Motivation

• Synthesizing Hardware from Synchronous Data Flow (SDF) Specifications
  - SDF models provide natural algorithm concurrency
  - SDF models are statically scheduled
  - Many relevant DSP algorithms can be specified in SDF

• Increasing Use of FPGAs for Signal Processing
  - Increasing density of FPGAs (1M gates for ~$20)
  - Exploit hardware parallelism
  - System programmability through reconfiguration

• Goal: Generate FPGA circuits from arbitrary Ptolemy II SDF models
  - Target FPGAs using BYU JHDL Design Tools
  - Synthesize hardware from arbitrary actors
Synthesizing Hardware from SDF

- Many SDF synthesis projects rely on predefined SDF libraries
  - Actor libraries provide hardware implementation
  - One to one mapping between SDF actors and synthesized hardware
- Disadvantages of library approach
  - Hardware options limited by library size
  - Custom actors may require composition of many fine-grain primitives
  - Application-specific libraries often required
  - Parameterized libraries often used

JHDL Hardware Generation

- Goal: synthesize hardware from arbitrary SDF actors defined in software
  - Describe custom hardware actors in software
  - Convenient specification for many operations
  - May coexist with library-based synthesis
- Approach
  - Specify actor behavior in software (Ptolemy II)
  - Specialize actor to model-specific parameters
  - Extract behavior of specialized actor
  - Synthesize corresponding hardware
Example: 3-Tap FIR Filter

- Actor composed of low-level primitives
  - Multipliers, Adders, signal limiter
  - Delay elements, Constants
  - Correspond to hardware elements
- Relatively cumbersome to create
public class SimpleFIR ... {
    ...
    public void fire() {
        int in = input.get(0); // Get token

        int mac = in * c0;
        mac += delay1 * c1;
        mac += delay2 * c2;

        if (mac > MAX) // clip result
            mac = MAX;
        else if (mac < MIN)
            mac = MIN;

        output.send(mac); // Send result

        delay2 = delay1; // update memory
        delay1 = in;
    }
    ...
}
**Hardware Generation Approach**

- Define custom actors in Java
- Create model with custom & existing actors
- Specialize actors and model
- Extract behavior of each actor
  - Disassemble byte-codes of specialized actor class file
  - Generate control-flow/dataflow graph (primitive operations)
  - Generate composite dataflow graph (predicated execution)
  - Extract internal state
- Generate a composite SDF graph (merge actor graphs)
- Perform graph/hardware optimization
- Generate hardware from synthesized SDF
  - Exploit Java-based JHDL Design environment
  - Generate EDIF netlist from JHDL hardware model

---

**Specifying Custom Actor Behavior**

- Custom actors can be created in Ptolemy II
  - See Chapter 5 of the Ptolemy II Design Guide "Designing Actors"
- Behavior defined in three “action” methods
  - prefire() Determines ability of actor to fire
  - fire() Read inputs and create new outputs
  - postfire() Update persistent state
- Hardware synthesis analyzes “action” methods to extract actor behavior
- Actors and model “specialized” using Ptolemy II Java code generator infrastructure
Java Classfile Disassembly

- Actor behavior extracted directly from compiled Java .class file
  - Common, well-supported standard
  - Eliminate need to parse Java source
  - Contains all necessary actor information
  - Tools readily available

- Soot Java Optimizing Framework
  - Developed at McGill University in Montreal

Generate Actor Control Flow Graph

```java
public class SimpleFIR ... {
    ...
    public void fire() {
        int in = input.get(0);
        int mac = in * c0;
        mac += delay1 * c1;
        mac += delay2 * c2;

        if (mac > MAX)
            mac = MAX;
        else if (mac < MIN)
            mac = MIN;
        output.send(mac);
        delay2 = delay1;
        delay1 = in;
    }
    ...
}
```

