A Co-design Strategy for Embedded Java Applications Based on a Hardware Interface with Invocation Semantics*

Andrew Borg, Rui Gao and Neil Audsley
University of York, UK
{aborg,rgao,neil}@cs.york.ac.uk

ABSTRACT

As programmable hardware technology gathers momentum, the partitioning of applications into hardware and software will prove to be an increasingly important research area. Co-design technologies that achieve this partitioning typically adopt a strategy in which a high level specification is used to synthesise both hardware and software. This paper proposes an alternative approach by which equivalencies between hardware and software components are defined, thereby providing a common interface between them. This allows logic to be moved between hardware and software while retaining the functional properties of the application. An investigation is carried out to derive equivalencies between software elements of the Java language and hardware components by appropriate wrapping of the latter. By developing a framework that captures these equivalencies, this paper shows how hardware/software partitioning of a system can be relegated to a late stage of system development and include both application and virtual machine logic.

Keywords

Java, co-design, JavaMen

1. INTRODUCTION

The Java programming model has become established in mainstream software development as a platform for general-purpose applications. Today, Java is penetrating into more niche markets, including that of small embedded devices such as mobile phones. In particular, the Micro Edition of the Java 2 Platform (J2ME) provides an application development environment that specifically addresses the needs of embedded devices such as personal digital assistants and set-top boxes. Broadly speaking, the J2ME gives a subset of the full Java standard edition with cut-down functionality in the virtual machine and a smaller set of core and application libraries. As with standard Java (J2SE), the implementation of J2ME typically takes the form of an implementation of a virtual machine that runs as a process of a native operating system and interprets or JIT's Java bytecode. Alternative execution models include direct compilation to native code using compilers such as GCJ [1] and, more recently, hardware execution by means of Java co-processors and processors. Direct compilation overcomes the performance barrier of interpretation or JIT compilation in software virtual machines at the cost of cross-platform portability and a significantly larger code size. This latter overhead occurs because the runtime semantics must be embedded in application code. On the other hand, hardware execution preserves cross-platform portability of code while maintaining high performance, but at the cost of more expensive hardware.

As optimisation of software application and virtual machine logic reaches its peak, hardware support for these types of logic becomes necessary. Hardware support for virtual machine logic typically involves the execution of Java bytecodes in hardware and/or hardware support for higher level operations such as thread scheduling, class loading and garbage collection. Hardware support for application logic is not supported within a standard Java framework. This results in ad hoc solutions being provided for accessing available hardware with limited flexibility in specifying the hardware/software boundary.

This paper proposes a novel framework, called the *JavaMen* framework, that delivers two contributions: firstly, it provides a standard hardware interface for using hardware components in a Java environment; secondly, it allows the developer to decide at deployment whether to use hardware or software components to execute logic. Developing this framework requires an investigation into the equivalencies between the semantics of software elements (objects and methods in particular) and the semantics of hardware as well as an investigation into how these equivalencies can be achieved. A standard hardware interface is proposed that allows hardware developers to wrap their devices with the

This work is funded by the Next Wave Technologies Program (UK Government DTI) in conjunction with Sun Microsystems.

JTRES '06, October 11-13, 2006 Paris, France

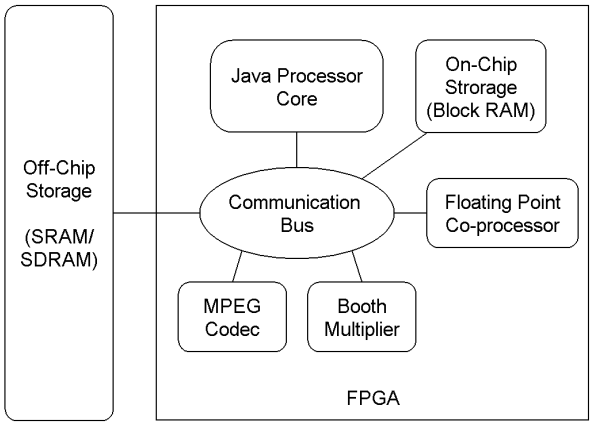

Figure 1: Hardware Architecture Targeted by the JavaMen Framework

appropriate logic required to interface with the JavaMen framework. Functionally equivalent software logic is also provided as regular Java code to achieve the co-design strategy.

The rest of this paper is set out as follow: Section 2 gives the background to this work and the contributions the JavaMen framework aims to deliver. Section 3 investigates two possible interfaces for hardware components within the JavaMen framework: a *method interface* and an *object interface*. Section 4 compares these two approaches within the context of a simple example, a hardware Booth multiplier. Section 5 introduces the API and semantics for hardware invocation in an implementation in which hardware components are wrapped within a method interface. An outline of a prototype implementation we have developed and a key component of this architecture called the "object manager" are described. Finally, Section 6 identifies future work and Section 7 concludes.

2. BACKGROUND

Programmable logic devices such as FPGAs provide an opportunity for application developers to offload computationally expensive execution to dedicated hardware that is relatively cheap and highly flexible. The architecture targeted by this work is one in which a Java processor is available as a softcore within an FPGA and in which the remaining real estate on the FPGA can be taken up by softcores specific to a particular application or domain. This arrangement is depicted in Figure 1. Examples of softcores may include encoders, decoders, signal processors and encryption and decryption cores. For a given application, the goal is to provide a framework by which softcores can be added to the FPGA as necessary to improve performance of the application through serial speedup or parallel execution. The JavaMen framework seeks to leverage the advantages of low cost programmable hardware by providing hardware support for Java at both the virtual machine and application levels within a co-design strategy.

