ScalaPipe:
A Streaming Application Generator

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

Washington University in St. Louis

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

Computation kernels, or blocks connected by explicit communication channels

Advantages:
- Performance
- Reuse
- Abstraction

Systems:
- Auto-Pipe [Fr06]
- Streams-C [Go00]
- StreamIT [Th02]
Example: Solution to Laplace’s Equation

- PDE with several uses, including stationary heat diffusion
- Solvable using a Monte-Carlo technique
Streaming Implementation

Random → Walk → Print
Parallel Walks

Random → Split → Walk → Average → Print

Walk → Split → Walk

Print → Average → Walk → Split → Random
Auto-Pipe & X

X Description -> X compiler

C Block -> Application

VHDL Block -> Application
Laplace Application in X

```plaintext
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;
}
```

Labels:
- `rand`
- `split`
- `walk1`
- `walk2`
- `avg`
- `print`

Edge Connections:
- `e1`: `rand` to `split`
- `e2`: `split.y0` to `walk1`
- `e3`: `split.y1` to `walk2`
- `e4`: `walk1` to `avg.x0`
- `e5`: `walk2` to `avg.x1`
- `e6`: `avg` to `print`

Block instances:
- Blocks are implemented externally in C or an HDL.
Observation 1

As the number of walk blocks increases, the amount of configuration code increases

Two Walk blocks:

\[
\begin{align*}
e1: & \text{ rand } \rightarrow \text{ split;} \\
e2: & \text{ split.y0 } \rightarrow \text{ walk1;} \\
e3: & \text{ split.y1 } \rightarrow \text{ walk2;} \\
e4: & \text{ walk1 } \rightarrow \text{ avg.x0;} \\
e5: & \text{ walk2 } \rightarrow \text{ avg.x1;} \\
e6: & \text{ avg } \rightarrow \text{ print;}
\end{align*}
\]

128 walk blocks requires 896 lines of X!

Four Walk blocks:

\[
\begin{align*}
e1: & \text{ rand } \rightarrow \text{ split1;} \\
e2: & \text{ split1.y0 } \rightarrow \text{ split2;} \\
e3: & \text{ split1.y1 } \rightarrow \text{ split3;} \\
e4: & \text{ split2.y0 } \rightarrow \text{ walk1;} \\
e5: & \text{ split2.y1 } \rightarrow \text{ walk2;} \\
e6: & \text{ split3.y0 } \rightarrow \text{ walk3;} \\
e7: & \text{ split3.y1 } \rightarrow \text{ walk4;} \\
e8: & \text{ walk1 } \rightarrow \text{ avg1.x0;} \\
e9: & \text{ walk2 } \rightarrow \text{ avg1.x1;} \\
e10: & \text{ walk3 } \rightarrow \text{ avg2.x0;} \\
e11: & \text{ walk4 } \rightarrow \text{ avg2.x1;} \\
e12: & \text{ avg1 } \rightarrow \text{ avg3.x0;} \\
e13: & \text{ avg2 } \rightarrow \text{ avg3.x1;} \\
e14: & \text{ avg3 } \rightarrow \text{ print;}
\end{align*}
\]
Our Approach

Type-safe generator language

```scala
val Laplace = new AutoPipeApp {

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

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

  val result = iteratedFold(walks, AverageU32)
  Print(result)
}
```

Same code can generate 1 Walk block or 128 Walk blocks.
Observation 2

Moving blocks to a new device requires reimplementation
Our Approach

A single language for block implementations

- ScalaPipe Block
- HDL Implementation
- C Implementation
- Others
Observation 3

Changing the data type requires new block implementations

```verilog
module ShiftRightU32(...);
    input wire[31:0] input_x;
    output wire[31:0] output_y;
    ...
    output_y <= input_x >> 1;
    ...
endmodule

module ShiftRightS64(...);
    input wire[63:0] input_x;
    output wire[63:0] output_y;
    ...
    output_y <= input_x >>> 1;
    ...
endmodule
```
Our Solution

Polymorphic block implementations

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

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

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

Same implementation works for integral, fixed point, and floating point types.
Observation 4

The block interface for blocks on the same resource is a bottleneck
Our Approach

Single compiler for both the block language and coordination language.
ScalaPipe

Source code (Scala) → Scala compiler → Generator → Application 1 (e.g. 2 Walks)

ScalaPipe Library

- Coordination DSL
- Block DSL

Application 2 (e.g. 8 Walks)
AverageU32 Block

val AverageU32 extends AutoPipeBlock {
  val in0 = input(UNSIGNED32)
  val in1 = input(UNSIGNED32)
  val out = output(UNSIGNED32)

  out = (in0 + in1) / 2
}

in0

AverageU32

in1

out
Polymorphic Average Block

class Average(t: AutoPipeType) extends AutoPipeBlock {

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

    out = (in0 + in1) / 2
}

val AverageU32 = new Average(UNSIGNED32)

t can be any of the following:
• Signed or unsigned integer of any width
• Fixed point type
• Floating point type
class Repeat(v: Int, count: Int) extends AutoPipeBlock {

    val in  = input(SIGNED32)
    val out = output(SIGNED32)
    val tmp = local(SIGNED32)

    tmp = in
    if (tmp == v) { // Evaluated at run time
        for (i <- 1 to count) { // Expanded at compile time
            out = tmp
        }
    } else {
        out = tmp
    }
}
External AverageU32

