Code Generation for Embedded Java with Ptolemy

Martin Schoeberl, Christopher Brooks, and Edward A. Lee

UC Berkeley, Berkeley, CA, USA
{mschoebe,cxh,eal}@eecs.berkeley.edu

Abstract. Code generation from models is the ultimate goal of model-based design. For real-time systems the generated code must be analyzable for the worst-case execution time (WCET). In this paper we evaluate Java code generation from Ptolemy II for embedded real-time Java. The target system is the time-predictable Java processor JOP. The quality of the generated code is verified by WCET analysis for the target platform. Our results indicate that code generated from synchronous data-flow and finite state machine models is WCET analyzable and the generated code leads to tight WCET bounds.

1 Introduction

In this paper we investigate Java code generation from a model-based design. Specifically we use Ptolemy II [4] for the modeling and code generation and target the embedded Java platform JOP [12]. Ptolemy II is extended with run-time modeling actors to represent low-level I/O operations of the target platform in the simulation environment. The Ptolemy II code generation framework is extended with code generating actors to implement the low-level I/O operations.

The target platform includes a worst-case execution time (WCET) analysis tool. Therefore, we are able to analyze the WCET of various actors provided by Ptolemy II and the WCET of the generated application. Comparing WCET analysis with measured execution time on the target shows that code generated from Ptolemy II models results in Java code where a tight WCET bound can be derived. WCET analysis also reveals that modeling of applications without considering the execution time (e.g., using double data type when not necessary) can result in surprisingly high execution time.

The paper is organized as follows: in the remainder of this section we give some background information on Ptolemy II and JOP. In Section 2 code generation from models and the implications for low-level I/O operations on an embedded platform are described. Implementation details on the embedded platform with a few example applications is given in Section 3. In Section 4 we show that the generated code is WCET analyzable and usually leads to tight WCET bounds. We discuss our findings in Section 5. The paper is concluded in Section 6.

1.1 Ptolemy II

Ptolemy II [4] is a modeling and simulation tool for heterogenous embedded systems. Ptolemy II allows generation of C and Java code from the models. We use the latter feature to build embedded applications targeted for a Java processor.

Ptolemy II is a Java-based component assembly framework with a graphical user interface called Vergil. The Ptolemy project studies modeling, simulation, and design of concurrent, real-time, embedded systems. The principle in the project is the use of well-defined models of computation that give the individual models execution semantics.

In Ptolemy II, models of computations are also known as domains. In this paper we focus on the synchronous dataflow (SDF) domain [9] combined with modal models [5]. The Giotto [7] and the synchronous/reactive [3] domain are also interesting code generation targets and we assume they would work with the presented infrastructure. As target platform we have chosen an embedded real-time Java processor.

The building blocks of a Ptolemy II model are actors. Actors exchange data by passing tokens between ports that connect actors. The actual execution semantics of connected actors is defined by the model of computation or execution domain. In our evaluation, the synchronous data-flow (SDF) domain is explored. SDF handles computations on streams. An SDF model constructs a network of actors that are then scheduled for execution. The schedule of the firings can be computed statically [9], making this domain an easy target for generation of efficient code.

The modal model [5] domain adds finite state machines (FSM) to the computing power of the SDF domain. Modal models can represent simple FSMs or provide an abstraction of mode changes. In the later case the FSM represents different modes in the applications and the states are refined by submodels for each mode.

1.2 The Java Processor JOP

We target embedded Java systems with our code generation framework. As a first example we use JOP [12], a Java processor especially designed for real-time systems. JOP implements the Java virtual machine (JVM) in hardware. That means bytecode, the instruction set of the JVM, is the instruction set of JOP. The main feature of JOP is the ability to perform WCET analysis at bytecode level. This feature, and a special form of instruction caching, simplify the low-level part of the WCET analysis. Furthermore, performing WCET analysis at the bytecode level is simpler than performing it at the executable level for compiled C/C++ programs. In Java class files more information is available, e.g., receiver types, than in a *normal* linked executable.