2.1 Hardware Support for Java

The functionality of embedded devices is often highly specialised and, unlike their general-purpose counterparts, this often allows optimisations based on the application or domain particularities. Indeed, these optimisations are in some cases necessitated by the constrained nature of these devices. Specialisation can occur at two levels:

- The processor level: The specialisation of Java processor platforms can be based on an analysis of the application or domain. For example, LavaCore [2] provides a tool that can be used to identify the subset of used bytecodes, thereby eliminating the hardware structures or microcode of those bytecodes that are not used. FemtoJava [10] allows an analysis of the application to be carried out in order to fine-tune the hardware generated, thereby providing an application-specific hardware solution.

- The application/virtual machine level: Specialisation at the application level would typically involve the adoption of a mixed hardware/sofware solution of application logic and virtual machine logic that is driven by a co-design strategy. In this approach, the processor core and microcode set do not change but this logic is moved between software and hardware components external to the core.

Recent research into hardware support for Java generally targets virtual machine functionality and, primary, bytecode execution. There are two ways of improving bytecode execution using hardware. The first is to use a co-processor that works in conjunction with a general purpose CPU by either providing a fast bytecode translation to that CPU's native instruction set or by executing simple bytecode itself [3, 4]. The second option is to provide a Java-specific processor which is therefore limited, but specialised, to Java applications [5, 6, 12, 13]. In essence these latter architectures are similar; they implement a stack machine architecture with simpler bytecodes being executed in a small number of cycles and more complex ones executed in microcode or software. In addition to Java bytecode, hardware support for other virtual machine operations that are not defined at the bytecode level can also be provided. For example, the aJile Java processor [6] provides simple synchronisation and scheduling primitives in microcode.

Hardware support for low-level virtual machine logic (namely the bytecode interpreter) provides a solution that leverages the advantages of hardware execution without compromising Java's cross-platform capability. However, the highly specialised nature of embedded systems also motivates hardware support for application logic and higher level virtual machine logic. The absence of hardware concepts in Java means that abstraction is built into software in order to hide the complexities of low level hardware operation. This abstraction adds an overhead to execution that can be reduced if implemented in hardware.

2.2 Co-Design

Designing embedded systems often requires satisfying or optimising for a variety of constraints. Amongst others, these constraints include cost, power consumption and performance. The boundary of the hardware/software partition

plays a large part in defining these constrains. This boundary is often decided upon at the early stages of development and it is then hard to move this boundary at later stages. The motivation behind hardware/software co-design is to allow a unified development of these two parts of the system that delays the fixing of these partitions until a better understanding of the system emerges at later stages of development. Moving the boundary as further properties emerge is also simplified by the supporting tools. The co-design field has several niche research areas that range from finding suitable unified abstractions for specifying systems that can then be transparently mapped to either hardware or software to algorithms that automatically search the solution space of partitions for a particular partition that best satisfies a set of user-specified requirements and system constraints.

The methodology for co-designed applications generally involves a single specification of the application that is then decomposed into hardware and software parts. For example, SPARK [9] and SystemC [8] use a variant of a high-level language (such as C) in which the behavioural semantics of the application are described. Software and hardware synthesis is then performed on this specification and the software/hardware partition is thereby defined. The JavaMen approach adopts a different strategy to traditional co-design but with a similar goal of allowing delayed decision of the software/hardware partition. This strategy involves the specification of hardware interfaces that creates a layer of abstraction over application functionality and whether it is implemented in software or in hardware. The framework specifies this abstraction and defines how data and control flows are maintained across software and hardware. Since Java is an object-oriented language with control and data flow based on method invocation, the JavaMen framework specifies how this model is maintained at the hardware level through the specification of an invocation interface. This interface makes hardware components object-aware, meaning that they communicate with each other and software components in the same way as regular Java objects.

The virtual machine carries a significant amount of functionality such as support for the object model, memory management and multithreading that is external to the application. Limiting the solution space of a hardware/software partitioning to application logic may therefore result in a system in which the virtual machine's functionality becomes the constraining factor in a suitable performance/hardware-cost tradeoff. This problem can be addressed by breaking down the monolithic virtual machine into components that can cross the hardware/software partition in the same way as application logic within the proposed framework. Although a componentisation of the JVM is not investigated in this paper, the framework does not differentiate between virtual machine and application logic. Therefore, runtime functionality such as the scheduler or garbage collector can be implemented and interfaced in hardware in the same way as will be described for application functionality.

2.3 Summary of Contributions

The contributions in this paper address the two issues introduced in the previous two subsections, namely the absence of a standard hardware interface for adding hardware components to a Java-based system and the desire to choose hardware and software functionality within a co-design strategy. To this end, this paper provides two contributions. The first contribution of this paper is an investigation into two possible wrappings of hardware components the framework could provide in order to export object-oriented behaviour. These wrappings provide an interface between the hardware and software worlds that is absent in Java. Implementation considerations raised in providing these abstractions are also addressed. The second contribution of this work is to show how pairing of hardware components with software elements provide a co-design framework for embedded Java applications. The wrapping of hardware with object-oriented behaviour allows equivalences to be defined between the semantics of the Java language and the semantics of the interface of hardware components. This provides a framework in which hardware and software functionality can replace each other with minimal code modification.

3. ACHIEVING HARDWARE/SOFTWARE EQUIVALENCY

The approach taken to providing a standard hardware interface in JavaMen is to define an equivalency between hardware components and Java language components. This equivalency also provides a co-design solution as pairings of hardware and software allow migration of functionality between hardware and software. During development and testing, the application can use the software version of these components. If at deployment it is deemed necessary (for example for performance reasons) that the hardware version of that component is required, then this can be achieved without requiring any modification to the application. The first challenge is therefore to identify the Java-specific semantics in which hardware components are to be wrapped. Two possible solutions are to wrap hardware components as functional, stateless components, thereby allowing them to carry method-like[1] semantics or to allow these components to have state, thereby allowing them to carry object-like semantics.

