ScalaPipe: A Streaming Application Generator

Joseph G. Wingbermuehle
Roger D. Chamberlain
Ron K. Cytron

Joseph G. Wingbermuehle, Roger D. Chamberlain, and Ron K. Cytron, "ScalaPipe: A Streaming Application Generator," in *Proc. of Symposium on Application Accelerators in High-Performance Computing* (SAAHPC), July 2012.

Dept. of Computer Science and Engineering
Washington University in St. Louis

ScalaPipe: A Streaming Application Generator

Joseph G. Wingbermuehle, Roger D. Chamberlain, and Ron K. Cytron
Dept. of Computer Science and Engineering, Washington University in St. Louis
{wingbej,roger,cytron}@wustl.edu

***Abstract*—ScalaPipe is a streaming application generator for heterogeneous platforms. By using a collection of domain-specific languages (DSLs) embedded in the Scala programming language, ScalaPipe allows creation of streaming applications that can run on a variety of hardware, including traditional processors, graphics processors, and field-programmable gate arrays (FPGAs). Its *application DSL* allows specification of the application topology and resource mapping. Its *block DSL* allows the authoring of implementations for processing kernels, or blocks, which are used in the streaming application. ScalaPipe makes it easy to generate, modify, and instrument large, complex topologies and resource mappings while also exposing optimization opportunities.**

I. Introduction

The streaming approach to parallel programming is a popular programming paradigm. Examples of systems that use the streaming approach include Auto-Pipe [6], [11], S-Net [14], Streams-C [12], and StreamIt [29]. In the streaming paradigm, an application is decomposed into processing kernels, or blocks, which are connected using explicit communication channels.

Streaming applications enjoy many of the advantages of traditional programming paradigms [7]: the blocks of one streaming application can be deployed in another (reuse); and application developers need not typically be concerned with the implementation details within a block (abstraction). Languages that support streaming applications typically offer further support for the pipelined nature of such applications. For example, the buffers that must be provisioned between blocks may be statically determined so that all blocks can compute their outputs without deadlock [19]. Finally, because performance is usually paramount for streaming applications, developers are given some control over the mapping of an application's blocks to the resources available on a platform, such as traditional processors, graphics processors, and field-programmable gate arrays (FPGAs). These diverse resources encourage a developer to consider alternative implementations for a given application block. The exigencies of a given platform then constrain how blocks communicate and influence the performance obtained by an application under a given resource mapping.

Extant toolsets that support the development of streaming applications therefore provide tools for developing the blocks of an application along with some mechanism for configuring and allocating resources to those blocks on a streaming platform. We observe that these two activities currently take place in completely separate languages and tools. In this paper, we point out the deficiencies of current approaches and we examine the benefits of using a single language to realize both activities.

Our primary basis for comparison is the Auto-Pipe toolset [6], [11]. In Auto-Pipe, blocks are authored in a language suitable for the deployment target. For example, a block targeted for a traditional processor might be implemented in C or C++ whereas a block targeted for an FPGA device might be implemented in VHDL or Verilog. Auto-Pipe's X coordination language is used to specify how the blocks are to be connected and to what resources they should be assigned. From the X language description, the Auto-Pipe X compiler generates the necessary run-time infrastructure to connect the blocks, which might be on separate resources.

The Auto-Pipe coordination language approach works well for many problems [5]. One can easily write streaming applications that run on both traditional processors and FPGA devices simultaneously without concern for the low-level details of data movement. However, Auto-Pipe suffers from the following deficiencies, which we illustrate by example in Figures 3, 4, 5, and 6:

1) Creating applications can require writing a great deal of X code, which makes developing and maintaining such applications tedious and error-prone. Indeed, an application of $O(n)$ blocks can contain $O(n^2)$ connections. The X language is not Turing complete and serves essentially to declare the configuration of the blocks. Because it lacks any generative power, each connection must be explicitly declared.[1] X specifications are not programs and thus cannot be extended or proceduralized in the manner of most programming languages.

   Figure 5 shows the connections between blocks for an implementation of Laplace's algorithm, which we describe in greater detail in Section V. Here we observe that the two `Walk` blocks and their connections are explicitly specified. A small change to increase the number of such blocks requires tedious changes to Figure 5. A better approach would allow the number of such blocks to be abstracted parametrically so that the blocks and their connections can be automatically generated.

This work is supported by the National Science Foundation under grants CNS-09095368 and CNS-0931693.

[1] An exception allows simple split-join topologies to be declared in X, but anything that is not directly supported must be declared manually on a per-edge basis.

2) Conceptually simple changes to the application or resource mapping can require many changes to the corresponding X language description. Figure 6 shows a particular mapping for the Laplace application; changes to the streaming topology or its mapping to resources necessitate modifying the specifications in Figures 5 and 6. As with the first point, this is due to the static nature of the X language, which requires that blocks and connections between blocks be explicitly mapped to resources.
3) Blocks cannot be polymorphic. The type declarations shown in Figure 3 specify all inputs and outputs as unsigned, 32-bit integers, but the Laplace algorithm could operate equally well on shorter or longer integers. In Auto-Pipe, completely separate implementations are required to capture Laplace algorithm variations that operate on integers of different sizes. A more attractive approach would allow type abstraction via polymorphism, so that a single specification suffices. This is difficult in Auto-Pipe because the blocks are implemented outside of the X language.
4) Blocks must be implemented separately for each target platform. For example, the Laplace algorithm's `Random` block may initially be implemented in C, but an implementation in Verilog may be subsequently desired for better performance. Auto-Pipe provides no common language for block implementations, so blocks must be re-implemented if a change in target is required. While ScalaPipe currently makes no attempt at generating an efficient hardware implementation and substantial algorithmic changes may ultimately be imposed, the ability to prototype an existing algorithm in hardware provides a starting point for examining the tradeoffs between the implementations.
5) Finally, communication overhead can become substantial even for blocks that are mapped to the same resource. Even after colocation, the blocks must nonetheless marshal their communication through the blocks' interfaces. This requirement follows from the separate implementation and coordination languages in Auto-Pipe.