- Potentially more efficient
- External and internal blocks can be mixed

```java
val AverageU32 = new AutoPipeBlock {

    val in0 = input(UNSIGNED32)
    val in1 = input(UNSIGNED32)
    val out = output(UNSIGNED32)

    external("HDL", "AverageU32")

    // Optional internal implementation
}
```
Block Code Generation
HDL Code Optimizer

- Common subexpression elimination
- Dead store elimination
- Dead code elimination
- Strength reduction
- Copy propagation
- ASAP scheduling
Coordination DSL

Describes the topology and resource mapping

```scala
val Laplace = new AutoPipeApp {

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

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

    val result = iteratedFold(walks, AverageU32)
    Print(result)
}
```
Generating Pipelines

X language:

```
block pipeline {
    input UNSIGNED32 source;
    output UNSIGNED32 result;

    Inc inc1;
    Inc inc2;
    Inc inc3;
    Inc inc4;

    source -> inc1;
    inc1 -> inc2;
    inc2 -> inc3;
    inc3 -> inc4;
    inc4 -> result;
}
```

ScalaPipe:

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

```
val result = pipeline(source,
    Inc, 4)
```
Aspect-Oriented Resource Mapping

map(Random -> ANY_BLOCK, CPU2FPGA())

map(ANY_BLOCK -> Print, FPGA2CPU())
TimeTrial \cite{La11}

How do we find bottlenecks?

\texttt{measure(ANY\_BLOCK -> Walk, 'backpressure')}

\begin{tikzpicture}
  \node[rectangle, draw] (random) at (0,0) {Random};
  \node[rectangle, draw] (split) at (2,0) {Split};
  \node[rectangle, draw] (average) at (4,0) {Average};
  \node[rectangle, draw] (print) at (6,0) {Print};
  \node[rectangle, draw] (walk) at (2,2) {Walk};

  \draw[->, color=red] (random) -- (split);
  \draw[->, color=blue] (split) -- (average);
  \draw[->, color=blue] (average) -- (print);
  \draw[->, color=red] (walk) -- (split);
  \draw[->, color=blue] (walk) -- (average);

  \node at (1,1) {Walk};
  \node at (3,1) {Walk};

  \begin{axis}[
    title={% Backpressure},
    ylabel={% Backpressure},
    xlabel={Frame},
    xmin=1, xmax=61,
    ymin=0, ymax=100,
    xtick={10,20,30,40,50,60},
    ytick={20,40,60,80,100},
    grid=both,
    legend style={at={(0.5,-0.2)},anchor=north},
  ]
    \addplot[blue, thick, smooth] coordinates {
      (10, 90) (20, 85) (30, 92) (40, 88) (50, 95) (60, 91)
    };
    \addplot[blue, thick, smooth] coordinates {
      (10, 80) (20, 75) (30, 82) (40, 78) (50, 85) (60, 81)
    };
    \addplot[blue, thick, smooth] coordinates {
      (10, 70) (20, 65) (30, 72) (40, 68) (50, 75) (60, 71)
    };
    \addplot[blue, thick, smooth] coordinates {
      (10, 60) (20, 55) (30, 62) (40, 58) (50, 65) (60, 61)
    };
    \addplot[blue, thick, smooth] coordinates {
      (10, 50) (20, 45) (30, 52) (40, 48) (50, 55) (60, 51)
    };
    \addplot[blue, thick, smooth] coordinates {
      (10, 40) (20, 35) (30, 42) (40, 38) (50, 45) (60, 41)
    };
    \addplot[blue, thick, smooth] coordinates {
      (10, 30) (20, 25) (30, 32) (40, 28) (50, 35) (60, 31)
    };
    \addplot[blue, thick, smooth] coordinates {
      (10, 20) (20, 15) (30, 22) (40, 18) (50, 25) (60, 21)
    };
    \addplot[blue, thick, smooth] coordinates {
      (10, 10) (20, 5) (30, 12) (40, 8) (50, 15) (60, 11)
    };
    \addplot[blue, thick, smooth] coordinates {
      (10, 0) (20, 0) (30, 0) (40, 0) (50, 0) (60, 0)
    };
  \end{axis}
\end{tikzpicture}
Illustration of Use

CPU 0

RNG → Walk → Print

Time (s)

101s

CPU
Illustration of Use

- CPU
- FPGA

- 101s
- 174s

- Time (s)

- FPGA 0
  - RNG
  - Walk
  - Print

- CPU 0

- 83% Backpressure

- Illustration of Use
Illustration of Use

- CPU: 101s
- FPGA: 174s
- 16 Walks: 41s

Bar chart showing comparison of CPU and FPGA times for various operations.

Diagram:
- FPGA 0
- Walk
- Print
- CPU 0
- RNG
- Split
- 0% Backpressure
- Walk

Legend:
- Green: FPGA
- Blue: CPU
- Yellow: 16 Walks
Illustration of Use

- `CPU 0`
- `FPGA 0`
- `cRNG` → `Split` → `Walk` → `CPU 0`
- `Print`

Bar chart:
- CPU: 101s
- FPGA: 174s
- 16 Walks: 41s
- Custom RNG: 12s

Time (s)
The Current State of ScalaPipe

• Code generation for CPUs, FPGAs, and GPUs
• FPGA and GPU code generation is suboptimal
• No cross-block optimizations
The Future of ScalaPipe

• Improved code generation
  - Consume multiple items at a time
  - More Verilog and OpenCL C optimizations
• Support for more devices
• Library generation
• Cross-block optimizations
Conclusion

• ScalaPipe is a streaming application generator
• The block DSL allows code reuse across data types and platforms
• The coordination DSL allows easy generation of large and complex topologies
• Keeping everything in the same language exposes optimization opportunities
References