3.1 Hardware Components as Methods

If all hardware components to be used were stateless, then they could be described as individual methods. In this solution, communication with hardware objects from software is carried out by *invocation* in a similar way that native methods are used to communicate with hardware in standard Java. Similarly, the communication protocol between hardware components is also based on invocation. However, hardware components are similar to software objects in that they encapsulate both state and function. Although the intuitive approach would therefore appear to be to define the equivalency between software objects and hardware at the object level, there are a number of key differences that must be considered. In particular, the encapsulation of function and state in Java through the object model is separated at runtime with static class information and the Java bytecode of methods held in the class information structures (such as the constant pool) and instance information (state) held on a per-object basis.

[1]Note that using the term "method-like" rather than "function-like" implies an association with object-oriented semantics. This is discussed further in Section 3.1.

The wrapping of hardware components as methods can be achieved by maintaining the same separation between function and state but with an augmentation of invocation semantics. Hardware components therefore take on the purely functional form of bytecode: they are themselves stateless but on invocation are sensitive to the state of the system. This state at invocation can be obtained from two sources. If the hardware component implements a static method, then the state is that of a corresponding class; if it implements a virtual method, then the state must be provided to reflect the associated object instance. In either case, invocation of hardware components is defined as a two-stage process. The state of the hardware must first be updated to reflect the context of the object or class instance and the invocation is then carried out. If no hardware component is available, equivalent software logic can be used instead to process the method request. Although these components can therefore be viewed as "functions", parameters passed at invocation are required to always include an object reference or class reference, hence augmenting the invocation process with object-oriented semantics.[2] Although the semantics of invocation are therefore those of Java, the framework could be implemented in other object-oriented languages as long parameters are passed by value. Note that, crucially, the equivalency defined here requires parameter passing at invocation to be the same as that in software. For this reason, no serialisation or deserialisation is allowed - *object references passed to a hardware component can be used by that component to invoke other hardware or software components within the same flow of control.* The decision as to whether the invocation is handled in hardware or software is defined by the invocation protocol described in Section 5.

3.2 Hardware Components as Objects

If a single hardware component defines a set of methods that are then encapsulated as a single instance of an object, then the state of the object might be stored in the hardware component itself. Similarly, a number of hardware components could be grouped to define a class for which an instance of that class is related to one of these groups of components. With this approach, the static functionality of the object and the state of the object are combined and there is a one-to-one mapping between an object instance and the hardware implementing the functionality of the methods. In contrast to the first solution above, a hardware component can only be used for a single instance of an object. For example, if two objects O1 and O2 of the same type have an integer instance variable (i:int) that is used in executing some method (m), and if a hardware implementation of (m) is available that makes use of the instance variable (i), then only one of O1 and O2 can use this hardware component. This approach therefore restricts the ability to use available hardware throughout the application in the same way that bytecode is separated from object instances. Nevertheless, this approach affords a more transparent model to the developer while still allowing control over whether a hardware component or its software equivalent are used at runtime. This is achieved through Java's type system as described in the forthcoming example.

[2]Note that in the current design of the JavaMen framework, class references cannot be passed. Hence all invocation is equivalent to virtual invocation.

Listing 1: The Calculator Class

```
2  public class Calculator {
       private int A;
4      private int B;

6      public Calculator() {
           this.A=0;
8          this.B=0;
       }
10
       public void setA(int a) {
12         this.A = a;
       }
14     public void setB(int b) {
           this.B = b;
16     }

18     public int add() {
           return this.A + this.B;;
20     }
       public int multiply() {
22         return this.A * this.B;
       }
24     // other methods
   }
```

4. COMPARING THE TWO APPROACHES

In order to demonstrate the differences between these two approaches from a developer's perspective, consider a class `Calculator` as described in Listing 1. Instances of this class have state in this case in the integer values A and B. On instantiation of a `Calculator` object, a space is allocated in the heap in order to save this state. On invoking *setA()* and *setB()*, a **putfield** bytecode together with an index into the constant pool is received in the bytecode stream with the stack containing a reference to the object instance and the integer parameter passed to the method. This is used by the runtime to find the location in memory where the given value is to be stored. On invoking *add()* (or *multiply()*), the values of A and B are retrieved from the heap through the **getfield** bytecode. The bytecode **iadd** (**imul**) is received in the bytecode stream and the multiplication (addition) is carried out by the runtime, the result of which is placed on the stack. This value becomes the element on the top of the stack on invocation return.

Consider that the implementation of these virtual machine semantics must now take advantage of an available hardware multiplier. A high-level view of the architecture of this system is illustrated in Figure 2. The constant pool information is stored in an area of memory accessible by the Java processor.[3] The heap is located in another area of memory which is initialised on booting. The VM runtime carries out logic such as scheduling and garbage collection that is implemented as standard Java code. The role of the multiplier is trivial in this case - on a multiplication occurring, the state of the calculator object must be passed to the multiplier unit and the result returned to the runtime to be placed as the top element on the stack. Consider the following program fragment:

```
{
    Calculator C = new Calculator();
    C.setA(4);
    C.setB(6);
```

[3]This architecture is similar to that of JOP [12, 13], the Java processor we are using in this research.