To address the aforementioned issues, we developed a new system, ScalaPipe, to generate applications that use Auto-Pipe blocks as well as the Auto-Pipe blocks themselves. ScalaPipe is a set of domain-specific languages (DSLs) embedded in the Scala programming language [22]. Using Scala as the host language allows for much more expressive power when creating applications. Further, blocks implemented in the Scala DSL can be compiled to different targets (for example, traditional processors, graphics processors, and FPGAs) to generate families of blocks.

II. The ScalaPipe Approach

A ScalaPipe program is a Scala program that uses the ScalaPipe domain-specific languages to generate an Auto-Pipe application. ScalaPipe defines two main DSLs: the *application DSL* and the *block DSL*. The application DSL is used to connect blocks together and map them to resources. The block DSL is used to declare and (optionally) implement the blocks.

Because the streaming application is described in a DSL embedded in Scala, Scala language constructs can be used to generate potentially large and complex application topologies and resource mappings. The application DSL thus addresses problems 1 and 2 with Auto-Pipe and the X language.

In addition to authoring the application topology and resource mapping, ScalaPipe allows blocks to be implemented completely within the ScalaPipe block DSL. This allows one to use the same block code to operate on multiple data types and run on multiple resources. For example, in ScalaPipe it is possible to create a polymorphic adder block that adds together two values of any type. This block can then be used to generate code for both general-purpose processors and FPGA devices. For general-purpose processors, ScalaPipe generates C code that adheres to the Auto-Pipe software block interface. Likewise, for FPGA devices, ScalaPipe generates Verilog code that adheres to the Auto-Pipe hardware block interface. The block DSL thus addresses problems 3 and 4.

To address problem 5 with Auto-Pipe and X, the ScalaPipe compiler can be enhanced to combine multiple blocks that are connected together on the same resource. Although this feature has yet to be implemented, such an optimization would be possible to implement in the ScalaPipe compiler without requiring application or block code changes. In a similar vein, any improvements to the block code generation (for example, optimizations) would be available for all ScalaPipe applications without requiring application or block code changes.

The ease of creating new blocks and versatile resource mapping abilities of ScalaPipe allow the straightforward evaluation of alternative resource mappings. Any bottlenecks in the application can be tracked down using a performance monitoring system such as TimeTrial [18], which is integrated into ScalaPipe. These bottlenecks can then be addressed by a custom block implementation or by changing the application topology and resource mapping.

III. Block DSL

The ScalaPipe block domain-specific language provides a simple imperative programming language that can be used to implement blocks. To use the block DSL, one extends the `AutoPipeBlock` class. Within the block DSL, the inputs and outputs for the block are specified as well as the implementation. Note that it is possible to use existing blocks in target-specific languages such as Verilog. To use such blocks, an `external` statement is used to inform ScalaPipe of the implementation code. It is also possible to mix multiple external implementations for various platforms as well as an internal implementation (to be used if no matching external implementation is available for the desired target).

A simple add block is shown in Figure 1. This block takes two inputs, adds them together, and sends an output. For this block, we have specified that we already have a custom implementation available for FPGA devices. Note that the

```
val Add = new AutoPipeBlock {
   val in0 = input(SIGNED32)
   val in1 = input(SIGNED32)
   val out = output(SIGNED32)

   external("FPGA", "AddS32")

   out = in0 + in1
}
```

Fig. 1. Add block.

first parameter to `external` is the platform and the second parameter is the name of a directory containing the block implementation.

A. Language Features

The block DSL features many of the standard language constructs available in a traditional programming language for conditionals and looping. By using a version of the Scala compiler with language virtualization features [3], ScalaPipe is able to use the standard Scala control structures such as `if` and `while`. Because of this, a lot of Scala code can be easily reused in ScalaPipe with few changes.

In addition to the basic control structures and math operators, ScalaPipe provides a rich set of data types which includes primitive types such as integer, floating point, and fixed point types, fixed-length arrays, and structures. Blocks that use only these features can run on any resource that ScalaPipe supports. To allow more flexibility and the ability to interface with library code, ScalaPipe also allows function calls and pointer types to be used. However, blocks that use function calls and pointers can only be mapped to traditional processors.

B. Intermediate Representation

Before block code can be generated, it is necessary to turn the block DSL program into an intermediate representation. Since the block DSL is embedded in Scala, the issue of parsing is handled automatically. The DSL code turns into function calls where the variables in the DSL are actually objects to represent the variables, which are created in the `input`, `output`, and `local` functions. Because variables in the block DSL are objects, Scala code and block DSL code can be mixed allowing Scala to act as a macro language. This is similar to *lightweight modular staging* introduced in [25] where variable types determine whether an expression is executed when the application generator runs or if the expression is compiled into code to be executed later.

Since the Scala compiler cannot catch all possible errors that could arise in the block code, an error reporting mechanism is in place which captures the file name and line number so that this information can be reported if ScalaPipe detects an error with the block. To accomplish this, every statement in the block DSL throws and catches an exception to determine the current file name and line number. This allows ScalaPipe to provide meaningful error messages.