The runtime environment of JOP is optimized for small embedded applications with real-time constraints. JOP includes a minimal version of the Java library (JDK) and provides a priority based thread scheduler. Real-time threads on JOP are either periodic (time-triggered) or software driven event triggered. Hardware events can trigger an interrupt handler. The whole system, including the scheduler and interrupt handlers, is written in Java. In the context of code generated from Ptolemy II we use periodic threads to execute the models.

2 Code Generation from Models

Code generation for embedded systems is challenging due to several constraints. Embedded systems have severe resource limitations in processing power and in memory size. Although Java is known to have quite a large memory footprint (counted in MB), JOP's memory footprint is in the range of several KB. As a consequence, only a small part of the Java library is available. Code generated from Ptolemy II models is conservative enough to target small embedded Java systems.

2.1 Code Generation with Ptolemy II

Within Ptolemy II several approaches for code generation have been evaluated.

Copernicus: A code generation framework that included "deep code generation", where Java bytecode of the actors is analyzed and specialized for use in the particular model [11]. Copernicus deep code generation generated code for non-hierarchical, synchronous dataflow (SDF) models. JHDL [10] used Copernicus to generate FPGA code. However, one drawback of the deep code generation approach is that it is affected by changes to the Ptolemy II runtime kernel.

Codegen: A template based approach [20] where we define adapter classes associated with Ptolemy II actors. The adapter classes define the C or Java implementation or the corresponding Ptolemy II actor. The code generator performs type and width inference on the model and generates optimized code.

CG: Another template based approach that is currently under development. The CG effort is focused on composite code generation [19], where reusable code is generated for each group of actors. Composite code generation will allow model developers to modify the model and generate code only for changed portions. Composite code generation also allows generation of Java code for very large models.

For the experiments targeting JOP we use the Codegen and the CG code generation frameworks. Support for the low-level I/O with JOP has been added to both systems.

2.2 Runtime Support on JOP

Code generated for the SDF model of computation is usually executed periodically. The period is defined in the SDF director of the model. On JOP we leverage the available real-time scheduler and execute the SDF model in a periodic thread. Multiple periodic threads can be used to combine the SDF model with other code, written in Java or generated from a model. To reason about the timing properties of the whole system, all (periodic) threads need to be WCET analyzable. Given the periods and the execution time, schedulability analysis gives an answer if all deadlines will be met.

If the application consists of just an SDF model, then a simple, single threaded approach, similar to a cyclic executive, is also possible. The release to execute an iteration of the SDF model can be synchronized via an on-chip clock. At the end of one iteration the processor performs a busy wait, polling the clock, until the next release time. Cycle accurate periods can be obtained by performing this wait in hardware. We have implemented this hardware support, called *deadline instruction*, in JOP [16].

As Ptolemy II is mainly used for simulation, it lacks actors for interfacing low-level I/O. Within JOP we use so called hardware objects [14] to represent I/O ports as standard Java objects. We have implemented input and output actors for several I/O devices on JOP. Details on the I/O actors are given in the next subsections.

Listing 1.1. Implementation of a low-level output actor

```
public class JopSerialWrite extends Sink {

   public JopSerialWrite(...) {}

   private int lastVal;
   private int val;

   // Save the value for the output port.
   public void fire() {
      super.fire();
      lastVal = val;
      if (input.hasToken(0)) {
         lastVal =
            ((IntToken) input.get(0)).intValue();
      }
   }

   // Write the saved value to the output port.
   public boolean postfire() {
      val = lastVal;
      writePort(val);
      return super.postfire();
   }
}
```

2.3 Semantics of an Actor

Actors are the components in Ptolemy II that represent computation, input, and output. Details of actor semantics can be found in [10]. The two methods central to an actor are `fire()` and `postfire()`: The `fire()` method is the main execution method. It reads input tokens and produces output tokens. In some domains, `fire()` is invoked multiple times, e.g., to find a fixpoint solution in the synchronous/reactive domain. The `postfire()` method is executed once and shall update persistent state. The main points an actor developer has to consider are when to calculate the output (`fire()`) and when to update state (`postfire()`).