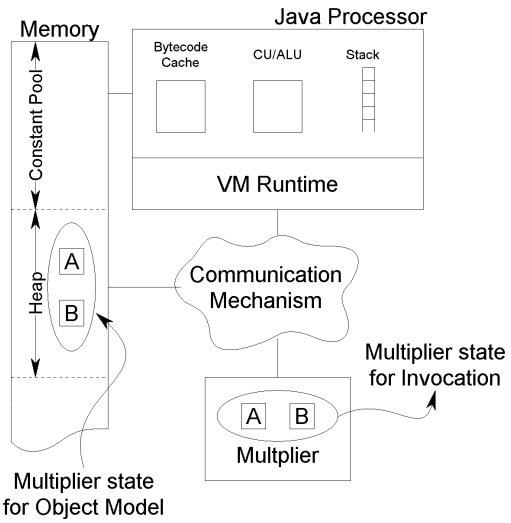

Figure 2: An Outline of the JavaMen Architecture

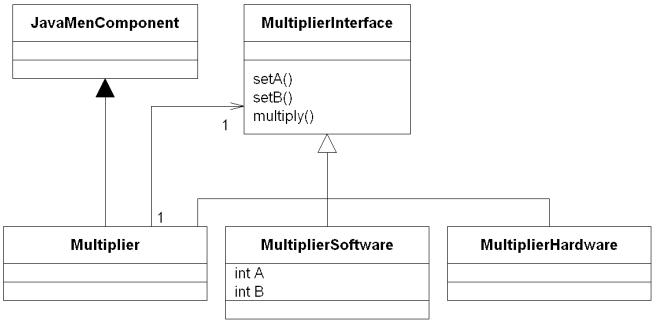

Figure 3: The Multiplier Example in a Model Defining an Equivalency Between Hardware and Objects

```
    System.out.println(C.multiply());
}
```

If the second of the two approaches listed above were adopted (§ 3.2), that is that the multiplier were considered to be an object, then the class structure of the application would have to be changed. In particular, the calculator object C would need to maintain a reference to an object of a new type (`Multiplier`) which has a state that is updated every time the state of C changes. Instantiation of a `Multiplier` object would return a virtual handle to either a hardware or software component. This transparency can be achieved by using Java's type system. In a simple prototype implementation we have developed, an interface `MultiplierInterface` defines the four methods of this class. Three classes are then defined that implement this class: `MultiplierSoftware`, `MultiplierHardware` and `Multiplier` (see Figure 3). This latter class acts as a delegate for passing invocations to either the software or hardware versions of the multiplier functionality. It also extends a class `JavaMenComponent` that identifies it as a JavaMen framework class and exposes specialised functionality (for example whether `this` instance is a hardware or software version). On instantiation of a `Multiplier` object, the JavaMen framework checks to see if a hardware

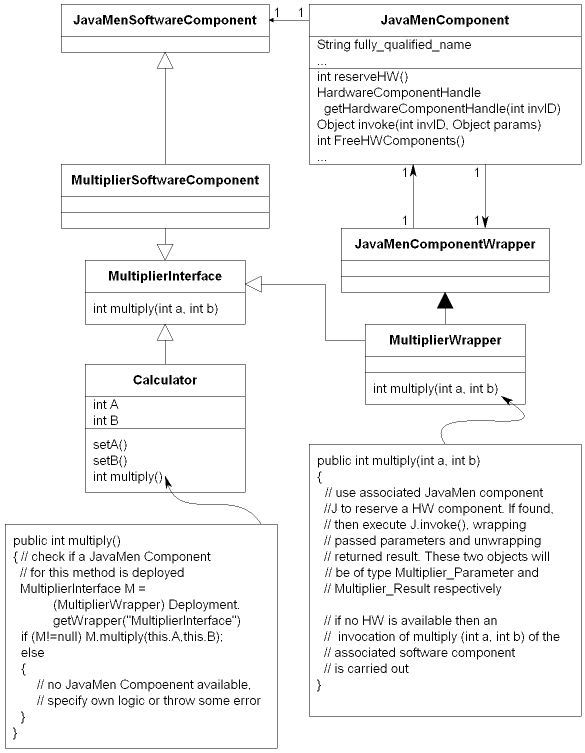

Figure 4: The Multiplier Example in a Model Defining an Equivalency Between Hardware and Methods

component is available. If it is, then a virtual handle to a `MultiplierHardware` instance is created and to which all invocations of the `Multiplier` instance are forwarded. The methods *setA()* and *setB()* therefore set the hardware registers of the multiplier with the appropriate value. Finally, the invocation of *multiply()* would start the multiplier and the result would be placed on the stack of the Java processor. The control and data flow of these operations would be handled by the JavaMen framework in a similar way to that described in Section 5. On the other hand, if no hardware component is available, a software instance of `MultiplierSoftware` is created and all invocations on the `Multiplier` instance are delegated here. In adopting this approach, a developer wishing to target this framework would therefore be required to deliver three classes: the software class, the hardware wrapper class and the interface that defines the equivalency between the former two classes. It is noted in passing that this approach requires consideration of non-availability of a hardware component at object instantiation as well as the "garbage collection" of these components when they are no longer referenced.

The first of the two approaches listed above (§ 3.1) is that adopted in our latest implementation of the JavaMen framework. Therefore, in this example and henceforth in this paper, the hardware component takes the form of a method for which the state of the instance C is stored in the heap. The instantiation of a `Calculator` object returns a handle to the representation of this object in the heap and the methods *setA()* and *setB()* modify this data. The developer of the `Calculator` class must provide the logic that attempts to find a suitable hardware/software pair deployed when *mul-*

tiply() in invoked. The JavaMen API is used to enquire as to whether such a pairing (with associated information on available hardware and equivalent software) is available. If it is, then the data values of the multiplication are handed over to this wrapper which will use the framework to decide whether a hardware multiplier is available at that time or whether the equivalent software will be executed. If no component of the requested type is available, then the `Calculator` class developer can provide custom logic for this method, including failure behaviour. Once again, this is a high-level description of this operation; the invocation protocol and low-level mechanism connecting the core to the hardware components is described further in Section 5.

