Java Bytecode to Hardware Made Easy with Bluespec SystemVerilog

Flavius Gruian Mehmet Ali Arslan

Lund University, Sweden
{Flavius.Gruian, Mehmet_Ali.Arslan}@cs.lth.se

Java Technologies for Real-time and Embedded Systems, 2012

Outline

1. Introduction
2. From Bytecodes to Hardware
3. Experimental Evaluation
4. Summary & Future Work

Motivation

Stack machines make for nice models but slow implementations, hence bytecode folding, JIT on 3-address machines

Unrolling the stack or part of it, allows for fast data access in hardware

Java processors could use a performance boost (e.g. hardware accelerators)

Bluespec SystemVerilog offers useful abstractions, good tool support, a few success stories

Motivation

Stack machines make for nice models but slow implementations, hence bytecode folding, JIT on 3-address machines

Unrolling the stack or part of it, allows for fast data access in hardware

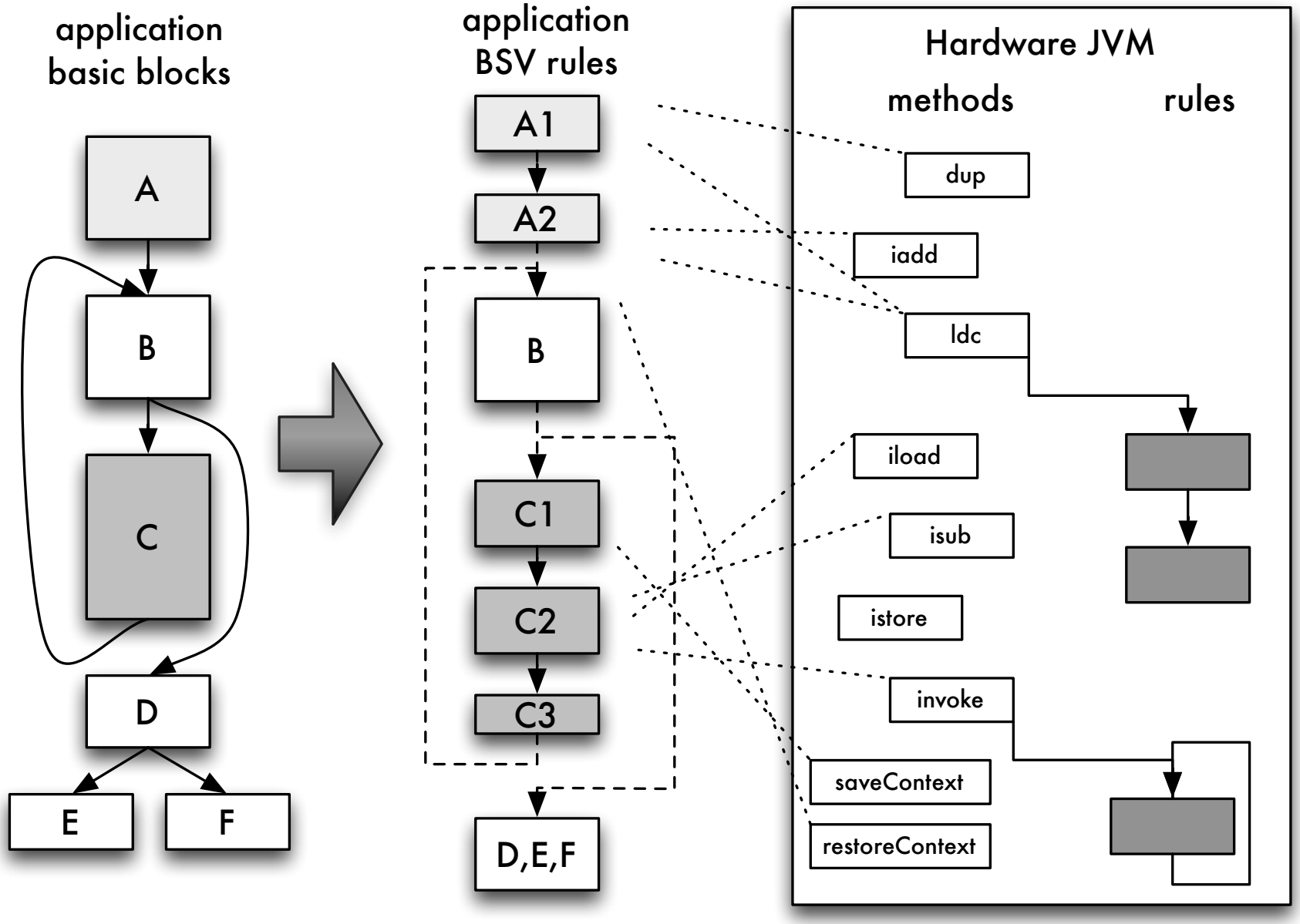
Java processors could use a performance boost (e.g. hardware accelerators)