Once the abstract syntax tree is built from the block code, it is either used directly for code generation or converted to a control flow graph. When generating code for traditional processors or graphics processors, the abstract syntax tree is used directly for code generation. However, for FPGAs, the abstract syntax tree is first converted to a control flow graph to enable better Verilog code generation.

C. Code Generation

For traditional processors, C code is emitted. For graphics processors, OpenCL C [27] code is generated. Finally, for FPGAs, Verilog is generated. Since the block language maps easily into the C programming language, the abstract syntax tree is used directly for generating code targeted for traditional processors. Unfortunately, generating code for graphics processors and FPGAs requires more work.

For graphics processors, it is desirable to allow multiple threads to run the same kernel on different data elements. To allow this, ScalaPipe checks the block to see if there is state that needs to be preserved across invocations. If there is no state to be preserved, then each element in the input queues can be processed in parallel. In this case, ScalaPipe will generate code to process each item in the input buffer in a separate thread. Note that this simple method for extracting parallelism leaves much to be desired. However, it provides a prototype for evaluating alternative resource mappings. If a higher performance implementation is sought, it is possible to substitute a custom implementation.

Like graphics processors, FPGAs present a problem for automatic code generation from an imperative-style language. To generate Verilog, the abstract syntax tree is first converted into a control flow graph. Each vertex in the control flow graph is then converted into a state in a state machine and the edges become state transitions. As is the case with the graphics processor code generation, this often results in a suboptimal implementation. As previously mentioned, we can get around this by a custom implementation if necessary. However, if the implementation typically does not require many resources, it may be possible to replicate the block many times.

IV. Application DSL

The ScalaPipe application domain-specific language is used to connect blocks together and map them to resources. To use the application DSL, one extends the `AutoPipeApp` class.

A. Overview

The application DSL allows one to specify how blocks are connected using functional composition. Each block takes a list of streams for input and returns a list of streams, which can then be passed to other blocks. To allow large and complex topologies to be generated, Scala can be used as a type-safe macro language.

In addition to the application topology, the application DSL is where resource mapping is described. The `map` function is used for this purpose. Given a stream or edge specification as a parameter, the `map` statement marks where data flows from one resource to another.

B. Resource Mapping

The application DSL creates a graph of blocks connected by streams. Some streams may change resources as indicated by a `map` statement. To map the blocks onto resources, ScalaPipe first assumes all blocks are unassigned and then processes the streams one-by-one until either all blocks are assigned a resource or no more changes occur. If any resources remain unmapped at this point, they are assumed to reside on a traditional processor. Note that it is possible for an invalid mapping to be specified. In this case, the error is reported when the ScalaPipe application generator is run.

Once the resource mapping is complete, ScalaPipe generates the blocks for the required resources and then creates an application with the queues and threads necessary to run the blocks. This application also includes any code necessary for routing data to other devices such as graphics processors or FPGAs. For graphics processors, ScalaPipe generates code to use OpenCL [27] for communication and compiling of the block code. For FPGA devices, code to communicate with the driver for the FPGA device is generated on the software side. On the hardware side, a top level file is generated to connect the blocks on the FPGA device.

V. Solution to Laplace's Equation

As an example to demonstrate ScalaPipe, we present an application to compute the solution to Laplace's equation. Laplace's equation is a second-order partial differential equation (PDE) [28] with several uses, including modeling stationary diffusion (such as heat) and Brownian motion. For heat, given the temperature at the boundaries of an object, solutions to Laplace's equation provide the interior temperatures at equilibrium.

There are multiple ways of solving Laplace's equation. An analytic solution exists for simple boundary conditions, but for many boundary conditions, no analytic solution exists and a numeric solution must be sought [10]. The algorithm we present in this paper is based on a Monte Carlo technique. Although provably correct [24], the algorithm converges slowly. Nevertheless, this method is useful in situations where the solution to only a few interior points is needed.

Our application for solving Laplace's equation is comprised of a `Random` block which computes random numbers using the Mersenne twister algorithm [21], a `Walk` block which performs random walks in the area of interest, and a `Print` block to output the results. More blocks are required in the complete implementation to configure the boundaries and area of interest. For simplicity, these blocks are omitted in this discussion.

One way to increase the parallelism of this application is to distribute the n random walks among w concurrently operating `Walk` blocks. To support this, we introduce two additional block types: a `Split` block to divide its input among two outputs and an `Average` block to average the results from two inputs. By creating a tree of `Split` and `Average` blocks we can vary w to be any power of two. An example with $w = 2$ is shown in Figure 2.

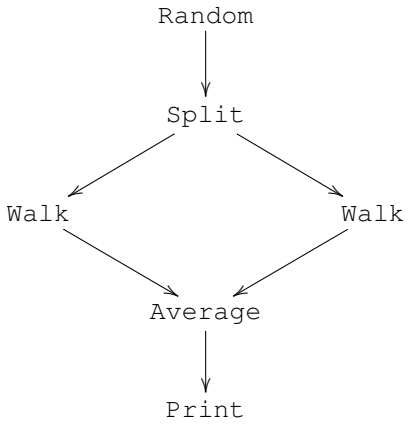

Fig. 2. Distributing the work among two `Walk` blocks.

A. X Implementation

Before showing the ScalaPipe implementation, we first present an implementation of the Laplace application in AutoPipe's X language. To implement the application using X, we must first declare the blocks as shown in Figure 3. Note that this declaration is analogous to a header file and does not provide implementations for the blocks. The implementations are developed separately for each kind of resource on which the block can be deployed. Figure 4 shows how a C implementation of the blocks would be referenced in the X environment, with the `base` specifying the base filename for the C source code. Given this information, these blocks could be used on a traditional processor, but to use the blocks on an FPGA device we must develop an implementation for that device and we must modify Figure 4 accordingly.