The distribution of a JavaMen compatible hardware component in this case is composed of five classes. The first is an interface describing the method that the hardware/software JavaMen component implements (the `MultiplierInterface` class in this example).[4] All interfaces that describe this hardware/software wrapper always contain a single method definition. For example, `MultiplierInterface` contains only the method signature for *multiply(*`int` *a*,`int` *b)*. The second class in the distribution of a JavaMen compliant component is the wrapper class (the `MultiplierWrapper` class in this example) that implements this interface and provides the logic for interfacing with the JavaMen API to access JavaMen hardware/software component pairs. This class is essentially a stub that can be generated automatically based on the version of JavaMen framework it is to target.[5] This wrapper maintains a reference to an instance of `JavaMenComponent` which in turn contains a list of handles to hardware components (through instances of `HardwareComponentHandle` as described below) and a reference to an instance of `JavaMenSoftwareComponent`. The next two classes are wrapper classes for the parameters passed to the object component and the results returned. In this example, these classes are called `MultiplierParameters` `MultiplierResult` respectively. By defining types for these parameters, the framework allows validation of this data. The fifth and final class that must be provided by a distribution of a JavaMen component is the software logic in a class that implements `JavaMenSoftwareComponent` and the component's interface. In the multiplier example, the class `MultiplierSoftwareComponent` contains the software logic for the multiplier in the method *multiply()* which it is required to implement. The class diagram for the multiplier encapsulated as a JavaMen component is depicted in Figure 4. Note that the `Calculator` instance obtains (and may maintain) a reference to the singleton instance of `MultiplierWrapper`. It also contains the logic for the constructor and all other methods of the class, in particular *setA()* and *setB()* that modify the state of `Calculator` instances and passes this state to the multiplier only at the point of invocation.

[4]We have investigated the use of `java.lang.reflect.Method` to describe the interface but decided to avoid this approach in order to remove the requirement of reflection classes in the distribution.

[5]A similar stub generation mechanism to that used to generate RMI stubs could be used - currently these stubs are developed manually.

Finally, it is noted that defining an equivalency at the object level means that the invocation of hardware is implicitly distinguishable from that of a software invocation due to the different runtime type of the object to which the *Multiplier* class holds a reference. This time, however, there is no application-accessible class that encapsulates the hardware in a similar way to the `MultiplierHardware` class described above and that is therefore distinguishable from `MultiplierSoftware` by its runtime type. Instead, the handles to hardware components through `HardwareComponentHandle` are encapsulated in a `JavaMenComponent` and hidden from the developer. Defining an equivalency at the method level means that it may not be possible to determine at runtime whether a method invocation will be handled in hardware or in software. Although there are a number of ways in which either approach can be "forced" at runtime, the current invocation protocol of JavaMen components simply checks to see whether a hardware component for a method is available and uses its software equivalent only if it is not.

5. AN API AND SEMANTICS FOR HARDWARE INVOCATION

The JavaMen framework defines a two-level abstraction. The first level of this abstraction specifies the invocation semantics of JavaMen components. The second abstraction level defines the communication mechanism that connects the processor core to the hardware versions of these components. In this way, changes can be made to the underlying physical communication without changes being required to the hardware wrappers and while maintaining consistent invocation semantics across implementations. In practice, this means that developers of Java components do not need to create different wrappers for every possible communication mechanism.

5.1 Level 1: The Invocation Abstraction

The first level of abstraction is a Java-level abstraction that describes the invocation semantics of methods implemented in hardware. The `JavaMen` package provides classes that are used to specify the current deployment of the system (that is what JavaMen components are available) and to provide the functionality used by these components. This includes methods used by the framework to implement a simple invocation protocol that is independent of the physical connection between components and the Java runtime. The class `JavaMen.Deployment` describes the current deployment of the system. Although the class can be generated automatically, for example from an XML description of the deployment, it is currently created manually. The class `Deployment` maintains a static reference to an array of `JavaMenComponent` objects. An instance of `JavaMenComponent` consists of three fields: the first is a string identifying the interface describing the component as above, the second is an array of `HardwareComponentHandle` instances and the third is a singleton instance of `JavaMenSoftwareComponent` that is created at boot time. `HardwareComponentHandle` describes the second abstraction level as discussed further below; it is an interface that describes the functions that the implemented communication mechanism must implement. As an example, consider that two hardware multipliers are

available to be used by the application. In this case, the `Deployment` class indicates that two hardware multipliers exist, both of which implement the method specified in `MultiplierInterface`:

```
public class Deployment {
  public final static JavaMenComponent[]
    JavaMenComponents = new JavaMenComponent[]
    {
        new JavaMenComponent(
        ''calculatorpackage.MultiplierInterface'',
        new calculatorpackage.
            MultiplierSoftware(),
        new HardwareComponentHandle[] {
          new HardwareComponentHandle(1),
          new HardwareComponentHandle(2)
        })
    };
    //...
```

The class `Deployment` also maintains a reference to all wrapper classes which can be forwarded to applications through the method *getWrapper()*. As shown in Figure 4, this method takes a string parameter describing the fully qualified class name of the interface describing the required method. Since the wrapper class implements its associated interface, this method can be invoked by using the same signature. The invocation semantics of a JavaMen component are defined for the invocation of the single method of the wrapper class. Therefore, in the case of the multiplier, JavaMen invocation semantics are defined for the invocation of *multiply()* of the class `MultiplierWrapper`. These semantics have been kept as simple as possible at this stage. Briefly, the associated `JavaMenComponent` instance associated with this wrapper is queried using *HW_reserve()* in order to identify whether a hardware component is available. The returned integer encodes the hardware identifier of the reserved hardware component (if one was found) and a reservation number that must be passed with the subsequent invocation. In the event that a hardware component is available, a `HardwareComponentHandle` can be obtained using this integer to query the hardware and, ultimately, start the invocation. If no hardware component is available, then a handle to the software version of the `JavaMenComponent` instance is used to handle the invocation.