Bluespec SystemVerilog offers useful abstractions, good tool support, a few success stories

Question

Can we employ BSV and automation to generate accelerators for some of the existing Java processors?

From Java to Hardware, via BSV



Bluespec SystemVerilog

A hardware description language based on SystemVerilog:

typing strong, static type-checking, polymorphism

Bluespec SystemVerilog

A hardware description language based on SystemVerilog:

typing strong, static type-checking, polymorphism

modules as building blocks, encapsulating states and behavior, requiring and implementing interfaces

Bluespec SystemVerilog

A hardware description language based on SystemVerilog:

typing strong, static type-checking, polymorphism

modules as building blocks, encapsulating states and behavior, requiring and implementing interfaces

interfaces described as sets of methods

Bluespec SystemVerilog

A hardware description language based on SystemVerilog:

typing	strong, static type-checking, polymorphism
modules	as building blocks, encapsulating states and behavior, requiring and implementing interfaces
interfaces	described as sets of methods
methods	atomic, guarded, callable behavior with/without side effects

Bluespec SystemVerilog

A hardware description language based on SystemVerilog:

typing strong, static type-checking, polymorphism

modules as building blocks, encapsulating states and behavior, requiring and implementing interfaces

interfaces described as sets of methods

methods atomic, guarded, callable behavior with/without side effects

rules atomic, guarded behavior snippets in modules, may trigger in every execution cycle (`always` in Verilog), and finish within the same cycle

Bluespec SystemVerilog

A hardware description language based on SystemVerilog:

typing	strong, static type-checking, polymorphism
modules	as building blocks, encapsulating states and behavior, requiring and implementing interfaces
interfaces	described as sets of methods
methods	atomic, guarded, callable behavior with/without side effects
rules	atomic, guarded behavior snippets in modules, may trigger in every execution cycle (`always` in Verilog), and finish within the same cycle
clock	is not explicitly visible (determined by the longest rule)

The compiler generates a conflict free schedule for rules/methods, and needed control logic.

Using BSV

BSV compiles to:

- **SystemC** for modeling alongside other SystemC modules
- **Verilog** for synthesis, easy to combine with other VHDL/Verilog
- **Bluesim** host executable, fast, cycle accurate simulator