Given these block definitions, we next specify the topology of the application as shown in Figure 5. Here we first declare block instances, such as the two `Walk` blocks `walk1` and `walk2`. Next, the connections between the blocks are specified. The edges carry labels so they can be referenced in the resource mapping we describe below. Although the connections between `Split`, `Walk`, and `Average` blocks have structure, the connection specification requires that each edge be explicitly specified. While this may be suitable for small topologies, it becomes cumbersome for large topologies. Further, modifying the number of `Walk` blocks from $w = 2$ to $w = 4$ requires changing most of the code.

After specifying the topology, we declare the available resources and the mapping of blocks to resources as shown in Figure 6. For illustrative purposes, we present a resource mapping where the `Random` and `Print` blocks are each mapped to a separate processor and all other blocks are mapped to the FPGA device. Because edges and block instances must be explicitly mapped, this approach can become unwieldy and error-prone as the number of blocks and complexity of the mapping increases.

```
block Random {
   output UNSIGNED32 y;
};

block Split {
   input  UNSIGNED32 x0;
   output UNSIGNED32 y0;
   output UNSIGNED32 y1;
};

block Walk {
   input  UNSIGNED32 x0;
   output UNSIGNED32 y0;
};

block Average {
   input  UNSIGNED32 x0;
   input  UNSIGNED32 x1;
   output UNSIGNED32 y0;
};

block Print {
   input  UNSIGNED32 x0;
};
```

Fig. 3. X block declarations.

```
platform C_x86 {
   impl Random(base="Random");
   impl Split(base="Split");
   impl Walk(base="Walk");
   impl Average(base="Average");
   impl Print(base="Print");
};
```

Fig. 4. X block implementation references.

```
block top {
      Random   rand;
      Split    split;
      Walk     walk1;
      Walk     walk2;
      Average  avg;
      Print    print;

e1:   rand     -> split;
e2:   split.y0 -> walk1;
e3:   split.y1 -> walk2;
e4:   walk1    -> avg.x0;
e5:   walk2    -> avg.x1;
e6:   avg      -> print;
};
```

Fig. 5. Topology of the Laplace application in X.

```
resource proc[2] is C_x86 {
   (file="proc_1_.cpp", cpunum=0),
   (file="proc_2_.cpp", cpunum=1)
};
resource fpga is HDL;
resource interconnect is DMA;

map fpga = { app };
map proc[1] = { app.rand };
map proc[2] = { app.print };
map interconnect = { app.e1, app.e2 };
```

Fig. 6. X resource mapping.

B. ScalaPipe Blocks

The blocks used in the ScalaPipe version of the Laplace application provide the same functionality as those in the X version. However, in the ScalaPipe version, the blocks are implemented completely in ScalaPipe using the block DSL. To save space, we show code only for the `Random` and `Average` blocks in Figures 7 and 8.

The `Random` block shows the ease with which blocks can be implemented or prototyped in ScalaPipe. This block declares an output parameter, *out*, and several local state parameters. The body of the `Random` block demonstrates several features of the ScalaPipe block DSL including conditionals and loops. When run, the `Random` block will initialize its internal state and then output random values until the program is terminated.

The code within a block can be thought of as executing in a continuous loop. Each time an input port is referenced, a new value is expected. If no input is available, the block will wait for a value. Likewise, each time an output port is assigned, a new value is enqueued for the consumer block (execution will block if the queue is full). Blocks that require no inputs, such as our `Random` block, run continuously until they execute a `stop` statement or the program terminates.

The `Average` block in Figure 8 is an example of a polymorphic block capable of working with multiple data types. The parameter to the constructor of this block, *t*, is of type `AutoPipeType`, which is the base class for all data types in ScalaPipe. The `Average` block can thus operate on any valid ScalaPipe type for which the + and / operators are defined. The `Average` block reads its inputs (blocking if necessary), averages them, and then outputs the averaged values. The process then repeats as long as the block has input.

To create an instance of the `Average` block that can operate on 32-bit unsigned integers, we simply specify:

```
val AverageU32 = new Average(UNSIGNED32)
```

Here we only give a glimpse of the capabilities of the ScalaPipe block DSL. The block DSL supports many more features, including a full array of operators and data types as well as support for calling external functions. Provided external functions are not used, all of the supported features of

```
val Random = new AutoPipeBlock {

   val out = output(UNSIGNED32)

   val mt =
      local(new AutoPipeArray(UNSIGNED32,
            624))
   val index = local(UNSIGNED32, 0)
   val configured = local(BOOL, 0)
   val i = local(UNSIGNED32, randSeed)
   val j = local(UNSIGNED32)
   val y = local(UNSIGNED32)

   if (configured) {
      if (index == 624) {
         i = 0
         while (i < 624) {
            j = i + 1
            if (j == 624) {
               j = 0
            }
            y = mt(i) >> 31
            y += mt(j) & 0x7FFFFFFF
            j = i + 397
            if (j > 623) {
               j -= 624
            }
            mt(i) = mt(j) ^ (y >> 1)
            if (y & 1) {
               mt(i) ^= 0x9908b0df
            }
            i += 1
         }
         index = 0
      }
      y = mt(index)
      y ^= (y >> 11)
      y ^= ((y << 7) & 0x9d2c5680)
      y ^= ((y << 15) & 0xefc60000)
      y ^= (y >> 18)
      index += 1
      out = y
   } else {
      mt(index) = i
      i = 0x6c078965 * (i ^ (i >> 30))
      i += index
      index += 1
      if (index == 624) {
         configured = 1
      }
   }
}
```