The JavaMen framework requires object references to also be passed as parameters, meaning that these object references are received by the hardware. As with a reference in standard Java, the value of this reference is implementation dependent. The hardware components can obtain the data associated with these objects by invoking methods of an "Object Manager" that forms part of the framework specification and is described in Section 5.3. The object manager is in effect a front end to memory that exports **new**, **putfield** and **getfield** bytecode functionality as well as other methods, thereby providing an interface to physical memory in a similar way to that provided by the Java language. In this way, the parameter passing semantics of software invocation (pass-a-reference-by-value for reference types and pass-by-value for primitive types) are maintained for hardware invocation. Currently, hardware handles cannot be passed between hardware components. This means that any invocation between hardware components must first pass through the software level. This will be addressed in future by providing equivalent hardware logic to that of JavaMen's invocation protocol currently implemented in software.

5.2 Level 2: The Communication Abstraction

Java does not provide any mechanism for direct access to hardware. Therefore, interfacing with hardware is usually achieved by using native methods that target a particular platform. In a hardware processor, accessing hardware is often achieved by ad-hoc techniques that would incur a significant burden at deployment if functionality is to be moved between software and hardware. For example, aJile [6] allows access to external hardware through user-defined extended bytecodes. This allows Java applications to access this hardware, but the interface is specific to each component and defined at a low level. JOP [12, 13] allows the logic for expensive bytecodes to be implemented in hardware or in Java. Although a common interface such as Wishbone [11] may connect hardware components, this interface is not exposed at the virtual machine level. In particular, this restricts parallelism and changes some Java semantics such as locking.

The JavaMen framework provides a communication mechanism between hardware and software through the invocation protocol described in Section 5.1. Since this protocol can be implemented by any physical communication, a second abstraction layer is introduced to allow pluggable communication mechanisms. This is achieved by separating the threading mechanism that deals with invocation from the logic that reads and writes to the hardware registers of JavaMen components in order to implement the invocation protocol. This second abstraction is defined at the Java level by a set of abstract classes and interfaces that are implemented by developers of different communication mechanisms and at the hardware level by defining a fixed wrapper for hardware components.

The Java class structure that describes this second abstraction level is given in Figure 5. The `Deployment` class maintains a reference to an instance of `CommunicationAbstraction` that is extended by the class that actually implements the communication protocol. For example, the class `WishBoneCommunication` is specified as the deployed runtime communication mechanism as follows:

```
public final static CommunicationAbstraction
        CA =WishBoneCommunication.getInstance();
```

`CommunicationAbstraction` is an interface that describes the methods that the physical communication must provide. In particular, `CommunicationAbstraction` requires subclasses to implement the *HW_reserve()* method that returns a handle to a physical hardware component. The runtime type of this handle in this case is `WishboneHardwareComponentHandle`, an instance of the superclass `HardwareComponentHandle`. The methods *setStateandStart()*, *checkResultReady()* and *ReadResult()* must also be implemented in `WishboneHardwareComponentHandle`. The logic in *invoke()* in `HardwareComponentHandle` uses the specified communication mechanism in the deployment (`WishboneHardwareComponentHandle` in this example) to forward these three method calls to the runtime instance of type *WishboneHardwareComponentHandle*.

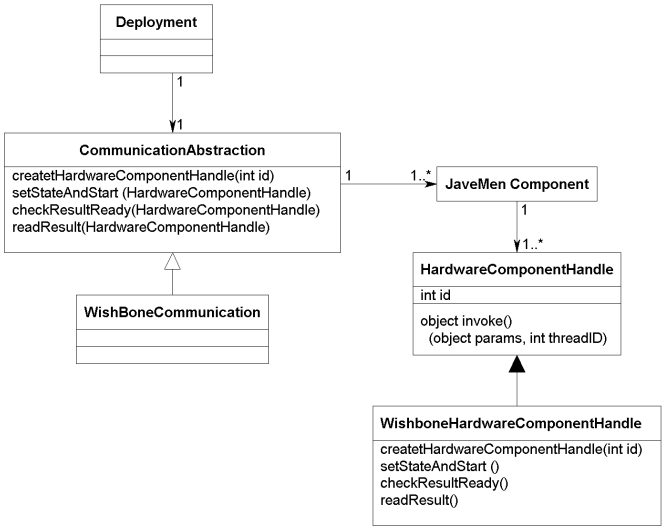

Figure 5: Realising Communication Abstraction through Wishbone

5.3 Prototype Implementation

In order to demonstrate the functionality of the framework, we implemented a Booth multiplier wrapped in the framework's method abstraction. As shown in Figure 6 the multiplier has two independent main components: a JavaMen hardware component wrapper and a standard Booth multiplier. The JavaMen hardware component wrapper is further divided into two parts: a bus protocol interface and an invocation/communication protocol interface. The bus protocol interface handles the bus-level data traffic, such as bus read, bus write, acknowledgment, etc. It uses technologies such as Finite State Machines (FSM) to translate incoming data stream parameters stored in registers, which are meaningful for the invocation/communication interface. In our implementation a Wishbone Bus is used to achieve the low-level communication. Hence a Wishbone bus controller is used here as the bus protocol interface. On top of the bus protocol, the invocation/communication interface implement the semantics of JavaMen invocation and communication. The modular design of the communication interface means that only minimal change is required when migrating to a new bus interface. The functional core here uses a standard Booth multiplier to efficiently carry out multiplications. Data and control signals between functional core and the hardware wrapper are exchanged via registers.

