BUILDING EMBEDDED DSP APPLICATIONS IN A JAVA MODELING FRAMEWORK*

Isidoros Sideris, Dimitris Pilitsos, George Economakos and Kiamal Pekmestzi

School of Electrical and Computer Engineering, National Technical University of Athens
Iroon Polytexneiou 9, GR-15780 Athens, Greece
phone: + (30) 210-7221800, fax: + (30) 210-7722428, email: isidoros@microlab.ntua.gr
web: www.microlab.ntua.gr

ABSTRACT

In this paper we present a methodology for modeling embedded DSP applications based on the Java language. Models are written in a framework which allows detailed cycle accurate modeling in the area of focus with less detailed and more efficient statistical models for the rest. The modeling is based on a Java extension package that allows hardware modeling by providing classes and primitive calls, in the same way SystemC extends C. By adopting Java in hardware modeling great productivity gains can be brought to the hardware designer or computer architect, since it has simple object model and prevents us from the error prone pointers. As a modeling example, we evaluate the performance of an audio encoding application on top of a stack folding optimized Java processor.

1. INTRODUCTION

DSPs have become a ubiquitous enabler for integration of audio, video and communications. As 3G mobile phones and applications are prevailing, efficient Java execution is becoming a crucial component in system performance. Java processors [10] have been introduced to offer hardware acceleration for Java applications. However, their complex instruction set and the close relationship of hardware and software makes experimentation and evaluation of new microarchitectural techniques difficult.

In this paper we propose a methodology for experimentation of microarchitectural techniques for embedded DSP applications executed on a Java processor. We discuss trace driven and execution driven performance evaluations. All implementations use a modeling framework we built on top of Java language, which enables modeling of communicating threads and supports various levels of details throughout the whole design model.

1.1 Parallel and Statistical Modeling

When modeling a processor core in order to evaluate its performance, we are likely to use a model in simplescalar [7, 3] style, in which we take into consideration buffer sizes and statistical behaviors of the subsystems. In some cases performance is very dependent on the communication and execution overlap of two units, something that it can't be modeled in such a framework. A hardware model with communicating parallel threads is needed. But this is at the expense of performance.

*This work was partially funded by the Greek Ministry of Development, General Secretariat for Research and Technology, project PENED 03$E\Delta$908

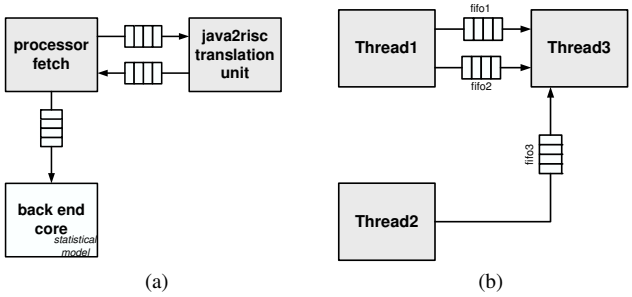

Figure 1: (a) Combined parallel-statistical model, (b) Threads Communicating Example

One solution is to combine different styles of modeling, that is communicating parallel threads in the area you are focusing, and statistical modeling in the other parts of the processor core. A submodel consisting of parallel communicating threads, which simulates a processor subsystem in a cycle accurate basis, communicates with a statistical model thread with FIFOs. The parallel model takes into account unit's operation overlaps, and feeds a buffer of the statistical model through a FIFO. The statistical model takes as input this buffer, and estimates the total performance.

Stats can come either from the statistical model, or from the parallel model by inserting probes in various places. Both model's parameters can be altered, in order to determine how performance is affected.

To make things clear, consider the example of figure 1a, which models a Java processor with a translation unit that transforms java bytecodes to RISC like instructions. The backend processor core is a classic RISC out of order core. The translation of a requested java bytecodes line is variable. We model the processor fetch and the translation unit as two communicating parallel threads. The processor fetch thread feeds the statistical model thread, which models the out of order core. We focus on the translation technique, so we maintain a greater level of detail in this submodel, while keeping an appropriate level in the other parts, in order to keep the whole design realistic.

1.2 Java as a Hardware Modeling Language

The adoption of Java in hardware modeling can provide the hardware designer or computer architect with real productivity gains. Its simple object model and the lack of error prone pointer issues, makes programming really fast and productive. Java platforms, as Eclipse [1], can be used to offer their

```
public class HwMethodRunExample extends HwThread
{
  public boolean hwmethodrun()
  {
    System.out.println("Hello");
    addEvent(new HwEvent(sched,10, "clk"));
    return false;
  }
}
```

(a)

```
public class HwThreadRunExample extends HwThread
{
  public void hwthreadrun()
  {
    synchronized(sched)
    {
      hwwait();
      //setup preamble
      addEvent(new HwEvent(sched,-1, "CLKPOSEDGE"));
      waitcycle();
      //main body
      for(int i=0;i<100;i++)
      {
        System.out.println("["+sched.time+"]"+name+i);
        waitcycle();
      }
      waitcycle();
    }
  }
}
```

(b)

Figure 2: (a) hwmethodrun(), (b) hwthreadrun()

debugging facilities to the designer. Breakpoints, step run, and full object fields visibility are just some of them.