Fig. 7. Random block.

```
class Average(t: AutoPipeType)
      extends AutoPipeBlock {

   val in0 = input(t)
   val in1 = input(t)
   val out = output(t)

   out = (in0 + in1) / 2
}
```

Fig. 8. Average block.

```
val Laplace = new AutoPipeApp {

   val random = Random()
   val splits = random.iteratedMap(
                  levels, SplitU32)

   val walks =
      Array.tabulate(1 << levels) {
         x => Walk(splits(x))()
      }

   val result = iteratedFold(walks,
                  AverageU32)

   Print(result)
}
```

Fig. 9. Laplace application.

the block DSL can be compiled to C, OpenCL C, or Verilog as required.

C. ScalaPipe Application Topology

The code for creating the topology for the Laplace application is shown in Figure 9. Here we use the blocks we previously defined. Each time we use function application on a block (for example, `Random()`), a new instance of that block is created. The inputs to the block are arguments to the function application and the outputs are the return value.

There are two Scala functions of interest in this example: `iteratedMap` and `iteratedFold`. By using these functions, we can create trees of `Split` and `Average` blocks of arbitrary depth. Therefore, rather than explicitly creating the blocks and linking them together as in X, we are generating the application topology. The `iteratedMap` and `iteratedFold` functions are Scala functions that were implemented for ScalaPipe to make generating such topologies easier.

In addition to the functions that ScalaPipe provides for generating topologies, it is also possible to create custom functions. As a simple example, consider a block that increments an integer. We could use this block to add five to an integer by creating a pipeline of five `Increment` blocks as shown in Figure 10. However, we note that doing this manually not

```
val plusOne   = Increment(source)
val plusTwo   = Increment(plusOne)
val plusThree = Increment(plusTwo)
val plusFour  = Increment(plusThree)
val result    = Increment(plusFour)
```

Fig. 10. Increment pipeline.

```
def pipeline(s: Stream,
             b: AutoPipeBlock,
             n: Int): Stream = {
   if (n > 0) {
      pipeline(b(s), b, n - 1)
   } else {
      s
   }
}
```

Fig. 11. Function for creating pipelines.

only requires a lot of code, but it also prevents us from easily switching to a different pipeline length.

To allow us to create such pipelines of arbitrary length, we can write a function to do it for us. A function to create a pipeline of length n is shown in Figure 11. This function takes a stream, s, as input and a block, b. It then creates a pipeline n long. Figure 12 shows the updated program to take advantage of this function.

Not only is it possible to use functions specific to ScalaPipe programs for topology or block generation, but we can also use functions from the Scala standard library. For example, in Figure 9, we use the `Array.tabulate` function, which creates an array of the specified length by executing a function for each element. In this case, we use `Array.tabulate` to create the `Walk` block instances.

D. ScalaPipe Resource Mapping

ScalaPipe takes an aspect-oriented [16], [17] approach to resource mapping. That is, the resource boundaries are specified as an edge between two blocks and the type of resource movement is indicated. In Figure 13, the first `map` statement states that all edges out of a `Random` block move from a traditional processor to the first FPGA device. Note that the special block `ANY_BLOCK` is used to match any block. The second statement states that all edges going to the `Print` block move from an FPGA to the second processor (CPU number 1). For comparison purposes, the X resource mapping equivalent to the ScalaPipe resource mapping in Figure 13 is shown in Figure 6.

```
val result = pipeline(source,
                      Increment, 5)
```

Fig. 12. Updated increment pipeline.

```
map(Random -> ANY_BLOCK, CPU2FPGA())
map(ANY_BLOCK -> Print, FPGA2CPU(id = 1))
```

Fig. 13. ScalaPipe resource mapping.

In addition to the resource mapping shown here, other types of mappings are possible. For example, by specifying -1 as the CPU ID, for each edge that the resource mapping matches, a new, unused processor will be used. Also, to allow for cases where an aspect-oriented approach is awkward, it is possible to specify the output of a block instead of an edge (for example, `result` in Figure 9).

E. Evaluation

With the application implemented completely in ScalaPipe, we can now evaluate its performance with various numbers of `Walk` blocks on different resources. Our tests are performed on a Linux system with two six-core AMD processors running at 2.6 GHz. This system also includes two custom FPGA boards connected to the PCI-X bus. Each FPGA board contains two Xilinx Virtex-4 FPGAs.

We first run the simple version of the Laplace application with one `Walk` block on a single processor core. The program takes 101 seconds to generate the output shown in Figure 14. At this point, there are many possible ways to exploit the parallel nature of the problem. However, due to the availability of an FPGA device, we will illustrate its use. In ScalaPipe, we could just as easily use multiple traditional processors or even multiple machines in a cluster. In practice, the decision of whether to use an FPGA or a traditional processor would likely be guided by performance and how other parts of the application are mapped. Nevertheless, because ScalaPipe generates the code for the blocks, moving the processing to an FPGA device requires little effort. To move the processing to the FPGA device, we insert the following `map` statement into the application:

```
map(ANY_BLOCK -> Print, FPGA2CPU())
```

Running the application again, we discover that, unfortunately, we have increased the running time to 579 seconds. This is not too surprising since we have yet to modify the application to take advantage of the FPGA device. Note that the HDL code that ScalaPipe generates is functionally correct, but ScalaPipe currently makes no attempt to optimize it. Instead, ScalaPipe creates a state machine to execute the block code sequentially.