- Identify basic blocks
- Annotate control dependencies
- Identify intervals
  - One or more basic blocks
  - Single entry point and single exit point
  - May require addition of join nodes (with appropriate conditional)
- Predicated execution graph
public class SimpleFIR ... {
    ...
    public void fire() {
        int in = input.get(0);
        int mac = in * c0;
        mac += delay1 * c1;
        mac += delay2 * c2;
        if (mac > MAX)
            mac = MAX;
        else if (mac < MIN)
            mac = MIN;
        output.send(mac);
        delay2 = delay1;
        delay1 = in;
    }
    ...
}
public class SimpleFIR ... {
    ...
    public void fire() {
        int in = input.get(0);
        int mac = in * c0;
        mac += delay1 * c1;
        mac += delay2 * c2;

        if (mac > MAX)
            mac = MAX;
        else if (mac < MIN)
            mac = MIN;
        output.send(mac);
        delay2 = delay1;
        delay1 = in;
    }
    ...
}
public class SimpleFIR ...
{
    ...
    public void fire() {
        int in = input.get(0);
        int mac = in * c0;
        mac += delay1 * c1;
        mac += delay2 * c2;
        if (mac > MAX)
            mac = MAX;
        else if (mac < MIN)
            mac = MIN;
        output.send(mac);
        delay2 = delay1;
        delay1 = in;
    }
    ...
}

- Generate dataflow graph for each basic block
  - Vertices: Java primitive operations
  - Edges: Data dependencies between operations
  - Some parallelism extracted from sequential byte codes

- Predicated control-flow graph

---

Byte Code

r0 := @this;
load.r r0;
fieldget SimpleFIR.input;
virtualinvoke getInt;
store.i i0;
load.i i0;
push 3;
mul.i;
load.r r0;
fieldget SimpleFIR.delay1;
push 5;
mul.i;
add.i;
load.r r0;
fieldget SimpleFIR.delay2;
push 5;
mul.i;
add.i;
store.i i7;
load.i i7;
push 5;
ifcmple.i label0;

---
**Merge Dataflow Graphs**

- Merge each dataflow graph into a single dataflow graph
  - Insert into predicated execution graph
  - Resolve mutually exclusive variable definitions with select nodes
- Single dataflow graph for actor behavior

**Extract Actor State**

```java
public class SimpleFIR ... {
    ... public void fire() {
        int in = input.get(0);
        int mac = in * c0;
        mac += delay1 * c1;
        mac += delay2 * c2;
        if (mac > MAX)
            mac = MAX;
        else if (mac < MIN)
            mac = MIN;
        output.send(mac);
        delay2 = delay1;
        delay1 = in;
    }
    ...}
```

- State contained in class field variables
  - Read followed by a write
  - Last value written to variable is variable state
- Graph updated to contain sample delay nodes
  - Sample delay node added for state variables
- State should be set in postfire() method
public class SimpleFIR ... {
    ...
    public void fire() {
        int in = input.get(0);
        int mac = in * c0;
        mac += delay1 * c1;
        mac += delay2 * c2;
        if (mac > MAX)
            mac = MAX;
        else if (mac < MIN)
            mac = MIN;
        output.send(mac);
        delay2 = delay1;
        delay1 = in;
    }
    ...
}

• Generate hardware circuit for each Java primitive operation
  - Arithmetic
  - Logical operations
  - Delay elements

• Create circuit in JHDL data structure
  - Circuit simulation & viewing
  - EDIF netlist generation
Limitations

- Currently limited to feed-forward behavior
  - No loops
  - No recursion
  - Limited method inlining
- Hardware types limited
  - Scalar primitive types
    - 32-bit integers (no bit-width analysis)
    - 1-bit Boolean
  - Custom Port/Token object used
- No resource sharing
Conclusions and Future Work

• JHDL hardware generation provides ability to synthesize hardware for arbitrary actors
  - Convenient design specification
  - Reduces reliance on limited actor libraries

• Development ongoing

• Future Work
  - Bit-width analysis & support
  - Support additional standard Ptolemy types
  - Loop unrolling
  - Resources sharing and scheduling