Utility classes such as gzip classes, graphical classes and various third party can be used to build an efficient and user friendly simulation platform. Distributed virtual machines can also be used, to lower the total simulation time.

The aforementioned types of modeling can both be written in Java. The way of writing the statistical model is straightforward. The parallel model needs a simulation kernel. Java is based on threads, so parallel modeling fits ideally. We have implemented an extension package to Java, that makes feasible this type of modeling. The package consists of classes for modeling a thread, classes for communication between threads. The extension is similar to the extension of SystemC [5, 8] to C.

Our framework combines the SystemC [5] high level design benefits and Java's programming facilities. JHDL [2] is another language for hardware modeling based on Java. However, it targets FPGA design and reconfigurable systems.

2. JAVA HARDWARE MODELING PACKAGE

2.1 Overview

The package uses Java's inherent multithreading to provide structures for hardware modeling. A hardware model consists of communicating threads. The package provides the class HwThread, which is a direct subclass of java.lang.Thread and can be subclassed again by a user class that provides the desired functionality. HwThread class has two basic methods, hwmethodrun() and hwthreadrun(). In a user HwThread subclass, one of them must be overloaded by a method with user logic. All HwThread subclasses' objects

```
public class ClkThread extends HwThread
{
  public boolean hwmethodrun()
  {
    int value=((Integer)sin[0].read()).intValue();
    System.out.println("time="+sched.time+", clk="+
                              value+"THREAD"+name);
    sout[0].write(new Integer((value==0)?1:0));
    addEvent(new HwEvent(sched,-1, "CLK"));
    return true;
  }
  //constructor
  public ClkThread(String name, HwScheduler scheduler,
      HwSignal[] sin, HwSignal[] sout, HwFifo[] infifo,
    HwFifo[] outfifo, HwTransmitFifo[] intransmitfifo,
      HwTransmitFifo[] outtransmitfifo)
  {
    super(name, scheduler, sin, sout, infifo, outfifo,
    intransmitfifo, outtransmitfifo);
  }
}
```

Figure 3: Clock example

are managed by a scheduler (class HwScheduler), which arranges their execution in order to ensure synchronization and time consistency. It also handles the structures needed for communication, by updating signals and FIFOs when simulation time is to be updated, so that threads have the correct input values at each time slot.

Figure 2(a) contains a HwThread subclass which overrides method hwmethodrun(), while in figure 2(b) hwthreadrun() is overridden. hwthreadrun() differentiates from hwmethodrun() in that it can be executed only once. It can suspend its operation by calling a wait method and it can resume from the point where it was suspended. The hwmethodrun() is executed as a function, which does not suspend and continues running until it returns. It can however dynamically modify its sensitivity list, so that it can be executed again when a specified event is triggered.

HwThreadRunExample consists of a setup preamble and the main body. In the preamble the sensitivity list is defined. It can however be changed dynamically in the main body. In the example we make the thread sensitive to positive clock edge. We are releasing the control by calling the waitcycle() method. After the call of this method, the thread will suspend its operation and it will resume when the positive clock edge event is triggered. We have the potential to define our events and trigger them, and make some thread sensitive to our events. However, it is convenient to keep the event triggering mechanism simple by having only the positive edge of the clock as a reference event.

2.2 Communication

The package supports communication between threads through signals and FIFOs. The example of figure 3 contains a thread which reads the signal sin[0] and writes the signal sout[0]. The HwThread class contains some array reference fields for the input and output signals and FIFOs. Before the instantiation of a HwThread object, we create the appropriate signals and FIFOs, then some arrays of them, and finally we pass these arrays to the constructor of HwThread (or a user subclass of HwThread), as illustrated in figure 4. An HwSignal object is created for the signal clk. We add this signal to the input and output list of thread thread1. So, the

```
public class HwThreadInstantiationExample extends
                                          HwScheduler
{
  public static void main(String[] args)
  {
    HwThreadInstantiationExample sched=
           new HwThreadInstantiationExample();
    sched.setDaemon(false);
    sched.start();
  }
  public void setupHwThreads(HwEvent startEvent)
  {
    HwSignal clk=new HwSignal(this, "CLK",
                              new Integer(0));
    HwSignal[] sin={clk};
    HwSignal[] sout={clk};
    HwThread thread1=new ClkThread("clk", this, sin,
                        sout, null, null, null, null);
    waitingList.add(thread1);
    thread1.addEvent(startEvent);
    //start threads
    thread1.start();
  }
}
```