There are many things we could attempt to speed up the program, but we do not know where the bottleneck is. Therefore, we use TimeTrial [18] which is integrated into ScalaPipe. TimeTrial instruments the queues between blocks of a streaming application to report information such as queue occupancy, data rate, and backpressure. Using TimeTrial in ScalaPipe works much like resource mappings: an edge is specified along with the desired measurement. The following statement is used to instrument the edge between the `Random` block and the `Walk` block:

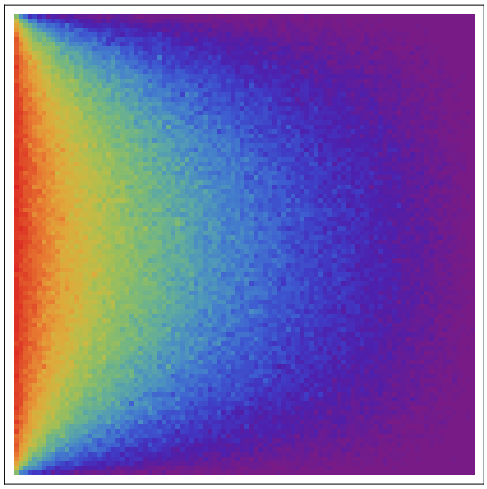

Fig. 14. Output of the Laplace program.

```
measure(Random -> ANY_BLOCK,
        'backpressure)
```

Running the program again, we see that there is backpressure 90% of the time. This indicates that the `Walk` block is the bottleneck. One way to speed up the `Walk` block is to divide the load among multiple `Walk` blocks. As coded, this can be accomplished by changing the *levels* parameter shown in Figure 9. We arbitrarily select 4, which gives us $2^4 = 16$ `Walk` blocks. We leave the TimeTrial measurement in the program, which will now report the backpressure between the `Random` block and the first `Split` block.

After modifying the program to use 16 `Walk` blocks, the execution time of the program drops from 579 seconds to 102 seconds. This is a speedup of about 5.7x over the previous run on the FPGA device and about on par with the implementation on a traditional processor. Although we are heading in the right direction, we might have hoped for a 16x speedup because we are now using 16 `Walk` blocks. Looking at the TimeTrial measurement of the backpressure between the `Random` block and the first `Split` block reveals that the edge has backpressure only 3% of the time. This indicates that the `Random` block is now the bottleneck.

Up until this point, we have implemented the entire application using only ScalaPipe. That is, there is no custom HDL in use. However, because we now know that the `Random` block is the bottleneck, we suspect a custom implementation of that block may provide the desired performance. Therefore, we point ScalaPipe to a custom implementation in Verilog (described in [26]) using an `external` statement. Using the custom block implementation, the application now takes 33 seconds, which is more than the 16x speedup we were seeking by moving to 16 `Walk` blocks and 3.1x faster than the original program on a single processor.

Now the `Walk` blocks are again the bottleneck. To improve the performance even further, we could add more `Walk` blocks or use a custom `Walk` block implementation. There is ample logic space on the FPGA device to accommodate more `Walk` blocks, but the block RAM resources are a limiting factor. This is because there is a buffer for every edge between blocks. The fact that the `Split` blocks are arranged in a tree means that there are many buffers that probably are not needed. We could address this manually by creating a split block with more than 2 outputs. Indeed, we could even use Scala to generate the block with the required number of outputs. However, assuming that we have met our performance goal, we stop here.

F. Discussion

By using ScalaPipe, we easily developed this application and isolated bottlenecks in its implementation. After isolating the bottlenecks, ScalaPipe allowed us to change the topology and focus only on improving those blocks that actually mattered to obtain our performance goal. In addition, because most of the blocks used in this application take a parameter to specify their type, the whole application could be modified to work with 64-bit values instead of 32-bit values with very few changes. In X, we would have to develop separate implementations and extensively modify the descriptions of the blocks.

An advantage to ScalaPipe that is not made explicit in the Laplace application is the code savings one obtains by using ScalaPipe to generate blocks instead of writing the blocks by hand. In ScalaPipe, one need not worry about the boiler-plate code that is necessary for Auto-Pipe blocks implemented in C or an HDL. This combined with the fact that the same block in ScalaPipe can be used to generate code for both traditional processors and FPGAs as well as for multiple data types means that the code savings can be substantial. Further, by having only a single block implementation, any changes to that block need to only happen in one place.

Although the code that ScalaPipe generates for blocks mapped to FPGAs is not optimized, the fact that we are able to easily explore the use of FPGAs is useful. We are able to move large parts of the application to an FPGA without writing any HDL code and then concentrate on only those blocks which prevent us from meeting our performance goals. In the future, it is conceivable that an improved Verilog code generator or the incorporation of an existing HDL code generation tool into ScalaPipe would make custom HDL code unnecessary in even more cases.

As we have shown, ScalaPipe is capable of generating multiple forms of code. From the application DSL, ScalaPipe generates the necessary wrappers to tie the blocks together and communicate with the various resources. From the block DSL, ScalaPipe can generate C code for blocks mapped to traditional processors, OpenCL C code for blocks mapped to graphics processors, and Verilog code for blocks mapped to FPGA devices.

Since the Scala language is used to generate the application topology, for a particular program, the amount of X code required for an equivalent program in Auto-Pipe is unbounded. For example, an Auto-Pipe version of the Laplace application requires 69 lines of X code when two walk blocks are used. With 64 walk blocks this number increases to 503 lines. Both

of these numbers are greater than the core of the ScalaPipe code used to describe the application, which is only about 10 lines. Note that the X code used for comparison does not contain extra white space or comments.