2.4 Input/Output Actors

An I/O operation often updates persistent state, e.g., output of a character to a terminal updates persistent state. Therefore, `fire()` is not the method where the I/O operation shall happen. For output the output data (token) needs to be saved in `fire()`, but the actual output action has to happen in `postfire()`.

It is less obvious, but reading an input value can also change the state of the system. As an example, reading from the input buffer of a communication channel (the serial line), consumes that value. Therefore, the actual read operation has to happen exactly once. The input value is read in the first in invocation of `fire()` per iteration and saved in a local variable. The output token for the read actor uses the value of that local variable on each invocation in one iteration. The start of a new model iteration is marked within the `postfire()` method and an actual I/O read is performed at the next `fire()`.

In the following we give examples of I/O actors for the serial port on JOP. For the access to the device registers we use *hardware objects* that represent I/O devices as plain

Listing 1.2. Java code template for a low-level output actor

```
/***preinitBlock***/
int $actorSymbol(val) = 0;
int $actorSymbol(lastVal);
static com.jopdesign.io.SerialPort
   $actorSymbol(ser) =
   com.jopdesign.io.IOFactory.
      getFactory().getSerialPort();
/**/

/*** fireBlock($channel) ***/
$actorSymbol(lastVal) = $actorSymbol(val);
// no check on available input token here
// so above assignment is useless
$actorSymbol(lastVal) = $get(input#$channel);
/**/

/*** postfireBlock ***/
$actorSymbol(val) = $actorSymbol(lastVal);
if (($actorSymbol(ser).status &
   com.jopdesign.io.SerialPort.MASK_TDRE)!=0) {

   $actorSymbol(ser).data = $actorSymbol(val);
}
/**/
```

Java objects [15]. The hardware objects are platform specific as they are representing platform specific I/O devices. However, the hardware object that represents a serial port interface, used in this section, has been used without alteration in five different JVMs (CACAO, OVM, SimpleRTJ, Kaffe, and JOP [14]).

Output Actor. Listing 1.1 shows the anatomy of a low-level output actor. The code shows how the output actor is implemented for the simulation mode in Ptolemy II. Within `fire()` an input token is read and stored in an actor private variable. The actual output, the state changing action, is performed in `postfire()`, which is guaranteed to be executed only once and after `fire()`.

The code generator can remove most of the overhead involved in methods calls. Therefore, the code template, shown in Listing 1.2, contains only the core logic. Block markers, within the `/*** ***/` style comment, delimit a code block. Macros (symbols starting with $) in the code template are substituted by the code generator. The macro `$actorSymbol(val)` generates a static variable with a unique name for the actor local variable `val`. `$get(input#$channel)` reads a token from a channel of the input port. Actor parameters, attributes assigned at design time, can be accessed with `$param(name)`.

The code example shows the usage of a hardware object to access the serial port. The hardware object, which represents the serial port registers, is *created* by a factory method. As hardware, and the representing objects, cannot be just created with `new`, a system specific factory is used to return an instance of a hardware object. In the `postfireBlock` the device is accessed via this hardware object.

Listing 1.3. Implementation of a low-level input actor

```
public class JopSerialRead extends Source {

   public JopSerialRead(...) {}

   private boolean firstFire = true;
   private IntToken val = new IntToken(0);

   // Read the input port on the first
   // invocation of fire(). Send the value
   // on each fire().
   public void fire() {
      super.fire();
      if (firstFire) {
         int v = readSerialPort();
         val = new IntToken(v);
         firstFire = false;
      }
      output.send(0, val);
   }

   // Enable read for the next fire().
   public boolean postfire() {
      firstFire = true;
      return super.postfire();
   }
}
```

Although the template code looks a little bit verbose, the generated code using this template is quite short. The `fire()` and `postfire()` methods are inlined to avoid the overhead of the method invocation.