Figure 4: HwThread instantiation

thread behaves as a clock thread, since it alternates the signal clk periodically. The signal is implemented internally as two values, currentvalue and value2update. The threads write the value2update variable. When it comes the time for time update, the currentvalue variable takes the value of value2update variable, so that the threads have the new value at their disposal at the very next time slice.

Consider the example of figure 1b where three parallel threads communicate with FIFOs. The code snippets in figures 5 and 6 show the use of FIFOs. FIFOs are objects of the class HwFifo. They are handled with the methods insert() and remove(). Note that a value inserted in the FIFO is not available to other threads until the next time slot. The FIFOs are very handy, since we can insert objects of any class, including user defined classes.

2.3 Scheduling Threads & Event Management

The scheduler works as follows. It maintains two queues of threads, the readylist which contains all the threads ready to run, and the waiting list which contains threads not yet ready to run (figure 7). The threads in the waiting list are waiting for the triggering of some event in their sensitivity list. When this comes they leave the waiting list and enter the ready list.

As long as the ready list contains threads, the scheduler removes one and dispatches it for execution. When the readylist gets empty, it triggers the closest in time event. This action fills the readylist with some threads. It advances also the time to the time of this event. The events can be created by the threads while executing, or as an effect of the signal update action if a transition is detected. Either the source, they are inserted in the queue with a certain timestamp.

For each thread extracted from the ready list and executed in the allotted time slot, references to their output signals and FIFOs are added to a list, so that they could be updated at the end of the current time slice.

```
public class Thread1 extends HwThread
{
  public void hwthreadrun()
  {
    synchronized(sched)
    {
      hwwait();
      addEvent(new HwEvent(sched,-1,"CLKPOSEDGE"));
      HwFifo fifo1=outfifo[0];
      HwFifo fifo2=outfifo[1];
      waitcycle();
      for(int i=0;i<100;i++)
      {
        System.out.println("Thread1:"+i);
        fifo1.insert(Integer.toString(i));
        fifo2.insert(new Object());
        waitcycle();
      }
      resetEvents();
      sched.notifyAll();
      hwwait();
    }
  }
}
```

(a)

```
public class Thread2 extends HwThread
{
  public void hwthreadrun()
  {
    synchronized(sched)
    {
      hwwait();
      addEvent(new HwEvent(sched,-1,"CLKPOSEDGE"));
      HwFifo fifo3=outfifo[0];
      waitcycle();
      for(int i=0;i<100;i++)
      {
        System.out.println("Thread2:"+i);
        fifo1.insert(new Integer(i*5));
        waitcycle();
      }
      resetEvents();
      sched.notifyAll();
      hwwait();
    }
  }
}
```

(b)

```
public class Thread3 extends HwThread
{
  public void hwthreadrun()
  {
    synchronized(sched)
    {
      hwwait();
      addEvent(new HwEvent(sched,-1,"CLKPOSEDGE"));
      HwFifo fifo1=infifo[0];
      HwFifo fifo2=infifo[1];
      HwFifo fifo3=infifo[2];
      waitcycle();
      for(int i=0;i<100;i++)
      {
        System.out.println("Thread3:"+i);
        if(!fifo1.isEmpty())
          System.out.println((String)fifo1.remove());
        Object obj=fifo2.remove();
        int i=0;
        if(!fifo3.isEmpty())
          i=((Integer)fifo3.remove()).intValue();
        waitcycle();
      }
      resetEvents();
      sched.notifyAll();
      hwwait();
    }
  }
}
```

(c)

Figure 5: Communicating threads example

```
public class FifosExample extends HwScheduler
{
  public void setupHwThreads(HwEvent startEvent)
  {
    //clkthread - omitted for clarity
    HwFifo fifo1=new HwFifo();
    HwFifo fifo2=new HwFifo();
    HwFifo fifo3=new HwFifo();

    //thread1
    HwFifo[] outfifo={fifo1, fifo2};
    HwThread thread1=new Thread1("thread1", this, null,
                        null, infifo, null, null, null);
    waitingList.add(thread1);
    thread1.addEvent(startEvent);

    //thread2
    HwFifo[] outfifo={fifo3};
    HwThread thread2=new Thread2("thread2", this, null,
                        null, infifo, null, null, null);
    waitingList.add(thread2);
    thread2.addEvent(startEvent);

    //thread3
    HwFifo[] outfifo={fifo3};
    HwThread thread3=new Thread3("thread3", this, null,
                        null, infifo, null, null, null);
    waitingList.add(thread3);
    thread3.addEvent(startEvent);

    //start threads
    clkthread.start();
    thread1.start();
    thread2.start();
    thread3.start();
  }
}
```