As far as block code generation is concerned, in addition to allowing code reuse, there is a significant amount of code savings for using internal blocks. The savings comes primarily from the fact that the boiler-plate code required for blocks and the interactions with the Auto-Pipe block interface are implicit in the internal block language. It should be noted that the savings can become less significant as the complexity of the block increases, however, we can still obtain a benefit from having a single, polymorphic implementation for multiple target platforms. As an example of the code savings, for the `Average` block shown in Figure 8, the ScalaPipe description is 9 lines (1 line for the actual implementation). The generated C code, on the other hand, is 74 lines and the generated Verilog code is 88 lines. Although a custom implementation of the `Average` block may be slightly shorter, an order of magnitude gain for such a simple block seems unlikely.

VI. Related Work

There is a vast body of related work spanning everything from streaming programming systems to high-level synthesis tools.

StreamIt [29] is a language with a Java-like syntax for creating streaming applications. There are a few notable differences between StreamIt and ScalaPipe or Auto-Pipe. First, StreamIt generates applications which run on traditional processors whereas Auto-Pipe is designed for heterogeneous platforms. Further, StreamIt puts restrictions on the application topology using pre-defined fork and join mechanisms. Like ScalaPipe, StreamIt computation kernels and coordination are specified in the same language. However, unlike ScalaPipe, StreamIt is a completely new language whereas ScalaPipe is a library built on top of Scala.

Optimus [15] is tool for compiling a streaming application into a hardware implementation. Optimus focuses more on the way streaming applications are mapped into hardware than on the source language, for which it uses StreamIt [29]. Unlike ScalaPipe or Auto-Pipe, Optimus targets only hardware rather than heterogeneous platforms.

Spidle [8] is a DSL for streaming applications. Spidle has goals similar to StreamIt. The use of a DSL to generate C code is similar to ScalaPipe, but Spidle lacks support for heterogeneous platforms.

Streams-C [12] is in many ways like ScalaPipe. Like ScalaPipe, Streams-C allows one to construct streaming applications by writing code in a high-level language that can compile to either traditional processors or FPGAs. However, unlike ScalaPipe, Streams-C uses comments embedded in the code for the application topology and resource mapping, whereas in ScalaPipe, the mapping and topology is generated using a DSL. Thus, the ScalaPipe approach has the advantage of allowing programmatic resource mapping and topology generation.

Merge [20] is a framework for heterogeneous multi-core systems. Merge provides a library-based approach for using heterogeneous systems based on MapReduce [9]. The Merge framework targets traditional processors and graphics processors. Although one can think of ScalaPipe as a library or framework, ScalaPipe uses this library to actually generate code for the target architecture.

Although Chafi et al. outline the use of DSLs embedded in Scala as an approach to utilizing heterogeneous systems [4], they focus on the use of a system called Delite which serves as an execution engine for DSLs. ScalaPipe, on the other hand, generates a complete application that can run outside of the Scala environment. Further, ScalaPipe uses two distinct DSLs: one to provide a general-purpose language for authoring blocks and a second for linking blocks together and assigning them to resources.

Lime [1] is a Java-compatible programming language that targets heterogeneous architectures including traditional processors and FPGAs. Lime allows the expression of many forms of parallelism rather than enforcing a particular model such as streaming, which is done in ScalaPipe. The Lime run-time system then attempts to dynamically partition the program onto the available resources.

Kiwi [13] is a system for converting C# programs into HDL. Kiwi uses existing concurrency and synchronization constructs to determine the parallelism in programs. Like ScalaPipe, the approach taken by Kiwi allows the same code to be used for both traditional processors and FPGAs. However, whereas Kiwi attempts to exploit parallelism from concurrency and synchronization primitives, ScalaPipe exploits parallelism from the streaming decomposition of the program.

The Open Component Portability Infrastructure (OpenCPI) [23] is a framework to simplify programming for heterogeneous processing environments including traditional processors, graphics processors, FPGAs, and digital signal processors. OpenCPI provides a component based architecture, where each component is reminiscent of a block in ScalaPipe. These blocks are specified in a language, or *authoring model*, which is appropriate for their target architecture. These components are then composed using either an application control interface or an XML specification. Unlike ScalaPipe, however, OpenCPI does not provide a way to specify components in a unified language for deployment to multiple targets.

VII. Conclusion and Future Work

In this paper we presented ScalaPipe, which is a collection of domain-specific languages embedded in Scala for generating streaming applications on heterogeneous platforms. ScalaPipe allows large complex topologies and resource mappings to be generated and modified easily without substantial changes to the application source code. Further, ScalaPipe allows blocks to be implemented either directly in ScalaPipe or externally. By implementing a block directly in ScalaPipe, the same source code can be used for different data types. Because ScalaPipe is capable of compiling blocks to C, OpenCL C,

and Verilog, the same source code can also target multiple platforms as needed.

An example application for solving Laplace's equation was used to demonstrate the ease with which such an application can be developed and modified. As shown, the combination of ScalaPipe with TimeTrial allows one to evaluate alternative resource mappings and topologies easily without wasting time on custom block implementations that may provide little benefit. Thus, through the use of TimeTrial statements in ScalaPipe, we were able to go from a simple traditional processor implementation to an FPGA implementation with better performance using only a single custom block implementation.