Input Actor. Listing 1.3 shows the concept of an input actor. This code example shows how an input actor is implemented for the simulation. An actor private Boolean variable is used to detect the first invocation of `fire()` to perform the actual I/O read function and save the value. On future invocations of `fire()` that value is returned. The flag to indicate the first firing of the actor is reset in `postfire()`. The code template for the input actor is similar to the one shown for the output actor.

3 Implementation

Models are executed on the target platform periodically. The period can be configured in the SDF director. We changed the code generator to output the configured period as a constant. On the target platform the model class is instantiated, initialized, and executed. For the periodic execution we use the periodic real-time threads available in the JOP runtime. The code in Listing 1.4 shows the execution of the model.
For the periodic execution of the generated model we had to change the Java code generator to provide access to the period of the SDF director. If the period of the SDF director is set to 0 the model is executed in a tight loop – in fact it is running at the maximum possible frequency.[1]

[1] In the implementation all values less than 100 microseconds are treated as period 0.

Listing 1.4. Execution of the model in a periodic thread

```
final Model model = new Model();
model.initialize();

int us = (int) (model.PERIOD * 1000000);
// If there is a useful period, run it in a periodic thread.
// If not, just in a tight loop.
if (us >= 100) {
   new RtThread(1, us) {
      public void run() {
         for (;;) {
            try {
               model.run();
            } catch (Exception e) {
               e.printStackTrace();
            }
            waitForNextPeriod();
   }}};
   RtThread.startMission();
} else {
   for (;;) {
      try {
         model.run();
      } catch (Exception e) {
         e.printStackTrace();
}}}
```

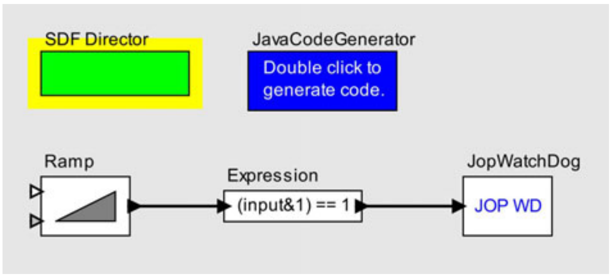

Fig. 1. A simple Ptolemy II model generating the watchdog trigger

3.1 Embedded Hello World

The first low-level I/O actor we implemented was the interface to the watchdog LED. With output to a single LED the embedded version of Hello World – a blinking LED – can be designed with Ptolemy II. Figure 1 shows a simple model that toggles the watchdog LED. The rightmost block is a sink block that sets the LED according to the received Boolean token. With a Ramp block and an Expression block the LED blinks at half of the frequency of the SDF director when executing the generated code on the target.

3.2 An SDF Example with a State Machine

Besides data flow actors that are driven by the synchronous data flow director, finite state machines (FSM) are needed to build meaningful embedded applications. We have ported the FSM code generator for C to the Java based code generation.

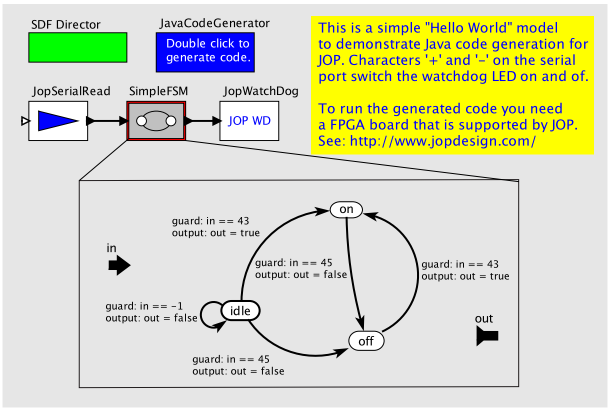
SDF Director
JavaCodeGenerator
Double click to generate code.
This is a simple "Hello World" model to demonstrate Java code generation for JOP. Characters '+' and '-' on the serial port switch the watchdog LED on and of.
To run the generated code you need a FPGA board that is supported by JOP. See: http://www.jopdesign.com/
JopSerialRead
SimpleFSM
JopWatchDog
JOP WD
in
out
guard: in == 43
output: out = true
on
guard: in == 45
output: out = false
guard: in == 43
output: out = true
guard: in == -1
output: out = false
idle
off
guard: in == 45
output: out = false