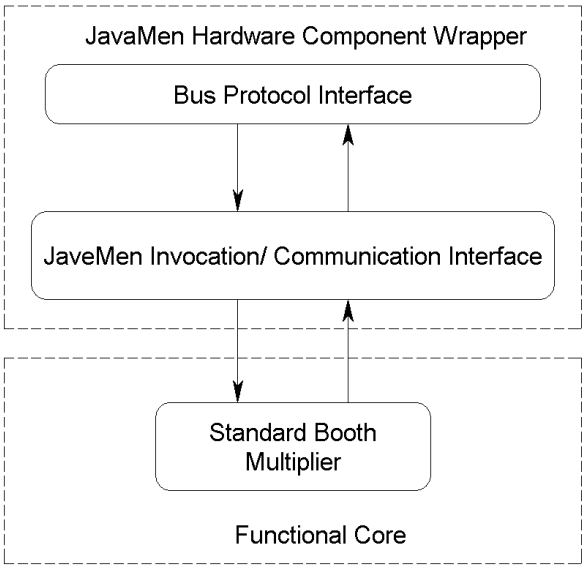

Figure 6: The Multiplier Architecture

Handling of a hardware invocation following determination of available hardware proceeds as follows. Initially, a virtual reference, which represents the location of an object that contains the data to be calculated, is written to the hardware wrapper from the Java processor. Subsequently, the wrapper fetches data from the program heap via an object manager, which resolves the reference. Once the calculation is complete, the wrapper creates a new object in the heap of the return type (in this case Multiplier_result) via the object manager and writes the result to the object. Finally, a virtual reference pointing to the object that contains the result is created polled for by the Java runtime.

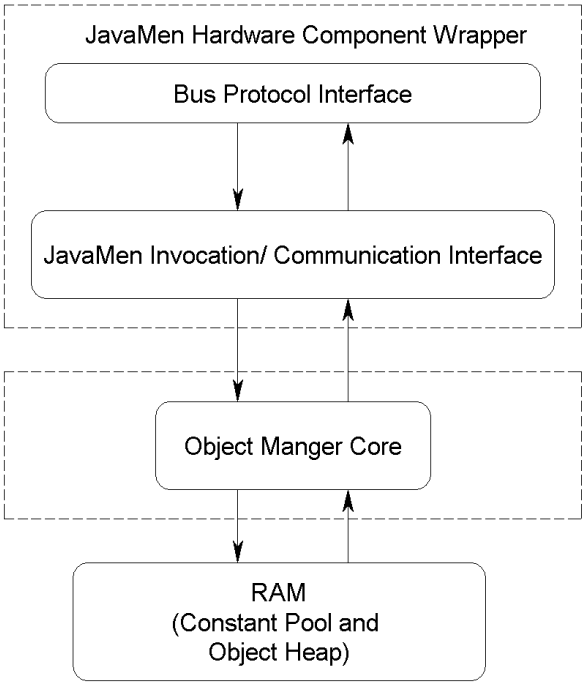

Figure 7: The Object Manager Architecture

A key goal of the JavaMen framework is to allow co-design of

the JVM as well as applications. The functionality of the object manager is an example of this as it defines logic in hardware that can also be implemented in software. Therefore, as with all JavaMen components, the object manager also has two parts: a hardware wrapper and the object manger core. Physically, the object manager core is made up of two parts: a front-end and a back-end. At the front end, the object manager provides both direct access [6] as well as indirect access through bytecodes such as **new**, **putfield** and **getfield** to access the heap as well as other calls from higher level virtual machine routines such as the garbage collector. This architecture is depicted in Figure 7. These operations are specified as JavaMen components and encoded into a sequence of logic operations and RAM accesses which are handled by the back-end to talk to the RAM module.

The hardware object manager does exactly the same job as an existing software method. For example, it initialises the program heap at boot time and manages the objects in the heap. Typically, when a request, is received (for example for the creation of a new object) the object manager will create a virtual reference for the object and access the program memory to obtain related information, such as size of the object, for the reference. Subsequently, the object manager will locate sufficient amount of memory for object in the heap. Finally, a reference of the object is returned to the device, which made the request.

6. FUTURE WORK

The JavaMen framework introduced in this paper is a novel approach to co-design as it introduces a common interface between hardware and software. Rather than moving from a high-level specification of a system and deriving a software/hardware partition through synthesis, our approach acknowledges that the integration of existing hardware (soft or hardcores) is an equally valid approach to achieving the benefits of codesign. The JavaMen framework attempts to simplify this integration by the specification of a common interface between the hardware and software partitions. However, forcing a Java interface onto all hardware requires an overhead in terms of hardware logic. Components that we have implemented which are more complex than the multiplier described int this paper are have proven relatively easy to implement. Another type of overhead involved is the temporal overhead incurred by the framework. The framework prototype is still being refined and exact cycle counts might prove to be over-optimistic at this stage. However, to give the reader a general idea of this overhead, it is noted that this overhead is minimal: On occasions in which the bus is not busy, the time required to carry out an invocation is of the order of tens of cycles. The granularity of the polling thread on a hardware's status is in fact the greatest overhead as this involves thread context switching. The checking of available hardware and the start of an invocation requires only a few low cost bytecodes in JOP and (currently) two method invocations. For the simplest of hardware components (such as a multiplier) this overhead may be comparable to a software invocation of a multiplication. It is believed that hardware support for any operations at least as complex as this can therefore benefit from hardware invocation as supported by the framework.