Figure 6: Communicating threads example

2.4 More Advanced Communication Mechanisms

In modeling processor pipelines, we usually have the two phases of the clock as reference events. Since we are modeling for performance, we do not want to get involved with subtle timing details. We may want to communicate with another subunit within the current cycle, that is to request data from a subunit and get the response. With HwFifo this is not possible, since the request will arrive the next clock cycle, and the response one cycle later. Thus, we can't model a combinational subunit.

To address this problem, we have added a variation of HwFifo, the HwTransmitFifo class. This class provides the methods transmit() and receive() for communication within the cycle. They are based on the special wait method waitdt(), which suspends for a minimum simulation step time dt. The cycle time is partitioned. Now the time has two components, a major time, and a minor time, which is the count of dts. The maximum time of dts in a given cycle, is determined by the thread with the maximum number of waitdt() calls. transmit() and receive() work as follows. Whenever a thread (TA) wants to communicate with another one (TB) within cycle, it calls the transmit() method. TB calls receive(). receive() calls continually the method waitdt() until a value is inserted in the corresponding FIFO. Then it removes the value, as in the normal FIFO. The waitdt() method releases control. The thread resumes in time t+dt.

HwTransmitFifo provides also a non blocking read method read(). Last but not least, the contents of the FIFO are removed every clock cycle, so as to reflect communication within cycle. Method read() can retrieve the last updated value, as if it were a signal.

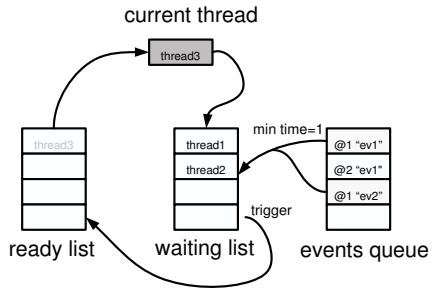

Figure 7: Scheduler

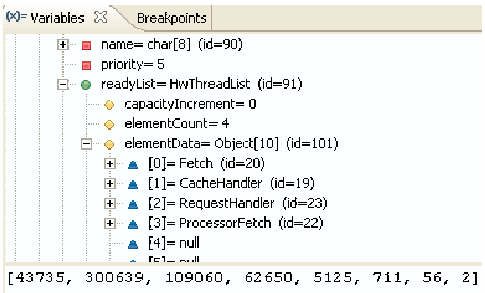

Figure 8: Variables view of Eclipse platform

2.5 Eclipse as a Debugging Platform

A Java IDE environment can be used for debugging our models. We have used the Eclipse platform, the debugging perspective of which provides several facilities for hardware debugging. We have full view of the object fields and method variables, and we can traverse the reference paths to access some other object's fields. Figure 8 illustrates this point. Here the instance fields of class Folder and the local variables of method hwthreadrun() are shown. Notice that we can access information from other HwThread objects, since Folder has a reference to its scheduler. In addition, by examining HwFifo objects, we can debug communication problems. The tool has also useful breakpoint mechanism. We can suspend and resume the execution at any time. We can also set breakpoints with hit count. When execution is paused for a breakpoint, we can do various actions. Firstly, we can resume up to the next breakpoint. The tool supports also step run in various forms (step into, step over, step return).

3. MODELING EXAMPLE

As an example we will present the modeling of a cache based stack folding technique for Java processors [11]. We will start with a brief description of the architecture, and then we will present some experimental results of a DSP application.

3.1 Architecture Description

The Java virtual machine instruction set is stack based, which causes serialization in the execution of Java programs. In order to exploit the inherent parallelism of Java programs we must eliminate stack accesses. This can be achieved by folding java bytecodes. For example an operation in the JVM

