# Using Hardware Methods to Improve Time-predictable Performance in Real-time Java Systems

Jack Whitham
Real-Time Systems Group
Dept. of Computer Science
University of York, York
jack@cs.york.ac.uk

Neil Audsley
Real-Time Systems Group
Dept. of Computer Science
University of York, York
neil@cs.york.ac.uk

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

## ABSTRACT

This paper describes hardware methods, a lightweight and platform-independent scheme for linking real-time Java code to co-processors implemented using a hardware description language (HDL). Intended for use in embedded systems, hardware methods have similar semantics to the native methods used to interface Java code to legacy C/C++ software, but are also time-predictable, facilitating accurate worst-case execution time (WCET) analysis.

By reference to several examples, the paper demonstrates the applicability of hardware methods and shows that they can (1) reduce the WCET of embedded real-time Java, and (2) improve the quality of WCET estimates in the presence of infeasible paths.

## 1. INTRODUCTION

Within embedded real-time systems, applications need to make good use of minimal hardware and energy resources. A popular approach involves the use of *application-specific co-processors*, which carry out otherwise CPU-intensive tasks within dedicated hardware [2, 4, 5, 11, 31]. The hardware is able to carry out work in less time and using less energy than equivalent software on a CPU through specialization and use of low-level parallelism. Both stream processing [9] and control tasks [17] can be implemented in this way. Within real-time systems, another major advantage of application-specific hardware is time-predictability [5]. Computing the *worst-case execution time* (WCET) of a program is a difficult problem for typical CPU designs due to the complex interactions between the machine code and the hardware [14,20,27], but the specialization of co-processors leads to simpler timing behavior.

Java is an increasingly popular language for implementing embedded applications [15]. CPUs such as the Java Optimized Processor (JOP) [25] execute Java code directly without the overhead of interpretation or JIT compilation [19]. However, making use of application-specific co-processors from Java software is challenging, because Java hides the details of the surrounding hardware environment. Standard Java provides no way to represent a co-processor as a Java object, so programmers must use devices via an operating system or the *Java native interface* (JNI) [6]. Some implementations of JNI add significant overheads, and none are suitable for embedded platforms where Java *is* the native machine code [25]. Other Java extensions have been proposed to address this, but suffer from a lack of abstraction [13,29] or over-complexity [8,30]. These problems work against the need for portability (a principal goal of Java) and the need for time-predictability in embedded real-time systems.

This paper describes a lightweight, flexible and platform-independent scheme for linking Java code to co-processors implemented in a *hardware description language* (HDL) such as VHDL [3] or Verilog [32]. The scheme is evaluated using the WCETs that can be obtained by analysis of various programs implemented using software only, and using both software and co-processor hardware. The test environment is an embedded real-time system based on the JOP CPU implemented using a Xilinx ML401 field-programmable gate array (FPGA) prototyping board [35].

The structure of this paper is as follows. Section 2 presents related work, and Section 3 describes the JOP hardware environment used for evaluation. Section 4 gives an abstract description of hardware methods. Section 5 presents details of a concrete implementation of hardware methods for the JOP CPU, and describes how the WCET of both Java code and co-processor functionality can be determined in this environment. Section 6 evaluates hardware methods quantitatively using benchmark programs and qualitatively through a discussion. Section 7 concludes the paper.

## 2. RELATED WORK

Platform-independent interfaces between software (or software and hardware) are not a new idea. The Java Native Interface (JNI) links Java code to non-Java ("native") code. It can be used on all Java platforms that support JNI [6]. JNI's interfaces are primarily *method-oriented*: that is, the functionality of native code is accessed via method calls with the same semantics as regular Java method calls. These *native methods* are typically written in C or C++, and because the interface is method-oriented, it is easy to make use of them from Java.

A key issue is access to Java variables from native code. These may include the parameters that native methods are

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*JTRES'09* September 23-25, 2009 Madrid, Spain
Copyright 2009 ACM 978-1-60558-732-5/09/09 ...$10.00.

called with, return values that are produced, and any other variables that might be indirectly accessible (e.g., fields in objects passed to native methods). JNI handles this by providing an *application programming interface* (API) to the *Java virtual machine* (JVM) which allows *primitive* Java types (e.g., `int`, `short`) to be translated to native types (e.g., 32-bit words). Native method programmers must use this API for each Java variable access. However, this effort can be avoided through the use of a higher-level *interface generator* such as the *Simplified Wrapper and Interface Generator* (SWIG), which hides the complexity behind a layer of abstraction [7]. SWIG produces both Java and native code so that the native methods need no awareness of JNI.

