilst the S# system is still under development, the initial results are encouraging, particular with the FPGA backend. There is more work to be done on the multicore C++ backend to make it competitive. By leveraging the existing work on StreamIt we should be able to substantially increase the speed of the code generated by the current C++ backend. This will involve more intelligent partitioning, code replication (fissing) and aggregation (fusion), together with a more optimized code generator. Ideally we would also like to target GPUs. Some existing work has been done in the context of StreamIt [Hagiescu et al. 2011][Huynh et al. 2012]. The results were encouraging when the majority of filters were stateless. However, in most of our examples this assumption does not hold, and so at this point it is unclear how efficient such a backend could be.

The StreamIt language supports a teleport messaging system which allows out-of-band messages to be delivered both upstream and downstream in a deterministic fashion. The prototypical example of the need for such a capability is a frequency hopping radio, where a transmitter and a receiver switch between a set of known radio frequencies, and they do so in synchrony with respect to a stream boundary. The receiver must switch its frequency at an exact point in the stream (as indicated by the transmitter) in order to follow the incoming signal. When a hop is detected, the detector must send a message upstream that must be delivered precisely with respect to the data stream. The S# system does not support teleport at present, but could do so in principle.

The FPGA backend works well when there is a single static algorithm running on the device. However, in many cases we will need a better scheduling strategy. At present we use up the FPGA sandboxes on a first-come/first-served basis. This is not ideal, par-

ticularly when we may not have compiled every subgraph for every available sandbox, or where some subgraphs will not fit in some of the sandboxes. Furthermore we may end up with two subgraphs assigned to different FPGAs that are linked together in a pipeline. In such cases it may be preferable to allocate them both to separate sandboxes on the same FPGA, with the routing done internally in the switching fabric.

At present the S# framework has no mechanism for suspending the execution of a graph, for example to allocate the sandbox to a different subgraph. This operation is problematic as we would need to flush the stream and retrieve any local state. When the computation was resumed we would then have to reload this state. It would be difficult to support such an operation using just the FPGA infrastructure, and so such a capability would also require the assistance of the S# FPGA code generator.

The goal of ”write-one, run everywhere” in this context is not unique to S#. The OpenCL initiative [Khronos 2013] has similar goals. At present OpenCL appears more suited to the kinds of block-based computation that map naturally to execution on a GPU. However, streaming extensions to OpenCL are currently under development which may make such a framework more suited to the kinds of application targeted by S#. The OpenCL implementations currently under development for FPGAs, for example [Altera 2013], require all kernels to be preloaded as there is no support for partial reconfiguration. This may also limit their usefulness for some applications. OpenCL requires users to express their algorithms in a kernel language, a variant of C99 with some limitations and additions. The kernels are then deployed from a host language, such as C++ or Java. This contrasts with the approach of S# where the same language is used for both tasks. In principle, the decompilation approach used by S# could also be applied to OpenCL[Mitchell 2013].

ACKNOWLEDGMENTS

The author would like to thank T. Vandeplas, Z. Hendrix and M. Berdondini of Agilent Technologies for providing FPGA advice and access to their partial reconfiguration framework.

REFERENCES

Agilent. 2013. SystemVue Electronic System-Level (ESL) Design Software. (2013). http://eesof.com/

Altera. 2013. OpenCL for Altera FPGAs. (2013). http://www.altera.com/products/software/opencl/opencl-index.html

I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. 2004. Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH*.

David Dye. 2012. *Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite*. Technical Report. Xilinx. http://www.xilinx.com/support/documentation/white_papers/wp374_Partial_Reconfig_Xilinx_FPGAs.pdf

Michael I. Gordon. 2010. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*. Ph.D. Dissertation. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Marc Gravell. 2009. Generic Operators. (February 2009). http://www.yoda.arachsys.com/csharp/miscutil/usage/genericoperators.html

Andrei Hagiescu, Huynh Phung Huynh, Weng-Fai Wong, and Rick Siow Mong Goh. 2011. Automated architecture-aware mapping of streaming applications onto GPUs. In *Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 467–478.

Huynh Phung Huynh, Andrei Hagiescu, Weng-Fai Wong, and Rick Siow Mong Goh. 2012. Scalable framework for mapping streaming applications onto multi-GPU systems. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 1–10. DOI:http://dx.doi.org/10.1145/2145816.2145818

ICSharpCode. 2013. ILSpy .NET Decompiler. (2013). http://ilspy.net/

Gilles Kahn. 1974. The semantics of a simple language for parallel programming. *Information Processing* (1974), 471–475.

U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. 2003. Programmable stream processors. *IEEE Computer* (2003).

Michal Karczmarek. 2002. *Constrained and Phased Scheduling of Synchronous Data Flow Graphs for StreamIt Language*. Master's thesis. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Khronos. 2013. OpenCL: The open standard for parallel programming of heterogeneous systems. (2013). http://www.khronos.org/opencl/

Edward A. Lee and David G. Messerschmitt. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computing* 36, 1 (1987), 24–35.

W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. 2003. Cg: A system for programming graphics hardware in a C-like language. *SIGGRAPH* (2003).

MathWorks. 2013. Simulink. (2013). http://www.mathworks.com/products/simulink/

Microsoft. 2006. C# Language Specification. ECMA-334. (June 2006). http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf

Microsoft. 2012. Common Language Infrastructure (CLI). ECMA-335. (June 2012). http://www.ecma-international.org/publications/standards/Ecma-335.htm

Kevin Mitchell. 2013. Generating OpenCL Kernels via Decompilation. HiPEAC European Network of Excellence on High Performance and Embedded Architecture and Compilation. (February 2013). http://www.hipeac.net/content/generating-opencl-kernels-decompilation

James Reinders. 2007. *Intel threading building blocks*. O'Reilly & Associates, Inc.

R. Rivest. 1992. The MD5 Message-Digest Algorithm. (April 1992). http://tools.ietf.org/html/rfc1321

Temple Smith and Michael Waterman. 1981. Identification of Common Molecular Subsequences. *Journal of Molecular Biology* 147 (1981), 195–197.

William Thies. 2009. *Language and Compiler Support for Stream Programs*. Ph.D. Dissertation. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

William Thies, Michal Karczmarek, Michael Gordon, David Maze, Jasper Lin, Ali Meli, Andrew Lamb, Chris Leger, and Saman Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *New England Programming Languages and Systems Symposium (NEPLS)*.

Xamarin. 2013. Mono. (2013). http://www.mono-project.com

Xilinx. 2012. AXI Reference Guide. (November 2012). http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug761_axi_reference_guide.pdf

Xilinx. 2013. Vivado High-Level Synthesis. (2013). http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/index.htm

Xillybus. 2013. Xillibus: An FPGA IP core for easy DMA over PCIe with Windows and Linux. (2013). http://xillybus.com/