Future work will consider more optimizations for the generated C and Verilog code. Although the C code generation is fairly straightforward, the C blocks could be optimized to process multiple data items at a time. Also, because ScalaPipe produces the block code for internal blocks, ScalaPipe could combine adjacent internal blocks that target the same resource to avoid the overhead of the queues between blocks and expose any cross-block optimizations that may be possible. As far as the Verilog code generation is concerned, there is ample opportunity for more efficient code generation. This could be accomplished either by improving ScalaPipe itself or by leveraging existing tools such as DWARV [31], LegUp [2], or ROCCC [30].

REFERENCES

[1] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: a Java-compatible and synthesizable language for heterogeneous architectures," in *Proc. of ACM Int'l Conf. on Object Oriented Programming Systems, Languages, and Applications*, 2010, pp. 89–108.

[2] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proc. of 19th ACM/SIGDA Int'l Symp. on Field Programmable Gate Arrays*, Feb. 2011, pp. 33–36.

[3] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun, "Language virtualization for heterogeneous parallel computing," in *Proc. of ACM Int'l Conf. on Object Oriented Programming Systems, Languages, and Applications*, 2010, pp. 835–847.

[4] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun, "A domain-specific approach to heterogeneous parallelism," in *Proc. of 16th ACM Symp. on Principles and Practice of Parallel Programming*, 2011, pp. 35–46.

[5] R. D. Chamberlain, J. Buhler, M. A. Franklin, and J. H. Buckley, "Application-guided tool development for architecturally diverse computation," in *Proc. of ACM Symp. on Applied Computing*, Mar. 2010, pp. 496–501.

[6] R. D. Chamberlain, M. A. Franklin, E. J. Tyson, J. H. Buckley, J. Buhler, G. Galloway, S. Gayen, M. Hall, E. B. Shands, and N. Singla, "Auto-Pipe: Streaming applications on architecturally diverse systems," *Computer*, vol. 43, no. 3, pp. 42–49, Mar. 2010.

[7] R. D. Chamberlain, J. M. Lancaster, and R. K. Cytron, "Visions for application development on hybrid computing systems," *Parallel Computing*, vol. 34, no. 4-5, pp. 201–216, May 2008.

[8] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, and C. Pu, "Spidle: a DSL approach to specifying streaming applications," in *Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering*, 2003, pp. 1–17.

[9] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[10] S. J. Farlow, *Partial Differential Equations for Scientists and Engineers*. Dover Publications, 1993.

[11] M. A. Franklin, E. J. Tyson, J. Buckley, P. Crowley, and J. Maschmeyer, "Auto-Pipe and the X language: A pipeline design tool and description language," in *Proc. of Int'l Parallel and Distributed Processing Symp.*, Apr. 2006.

[12] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language," in *Proc. of IEEE Symp. on Field-Programmable Custom Computing Machines*, Apr. 2000, pp. 49–56.

[13] D. Greaves and S. Singh, "Kiwi: Synthesis of FPGA circuits from parallel programs," in *Proc. of 16th Int'l Symp. on Field-Programmable Custom Computing Machines*, 2008, pp. 3–12.

[14] C. Grelck, S.-B. Scholz, and A. Shafarenko, "A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components." *Parallel Processing Letters*, vol. 18, no. 2, pp. 221–237, 2008.

[15] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah, "Optimus: efficient realization of streaming applications on FPGAs," in *Proc. of Int'l Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, 2008, pp. 41–50.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An overview of AspectJ," in *ECOOP 2001 – Object-Oriented Programming*, 2001, vol. 2072, pp. 327–354.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97 – Object-Oriented Programming*, 1997, vol. 1241, pp. 220–242.

[18] J. M. Lancaster, J. G. Wingbermuehle, and R. D. Chamberlain, "Asking for performance: Exploiting developer intuition to guide instrumentation with TimeTrial," in *Proc. of IEEE 13th Int'l Conf. on High Performance Computing and Communcations*, Sep. 2011, pp. 321–330.

[19] P. Li, K. Agrawal, J. Buhler, and R. D. Chamberlain, "Deadlock avoidance for streaming computations with filtering," in *Proc. of 22nd ACM Symp. on Parallelism in Algorithms and Architectures*, Jun. 2010, pp. 243–252.

[20] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," *SIGPLAN Not.*, vol. 43, pp. 287–296, March 2008.

[21] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, pp. 3–30, Jan. 1998.

[22] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the Scala programming language," École Polytechnique Fédérale de Lausanne, Tech. Rep. IC/2004/64, 2004.

[23] "OpenCPI - open componenet portability infrastructure," http://www.opencpi.org.

[24] J. F. Reynolds, "A proof of the random-walk method for solving Laplace's equation in 2-D," *The Mathematical Gazette*, vol. 49, no. 370, pp. 416–420, Dec. 1965.

[25] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs," in *Proc. of 9th Int'l Conf. on Generative Programming and Component Engineering*, 2010, pp. 127–136.

[26] N. Singla, M. Hall, B. Shands, and R. D. Chamberlain, "Financial Monte Carlo simulation on architecturally diverse systems," in *Proc. of Workshop on High Performance Computational Finance*, Nov. 2008.

[27] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science and Engineering*, vol. 12, pp. 66–73, 2010.

[28] W. A. Strauss, *Partial Differential Equations: An Introduction*. Wiley, 1992.

[29] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Proc. of 11th Int'l Conf. on Compiler Construction*, 2002, pp. 179–196.

[30] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in C with ROCCC 2.0," in *Proc. of IEEE Symp. on Field-Programmable Custom Computing Machines*, 2010, pp. 127–134.

[31] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, "DWARV: Delftworkbench automated reconfigurable VHDL generator," in *Int'l Conf. on Field Programmable Logic and Applications*, Aug. 2007, pp. 697–701.