Java interfaces to co-processor hardware can also make use of a method-oriented paradigm, but previous work has not always taken this approach. Direct access to memory-mapped registers is possible in Java via `RawMemoryAccess` objects, and this has been used to control hardware devices [13] but provides no way to map device functionality to Java primitives such as fields and methods. *Hardware Objects for Java* [29] extend this: each field within a *hardware object* shares a physical memory address with a co-processor register. Hardware objects have been implemented for the JOP CPU and two software JVMs (CACAO and SimpleRTJ). This interface is at a higher level than `RawMemoryAccess`. However, as with JNI and `RawMemoryAccess`, additional Java code needs to be generated to implement the method-oriented paradigm. This code must translate Java variables into a form that could be accessed by the hardware.

Accessing Java variables from the co-processor hardware is not a trivial problem when the variables are not primitive and when the variables are *vector* types (e.g., arrays). This occurs (1) because the physical format of those types is defined by the JVM [6], and (2) because the physical memory address of the objects may be changed by the JVM at any time (e.g., by garbage collection). JNI avoids the first issue by limiting the fields that can be directly accessed to primitives and arrays of primitives, and avoids the second issue by making copies of array objects as necessary [18]. These problems are created because the JVM is responsible for managing objects; their locations in memory and their structure. Two solutions have been explored in previous work:

(1) The object management functionality can be decoupled from the process of executing Java code so that co-processors can make use of it to directly access Java objects. Borg et al. implemented the *Object Manager* [8], a co-processor which controls memory allocation for all Java code running on JOP [25]. In this arrangement, co-processors and Java software share a common interface for any access to any object, and objects can be freely passed between hardware and software. However, this facility has a cost as it is not portable. Other JVMs would require completely different object managers to be written, since there is no standard for this functionality.

(2) Within *Java Hardware Threads*, all accesses to Java objects are routed through the software JVM [30]. Each co-processor is associated with a Java thread which acts as a *proxy*; this uses a channel called the *hardware thread interface* (HwTI) to communicate with the co-processor. This is bi-directional: Java code starts co-processor functionality by calling methods in the proxy thread object, and the co-processor accesses Java objects and methods by requesting them via the proxy thread. This offers a method-oriented protocol for co-processor access, and resolve the problem of accessing variables using a proxy thread. However, there is a performance overhead, because accesses must be serviced by software: they cannot go directly to memory. In contrast, JNI provides a way to copy variables into memory space controlled by the native methods, so the overhead is per-native call rather than per-access. Additional memory and performance overheads are created by the existence of a thread for each co-processor.

# 3. BACKGROUND

The Java processor JOP has been chosen as an initial implementation platform for hardware methods. JOP [25] is an implementation of the JVM in hardware with the main design constraint to be as time-predictable as possible. Most instructions, the bytecodes of the JVM, execute in constant time. Timing dependencies between instructions, which are common in pipelined processors, are completely avoided. Therefore, JOP is an easy target for WCET analysis. JOP is also a soft CPU, designed for implementation within an FPGA, so it is easily extended using the application-specific co-processors that provide the functionality of hardware methods.

As WCET analysis is simple for software on JOP, several tools have been developed to support JOP as the target processor. For the evaluation we use the tool WCA [28], which has been redesigned [16]. The WCET tool analysis the Java program at bytecode level, builds the *control flow graph* (CFG) of the program, transforms the CFP to constraints for an *integer linear programming* (ILP) problem, adds loop bounds to the constraints, and solves the ILP problem. An optional data-flow analysis can detect simple loop bounds. More complex bounds can be annotated in the Java source.

For access to main memory from JOP and from hardware methods, we leverage an arbiter designed for a *chip-multiprocessor* (CMP) version of JOP [22]. Three different memory arbiters are available in the JOP distribution: (1) a priority based arbiter, (2) a fair arbiter, and (3) a *time-division multiple access* (TDMA) arbiter. For a time-predictable system in general, only TDMA-based arbitration is analyzable. Knowing the TDMA schedule, WCET values for individual bytecodes can be derived. With these values WCET analysis of a CMP system is possible.