Fig. 2. A Ptolemy II model with a serial interface, an FSM, and the watchdog LED with the inner FSM shown

Figure 2 shows an example containing all ingredients to build embedded programs with Ptolemy II for JOP. The serial port actor reads commands from a host and forwards them to the state machine, which itself generates the output for the watchdog LED. The LED can be turned on with a '+' and turned off with a '-' character received on the serial line.

3.3 Lego Robot

As an example for an application we have modeled a line following robot. The hardware for this experiment is a LEGO Mindstorms robot with an interface of a JOP based FPGA board to the LEGO sensors and actuators. The interface contains several analog inputs, switch inputs, and pulse width modulated (PWM) output for the motors. Furthermore, the motor interface provides an input to read the actual speed of the motors. During the off cycles of the PWM the motor acts as generator and the produced voltage (also called back EMV), which is proportional to the actual revolving speed, can be read. The application periodically samples the input of the light sensor, calculates the control law, and puts out the drive values for the two motors.

4 WCET Analysis

JOP provides a simple low-level timing model for WCET analysis. As almost all Java bytecodes execute in constant time, the WCET analysis can be performed at bytecode level. Due to this simplification, several WCET analysis projects target JOP [6, 8, 17].

For our experiments we use the WCET analysis tool WCA [18], which is part of the JOP distribution. WCA implements two modes of WCET analysis: the implicit path enumeration technique (IPET) and WCET analysis based on model checking [8]. The results presented in this paper are based on the faster IPET mode of WCA.

Static WCET analysis gives an upper bound on the execution time of a program on a specific target platform. Programs are WCET analyzable when all loops and the recursion depths are bound. It is usually impossible to derive the *real* WCET due to two factors: (1) the analysis has not enough knowledge of impossible execution paths and (2) the processor cannot be modeled in full detail. The first issue is usually attacked by annotations in the source code of possible paths, whereas the second issue is solved by a simpler, but safe, model of the hardware. Within the JOP project we try to minimize the overestimation of the WCET bounds by providing an execution pipeline that is simple enough to be modeled for the analysis. However, it has to be noted that WCET analysis will almost always (except for trivial programs) give a conservative estimate, the WCET bound. Furthermore, when comparing WCET values given by the analysis with measurements one has to keep in mind that the measurement usually will not provide the real WCET value. If this would be the case, we would not need to statically analyze the program.

4.1 Model Examples

We have evaluated different models with WCET analysis and execution time measurements on the target. Table 1 shows the analyzed and measured execution time in clock cycles. The `WatchDog` example is the embedded hello world model, as shown in Figure 1. With such a simple model the measured execution time is almost the same as the analyzed time. The WCET for the FSM example `SimpleFSM`, which is the model from Figure 2, contains more control code and during the measurement the data dependent WCET path was not triggered. This model is an example where simple measurement does not give the correct WCET value.

Table 1. WCET bounds and measured execution time for different models on JOP

Model	WCET (cycles)	Measured (cycles)
`WatchDog`	277	244
`SimpleFSM`	9539	3843
`Filter`	1818	1734
`Follower`	3112	2997
`Add (int)`	656	604
`Add (double)`	19545	12092

The third application, `Filter`, represents a digital signal processing task with a finite impulse response filter. The source for the filter is a pulse generating actor. As the generated code is basically straight line code, the measured execution time and the WCET bound are almost identical.