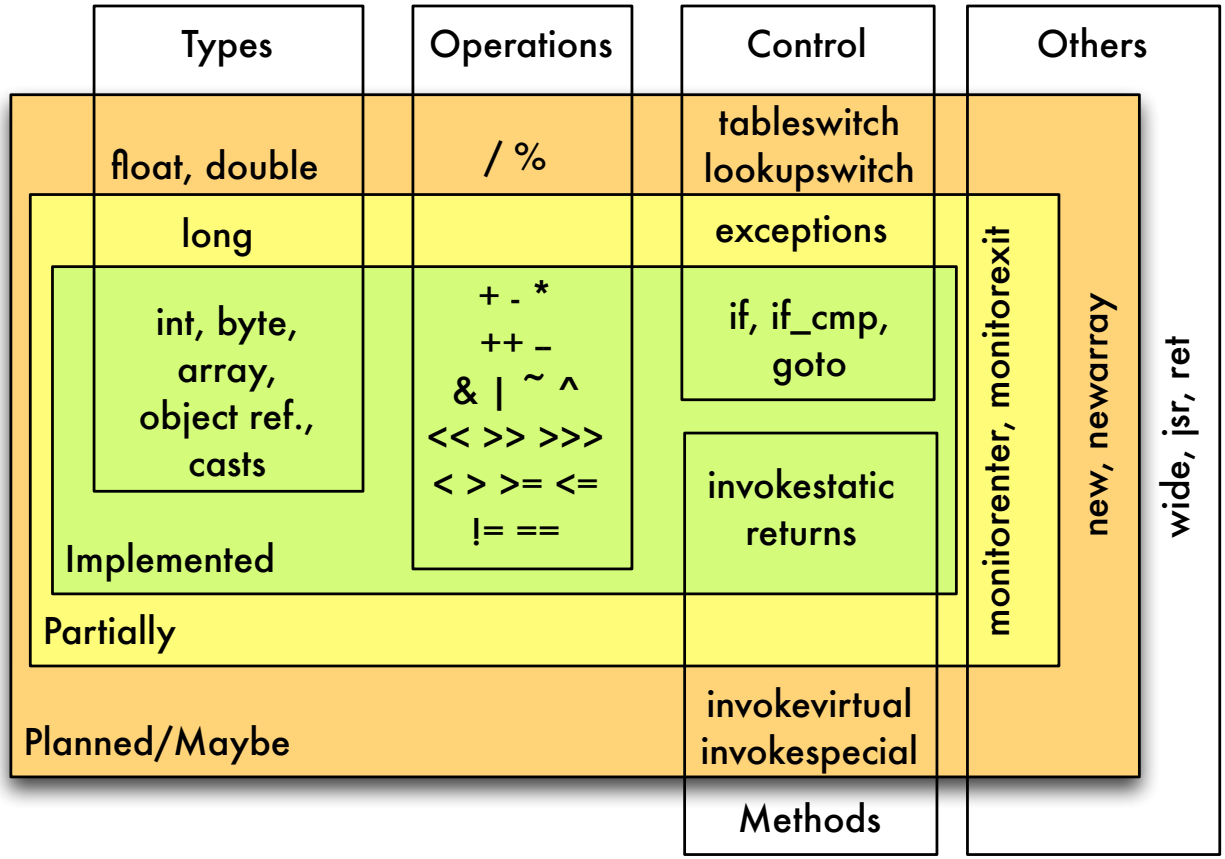
A number of BSV designs have been published, including a Java processor (BlueJEP) with hardware memory management.

Idea

Can we transform sequences of assembly code (Java bytecodes) to hardware using BSV high-level of abstraction constructs?

Our Hardware JVM

A BSV module, providing a subset of bytecodes as interface.
(123 bytecodes as methods and rules)



Bytecodes as Action Methods

Bytecodes transform a computing context into another context ...

- in one clock cycle = one method (see Listing 1)
- over several cycles = start method + several rules (see Listing 2)

Bytecodes as Action Methods

Bytecodes transform a computing context into another context ...

- in one clock cycle = one method (see Listing 1)
- over several cycles = start method + several rules (see Listing 2)

Contexts = operand stack, locals, constant pool address, Java pc

- implemented as lists of registered signals
- registered (saved) explicitly (method) or by certain bytecodes
- restored explicitly (method)

Bytecodes as Action Methods

Bytecodes transform a computing context into another context ...