In the future development of the JavaMen framework, a key requirement will be to understand the implications of hardware invocation on Java's concurrency model. In particular, the effect on locking behaviour of crossing the hardware/-software boundary in the same control flow and the leveraging of parallel execution had to be considered. In Java, all invocation is considered synchronous. Blocking invocations (such as blocking on a socket read) are hidden from the Java developer with context switching typically being handled by the underling operating system. In JavaMen, the available hardware components motivate a model of asynchronous invocation whereby the core can continue executing other threads while a thread waits for the result of an invocation. Once a hardware handle has been obtained from *HW_reserve()*, the invocation of hardware is carried out in three phases: passing the parameters and starting the invocation, testing whether a result is available and finally retrieving the result. Each hardware component must provide input and output signals that provide this information. Our current implementation uses a simple polling mechanism in the invoking thread to check when a hardware invocation is ready. Since JavaMen requires that hardware also invoke software objects, two possible solutions could have been adopted: an event mechanism in which a different thread handles a software invocation from hardware (similar to an interrupt model) and a "distributed" thread model in which a thread ID is passed at each invocation so that the same thread handles an invocation from hardware to software if it is part of the same flow of control. The latter approach is that chosen for JavaMen as it had the advantage of maintaining locking semantics. For this reason, all invocations must also forward a thread ID which is carried across the hardware/software partition and which is used by the JavaMen runtime to identify which thread to use to handle an invocation. The result of an invocation on a software element from hardware is written to a specified address. Currently, software invocation from hardware is limited to specific methods associated with each JavaMen component. The polling thread therefore also checks whether the current hardware invocation is requesting the invocation of one of these methods in which case that thread is used to carry out the invocation. In a future version of the framework, references passed at invocation can be used for direct invocation from hardware.

There has recently been innovative research into the generation of hardware from software languages such as Ada [14]. An similar approach for Java has also been investigated [7]. JavaMen does not preclude the use of such techniques as the way that hardware is generated (be it from traditional hardware languages or higher level languages such as Java) is orthogonal to the framework. It is acknowledged however that these technologies can prove useful in quickly generating the hardware and software equivalencies of the JavaMen components and would therefore be a useful addition within the general co-design approach.

Finally, the equivalency described in this paper is based on functional equivalency. In essence, hardware components replace software component in order to provide speedup and

[6]This is mainly required in order to allow programs to be downloaded to the physical RAM at boot time. A second use of this feature would be to allow the implementation of RTSJ-like physical memory classes.

therefore an improvement in performance. However, the real-time properties of the system are changed in the framework. Providing a schedulability analysis tool that considers the migration across partitions of functionality is a future goal of this work.

7. CONCLUSION

This paper has introduced the JavaMen framework, a novel approach to developing Java applications for embedded systems based on the definition of functionally equivalent hardware and software pairings. These paired entities (JavaMen components) allow the decision of whether hard ware or software elements are ultimately used to be delayed until just before deployment. Two approaches to defining this equivalency were proposed and a method based equivalency was adopted. A simple semantics for invocation of these components was defined together with a specification of how these semantics are implemented across arbitrary communication protocols. Although the full invocation cycle has been successfully implemented and tested on a simple multiplier, more complex components are being implemented to evaluate our approach. The next phase of this research is to define a fully componentised JVM that will allow as much functionality as possible to be defined as hardware/software pairing in a JavaMen component. In this way, deployments may range from "high-software" deployments in which the central core specifies a minimum stack machine to "high-hardware" deployments in which almost all virtual machine and software logic can be implemented entirely in hardware. Finally, we acknowledge that the equivalency between software and hardware invocation is valid only insofar as no real-time requirements are specified. An investigation into the real-time implications of our strategy is left for future work.

8. REFERENCES

[1] The GNU Compiler for Java. Available at: http://gcc.gnu.org/java/.

[2] Product specification available at: www.xilinx.com/products/logicore/alliance/dsi/dsi_java_proc.pdf.

[3] Information and white papers available at: www.arm.com/products/solutions/Jazelle.html.

[4] Information available at: www.nazomi.com.

[5] picoJava Microprocessor Core Overview. Available at: www.sun.com/microelectronics/picoJava/overview.html.

[6] Real-Time Objects on the Bare Metal: An Efficient Hardware Realization of the Java Virtual Machine. In *ISORC '01: Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, page 53, Washington, DC, USA, 2001. IEEE Computer Society.

[7] P. Andersson and K. Kuchcinski. Java to Hardware Compilation for non Data Flow Applications. In *Proceedings of the 8th Euromicro Conference on Digital System Design (DSD '05)*, pages 330–337, Washington, DC, USA, 2005. IEEE Computer Society.

[8] F. Ferrandi, M. Rendine, and D. Sciuto. Functional Verification for SystemC Descriptions Using Constraint Solving. In *Proceedings of the conference on Design, automation and test in Europe (DATE '02)*, page 744, Paris, France, 2002. IEEE Computer Society.

[9] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations. In *Proceedings of the 16th International Conference on VLSI Design (VLSID '03)*, page 461, New Delhi, India, 2003. IEEE Computer Society.

[10] S. A. Ito, L. Carro, and R. P. Jacobi. Making Java Work for Microcontroller Applications. *IEEE Design and Test*, 18(5):100–110, 2001.

[11] OpenCores.org. *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, b.3 edition, September 2002.

[12] M. Schoeberl. JOP: A Java Optimized Processor. In Z. T. R. Meersman and D. Schmidt, editors, *On the Move to Meaningfull Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2003)*, volume 2889 of *Lecture Notes in Computer Science*, pages 346–359, Catania, Italy, November 2003. Springer.

[13] M. Schoeberl. Design and Implementation of an Efficient Stack Machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*, Denver, Colorado, USA, April 2005. IEEE.

[14] M. Ward and N. C. Audsley. Hardware Compilation of Sequential Ada. In *Proceedings of the 2001 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2001)*, pages 99–107, Atlanta, GA, USA, November 2001.