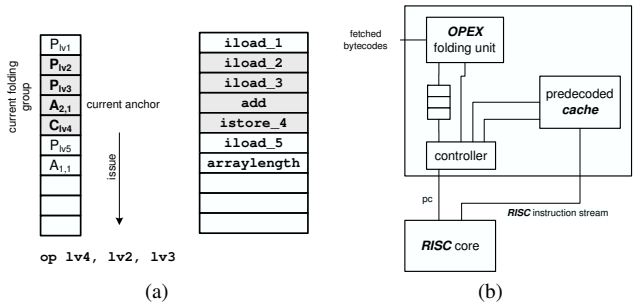

Figure 9: (a) Folding Operation, (b) Java2RISC Architecture

instruction set usually consists of a sequence of the form PUSH PUSH OP POP [6]. We have two loads in the stack, extraction of the two topmost words from the stack, computation of the result and then a pop from the stack and a store to a local variable. This sequence can be folded to just one RISC like instruction with the two local variables as sources, and the third as destination. The most straightforward approach is that of pattern based folding, in which the instruction buffer is examined for some specific folding patterns [6]. More complex algorithms such as OPEX [9] perform nested folding, as illustrated in figure 9a. OPEX categorizes instructions to producers, anchors and consumers. A producer is an instruction that pushes a local variable or a constant into the stack, while a consumer pops a value from the stack and stores it in a local variable. Anchor is an instruction that changes the order or the content of the stack by processing. Almost all processing java bytecodes fall into this category. The algorithm maintains a queue of recently fetched java bytecodes and tries to find the first anchor instruction. Then the corresponding folding group is extracted and issued as a RISC instruction.

Our architecture uses a modified version of OPEX, which allows storing of the folding results in a predecoded cache. By keeping the results in the cache, the instruction decoding throughput is increased substantially, since the most accesses will hit in the predecoded cache.

Figure 9b shows a diagram of the architecture. The RISC backend core requests RISC instructions based on the java bytecodes counter. If the requested block exists in the cache, it gets a line of up to 6 RISC instructions. Otherwise, the folder starts folding from the requested java bytecodes address. The folding results are returned to the processor core and are also stored in the cache for later use.

The line of RISC instructions must terminate if a control flow change instruction is encountered, such as invoke, return or branch. Besides the instructions, a line contains fields for the fall through address and the branch target (valid only for branches). The processor core consults these fields in order to make the next request.

3.2 Experimental Results

We ran our model with specjvm98 [4] _222_mpegaudio as input. Some of the stats gathered during the simulation run are presented here. Table 1 shows the number of Java bytecodes per cycle, the number of executed RISC instructions per cycle, the cache hit ratio and the percentage of cycles the folding unit is operating for various cache configurations. The simulations were run for 20 million cycles.

Table 1: Stats for various cache configurations

cache parameters	JIPC	RIPC	CHIT(%)	FLDPER(%)
size=2x4096	4.4981	3.1968	96.329	12.51
size=2x2048	4.4310	3.1401	93.765	14.51
size=1x2048	4.0606	2.9022	89.509	23.92
size=2x1024	3.4454	2.4819	90.173	52.50
size=4x256	3.5634	2.4864	69.067	55.03
JIPC: java bytecodes per cycle, RIPC: RISC instructions per cycle CHIT: cache hit ratio, FLDPER: % cycles folding unit operates				

The use of the predecoded cache has accelerated significantly the execution. The architecture performs well especially in kernels, where we have massive reuse of the folding results. By keeping the hit ratio in a logical range we can have speedup of 3-4 times over the pattern based folding.

4. CONCLUSION

The introduction of Java in microarhitecture modeling gives great productivity gains. Combining models of different levels of detail in the same environment is beneficial. We have discussed some methodologies for Java microarchitectures modeling and applied some of them in the powerful framework we developed on top of the Java language. In the future, we expect to improve our framework in terms of simulation speed, and to provide some utility classes for the computer architecture designer.

REFERENCES

[1] Eclipse open development platform. `http://www.eclipse.org/`.

[2] Jhdl. `http://www.jhdl.org/`.

[3] Simplescalar. `http://www.simplescalar.com/`.

[4] Spec jvm98 benchmark. `http://www.spec.org/osg/jvm98`.

[5] systemc. `http://www.systemc.org/`.

[6] *picojava-II Processor, Data Sheet*. Sun Microsystems, 1999.

[7] T. Austin, E. Larson, and D. Erst. Simplescalar: an infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[8] D. Black and J. Donovan. *SystemC: From the Ground Up*. Springer, 2005.

[9] M. El-Kharashi, F. Elguibaly, and K. Li. A robust stack folding approach for java processors: An operand extraction-based algorithm. *Journal of Systems Architecture*, 47(8):697–726, 2001.

[10] H. McGhan and J. O. Connor. picojava: A direct execution engine for java bytecode. *IEEE Computer*, 31(q0):ww–30, October 1998.

[11] I. Sideris, G. Economakos, and K. Pekmestzi. A cache based stack folding technique for high performance java processors. In *4th international workshop on Java technologies for real-time and embedded systems*, pages 48–57. ACM Press, 2006.