- in one clock cycle = one method (see Listing 1)
- over several cycles = start method + several rules (see Listing 2)

Contexts = operand stack, locals, constant pool address, Java pc

- implemented as lists of registered signals
- registered (saved) explicitly (method) or by certain bytecodes
- restored explicitly (method)

```
method ActionValue#(Context) isub(Context in);
        let r1 = in.stack[0];
        let r2 = in.stack[1];
        let r = r2 - r1;
        $display("isub␣[..,%d,%d␣->␣..,%d]",r1,r2,r);
        return Context {stack:cons(r, drop(2,in.stack)),
                locals:in.locals,cp:in.cp, jpc:in.jpc+1};
endmethod
```

Bytecodes to BSV Details

Sequences of bytecodes $\longrightarrow$ basic-*ish* blocks $\longrightarrow$ guarded rules:

guards are specific method id, Java pc

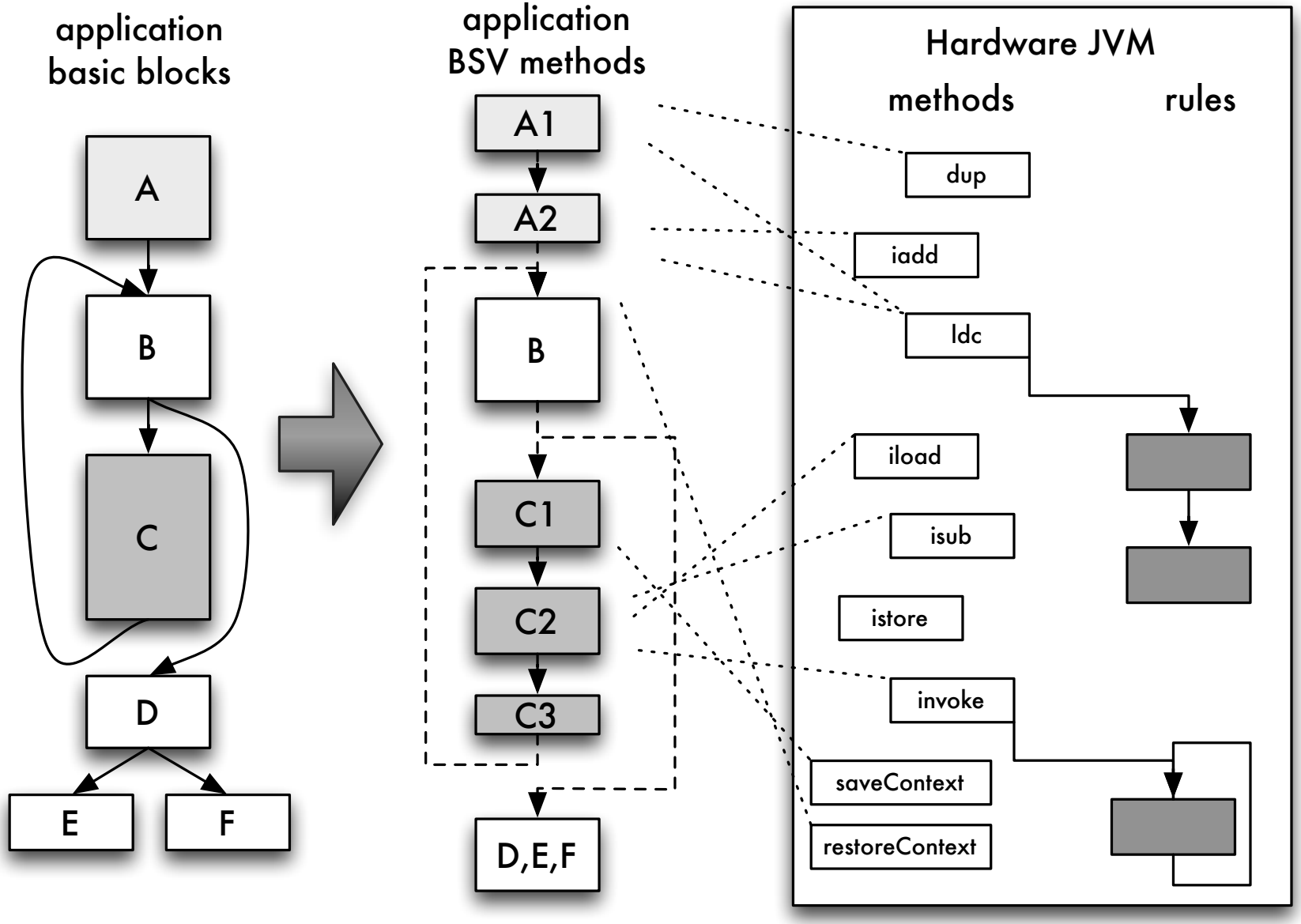
start by building (restoring) context from registers