The `Follower` example is the line following LEGO robot. Again, the generated code is easy to analyze and the WCET bound is tight. From the examples without data dependent control flow we see that the WCET bound is close to the measured execution time.

4.2 Data Types

The four examples use integer data types in the models. For an embedded system without a floating point unit this is the preferable coding style. JOP does not contain a floating point unit (FPU).[2] Floating point operations on JOP are supported via a software emulation of the floating point bytecodes. To demonstrate how expensive double operations (the default floating point type in Java) are, we compare a model with two ramp sources, an adder, and a sink actor for 32-bit integer data with the same model for 64-bit double data. In Ptolemy II the models are identical. Only the port type needs to be changed for the requested data type. The code generation framework then automatically instantiates the correct type of the actor. The example model (`Add (double)`) with double data type executes in the worst case about 30 times slower than the integer version. This illustrates the importance of choosing the right data type for the computation on an embedded platform. A developer, educated on standard PCs as computing platform is usually not aware of this issue, which is typical for a resource constraint embedded platform. Using WCET analysis, right from the beginning of the development, is important for economic usage of computing resources.

5 Discussion

The integration of JOP into Ptolemy II was relative straightforward. One of the authors had no real knowledge on the usage of Ptolemy II in general and the code base specifically. The documentation of Ptolemy II [2] and the source code are well organized. Getting used to design a few simple test models was a matter of hours. Adding new actors in the simulation and in the Java code generation base took less than two days, including familiarizing with the Ptolemy II sources.

On the JOP side we had to update the provided library (JDK) to better support double data types. As double operations are very expensive on JOP, their support was quite limited.

5.1 Low-Level I/O

Java, as a platform independent language and runtime system, does not support direct access to low-level I/O devices. One option to access I/O registers directly is to access them via C functions using the Java native interface. Another option is to use so called hardware objects [14], which represent I/O devices as plain Java objects. The hardware objects are platform specific (as I/O devices are), but the mechanism to represent I/O devices as Java objects can be implemented in any JVM. Hardware objects have been implemented so far in five different JVMs: CACAO, OVM, SimpleRTJ, Kaffe, and JOP. Three of them are running on a standard PC, one on a microcontroller and one is a Java processor. It happened that the register layout of the serial part was the same for all three hardware platforms. Therefore, the hardware object for the serial device, and the serial port actor, can be reused for all five JVMs without any change.

[2] An FPU for 32-bit float is available in the source distribution. As that unit is twice as large as JOP, the FPU is disabled per default.

5.2 Memory Management

One of the most powerful features of Java is automatic memory management with a garbage collector. However, the runtime implications of garbage collection in a real-time setting are far from trivial. The real-time specification for Java [1] provides scoped memory as a form of predictable dynamic memory management. The actors we used in our examples do not allocate new objects at runtime. For more complex actors, which use temporary generated objects, scoped memory can be used.

Another source of created objects are the tokens for the communication between the actors. In our examples we used only primitive data types and the code generation framework maps those token to primitive Java types. For complex typed tokens a solution for the generated objects need to be found, either using (and trusting) real-time garbage collection or provide a form of token recycling from a pool of tokens.

6 Conclusion

In this paper we have evaluated code generation for embedded Java from Ptolemy II models. The code generation framework can generate C and Java code. For our embedded Java target, the real-time Java processor JOP, we explored the Java code generation. Code generation in Ptolemy is template based and generates simple enough code that is WCET analyzable. The combination of the time-predictable Java processor as target and the generated code from Ptolemy enables WCET analysis of model-based designs. We argue that this combination is a step towards model-based design of hard real-time systems.

Both projects, Ptolemy II and JOP, used in the paper are available in source form; see `http://chess.eecs.berkeley.edu/ptexternal` and `http://www.jopdesign.com/`. The build instructions for JOP are available in Chapter 2 of the JOP Reference Handbook [13]. An example for the execution of code generated from Ptolemy models in a periodic thread can be found in `java/target/src/test/ptolemy/RunIt.java`.