end with saving context explicitly, or multi-cycle bytecodes

```
rule methodA_lXtolY( !jvm.busy() &&
        jvm.getCurrentMethod() == IdA && jvm.getCurrentJPC() == X );
        let in <- jvm.restoreContext();
        let lX <- jvm.bytecode1(in, opd);
        // ... more bytecode method calls ...
        let out <- jvm.bytecodeN(in, opd1, opd2);
        jvm.saveContext(out); // or
        // invokestatic, return, getstatic, ldc, ...
endrule
```

The choices: one rule per method (sometimes) $\longleftrightarrow$ one rule per bytecode

Bytecodes to BSV Concept



Advanced Features

invokes & recursion supported via a specified size context stack, limiting the depth of calls

object access using **JOP/BlueJEP** memory layout, through an **OPB** bus

memory management `new` bytecodes, garbage collection should be handled by the companion processor

exceptions have limited support, stack restore & jumps

multi-threading is limited, only as independent hardware JVMs.

Tools and Setup

Synthesis $\rightarrow$ device area, maximum clock frequency

- BSV compiler 2012.01.A, *BSV* $\rightarrow$ *Verilog*
- Xilinx ISE 14.1, *Verilog* $\rightarrow$ *FPGA*
- FPGA, Xilinx Spartan-6 (XC6SLX16)

Tools and Setup

Synthesis → device area, maximum clock frequency

- BSV compiler 2012.01.A, *BSV* → *Verilog*
- Xilinx ISE 14.1, *Verilog* → *FPGA*
- FPGA, Xilinx Spartan-6 (XC6SLX16)

Simulation → executed clock cycles

- Desktop, Linux
- BSV compiler 2012.01.A, *BSV* → *Bluesim* (executable)
- custom tools for parsing the output from instrumented code, as well as estimate JOP timing

Tools and Setup

Synthesis → device area, maximum clock frequency

- BSV compiler 2012.01.A, *BSV* → *Verilog*
- Xilinx ISE 14.1, *Verilog* → *FPGA*
- FPGA, Xilinx Spartan-6 (XC6SLX16)

Simulation → executed clock cycles

- Desktop, Linux
- BSV compiler 2012.01.A, *BSV* → *Bluesim* (executable)
- custom tools for parsing the output from instrumented code, as well as estimate JOP timing

Applications : ours (hand coded) vs. JOP software vs. Hanna2011 [17]

- **GCD** Euclid's algorithm, $(12365400, 906)$
- **Sieve2** Eratosthenes sieve, 100 primes
- **Qsort** recursive, 4000 values

Performance

Application	Method	Clock Cycles	Max. Clock (MHz)	Time (ms)
GCD	BSV (8 rules)	27,348	151	0.181
	Hanna	54,652	200	0.273
	JOP	218,790	93	2.353
Sieve2	BSV (12 rules)	32,475	152	0.214
	Hanna	16,023	125	0.128
	JOP	113,198	93	1.217
Qsort	BSV (30 rules)	1,669,820	117	14.272
	Hanna (Iter.)	486,520	125	3.892
	JOP	4,377,628	93	47.071

Device area: published data to compare to is lacking, see Table 2.

Finally...

Summary A method for translating Java bytecode sequences to hardware, via Bluespec SystemVerilog,

- well suited for automation
- intended for acceleration (e.g. of JOP, BlueJEP)

To Do

- complete the translator tool
- optimize the partitioning into rules
- add simple `new` bytecodes implementations

Thank you!

Thank you!

Questions?