References

1. Bollella, G., Gosling, J., Brosgol, B., Dibble, P., Furr, S., Turnbull, M.: The Real-Time Specification for Java. Java Series. Addison-Wesley, Reading (June 2000)
2. Brooks, C., Lee, E.A., Liu, X., Neuendorffer, S., Zhao, Y., Zheng, H.: Heterogeneous concurrent modeling and design in Java. Technical Report UCB/EECS-2008-28/29/37, University of California at Berkeley (April 2008)
3. Edwards, S.A., Lee, E.A.: The semantics and execution of a synchronous block-diagram language. Science of Computer Programming 48(1) (2003)
4. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. Proceedings of the IEEE 91(2), 127–144 (2003)
5. Girault, A., Lee, B., Lee, E.A.: Hierarchical finite state machines with multiple concurrency models. IEEE Transactions on Computer-aided Design of Integrated Circuits And Systems 18(6), 742–760 (1999)
6. Harmon, T.: Interactive Worst-case Execution Time Analysis of Hard Real-time Systems. PhD thesis, University of California, Irvine (2009)

7. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered language for embedded programming. Proceedings of the IEEE 91(1), 84–99 (2003)
8. Huber, B., Schoeberl, M.: Comparison of implicit path enumeration and model checking based WCET analysis. In: Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis, Dublin, Ireland, pp. 23–34. OCG (July 2009)
9. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proceedings of the IEEE 75(9), 1235–1245 (1987)
10. Lee, E.A., Neuendorffer, S., Wirthlin, M.J.: Actor-oriented design of embedded hardware and software systems. Journal of Circuits, Systems, and Computers 12(3), 231–260 (2003)
11. Neuendorffer, S.A.: Actor-Oriented Metaprogramming. PhD thesis, EECS Department, University of California, Berkeley (January 2005)
12. Schoeberl, M.: A Java processor architecture for embedded real-time systems. Journal of Systems Architecture 54(1-2), 265–286 (2008)
13. Schoeberl, M.: JOP Reference Handbook: Building Embedded Systems with a Java Processor. CreateSpace (August 2009) ISBN 978-1438239699, `http://www.jopdesign.com/doc/handbook.pdf`
14. Schoeberl, M., Korsholm, S., Kalibera, T., Ravn, A.P.: A hardware abstraction layer in Java. Trans. on Embedded Computing Sys., (accepted 2010)
15. Schoeberl, M., Korsholm, S., Thalinger, C., Ravn, A.P.: Hardware objects for Java. In: Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008), Orlando, Florida, USA, pp. 445–452. IEEE Computer Society, Los Alamitos (May 2008)
16. Schoeberl, M., Patel, H.D., Lee, E.A.: Fun with a deadline instruction. Technical Report UCB/EECS-2009-149, EECS Department, University of California, Berkeley (October 2009)
17. Schoeberl, M., Pedersen, R.: WCET analysis for a Java processor. In: Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006), pp. 202–211. ACM Press, New York (2006)
18. Schoeberl, M., Puffitsch, W., Pedersen, R.U., Huber, B.: Worst-case execution time analysis for a Java processor. Software: Practice and Experience 40(6), 507–542 (2010)
19. Tripakis, S., Bui, D.N., Geilen, M., Rodiers, B., Lee, E.A.: Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs. Technical Report UCB/EECS-2010-52, UC Berkeley (May 2010)
20. Zhou, G., Leung, M.-K., Lee, E.A.: A code generation framework for actor-oriented models with partial evaluation. In: Lee, Y.-H., Kim, H.-N., Kim, J., Park, Y.W., Yang, L.T., Kim, S.W. (eds.) ICESS 2007. LNCS, vol. 4523, pp. 786–799. Springer, Heidelberg (2007)