A co-processor which needs access to objects in the memory can be attached to the memory arbiter just like a CPU core. In the case of a TDMA arbiter the TDMA schedule has to be taken into account for WCET analysis of the hardware method. For a uni-processor system with co-processors the priority based arbiter is another option. During the execution of the hardware method, the CPU polls the status of the co-processor in a tight loop without accessing memory. In that case, it is more efficient to assign the whole memory bandwidth to the co-processor. With the priority based arbiter all cycles can be used by the co-processor.

```
COPROCESSOR mac_coprocessor
METHOD mac1
PARAMETER size int
PARAMETER alpha int[]
PARAMETER beta int[]
RETURN int
```

**Figure 1: Example of a hardware method description using an interface description language (IDL).**

```
public class mac_coprocessor {
    public static mac_coprocessor getInstance();
    public int mac1(int size, int[]alpha, int[]beta);
}
```

**Figure 2: Generated software interface for the MAC co-processor (Figure 1).**

```
entity mac_coprocessor_if is port (
clk                        : in std_logic;
reset                      : in std_logic;

method_mac1_param_size     : out vector(31 downto 0);
method_mac1_param_alpha    : out vector(23 downto 0);
method_mac1_param_beta     : out vector(23 downto 0);
method_mac1_return         : in vector(31 downto 0);
method_mac1_start          : out std_logic;
method_mac1_running        : in std_logic;

cc_out_data                : out vector(31 downto 0);
cc_out_wr                  : out std_logic;
cc_out_rdy                 : in std_logic;
cc_in_data                 : in vector(31 downto 0);
cc_in_wr                   : in std_logic;
cc_in_rdy                  : out std_logic );
end entity mac_coprocessor_if;
```

**Figure 3: Generated hardware interface for MAC.**

# 4. HARDWARE METHODS

In embedded real-time systems, resource overheads should generally be minimized. Therefore, the challenge addressed in this paper is the implementation of the *most important* features of previous work with minimal overhead. These are (1) method-oriented access to co-processor functionality, (2) parameter and return value passing and (3) JNI-like access to Java objects in memory. This section presents an abstract overview of hardware methods, which represent co-processor functionality in a manner that is analogous to the native methods implemented through JNI.

Section 4.1 describes the *interface description language* (IDL) that instructs an interface generator to generate appropriate Java and VHDL code. Section 4.2 gives an example architecture, and Section 4.3 discusses the sequence of operations used to activate a co-processor function in that architecture.

## 4.1 Interface Description Language

Figure 1 shows an example of a method description. The example represents a multiply-accumulate (MAC) unit, a device to speed up digital signal processing algorithms. The MAC co-processors acts on two arrays of numbers as shown in Figure 13.

The IDL code begins by declaring a co-processor (`mac_coprocessor`). A co-processor is a container of one or more hardware methods. The next line begins the definition of a method (`mac1`). Its parameters are specified (names and types) and the return type is also given. Note that the types are all primitives, or arrays of primitives - this restriction is necessary because only the JVM knows how to manipulate more complex objects.

This example shows only one co-processor containing one hardware method. The IDL code can describe more than one co-processor, and each co-processor can contain more than one method. Java itself could be used as the IDL, since introspection can be used to determine the methods defined by a class and the parameters that they use, but this would not provide any way to specify differences between Java types (e.g., `int`) and their hardware equivalents, which may have a narrower bit width. It also does not provide any way to specify additional information about each co-processor, such as WCET metadata.

From this IDL code, a Java class (the software interface) and VHDL component (the hardware interface) are generated. Given Figure 1, the Java class and VHDL component will have the external interfaces shown in Figures 2 and 3. The actual implementation of the class and the component are dependent on the platform.

Just as the Java side of the interface is embedded within a larger program, the VHDL component is also intended to be embedded within a co-processor. The co-processor may have additional links to the outside world, e.g., to the memory bus, but it communicates with the JVM via the generated VHDL component (e.g., `mac_coprocessor_if`).

## 4.2 Architecture Example

Figure 4 shows an overview of the link between Java and the co-processor as it appears within a JOP-based architecture. The components shown in this diagram can be divided into three groups.

(1) User-defined components (light gray). These describe the application as designed by the users of the architecture: programmers and designers. They are written in Java, IDL code, and HDL code. These components can be the same in any architecture that implements hardware methods.

(2) Generated components (white). These are automatically produced by the IDL code as described in the Appendix. They include Java code (Figure 2) and the interface for each co-processor (Figure 3). The generated code can be different for each architecture that implements hardware methods.

(3) Provided components (dark gray). These components are fixed parts of the architecture; neither generated nor user-defined. They include JOP and the *control channel interface* (CCI) hardware, which is a JOP-specific component used to communicate with the co-processors. This also includes the hardware objects abstraction [29], which can be used by the generated Java code to communicate with the CCI.

Figure 5 is a detailed view of the JOP implementation of hardware methods for the `mac_coprocessor` example given in Figures 1-3. Each arrow within this diagram is a group of wires defined within the HDL code. The co-processor interface (wires named `method_mac1`...) is the same on ev-

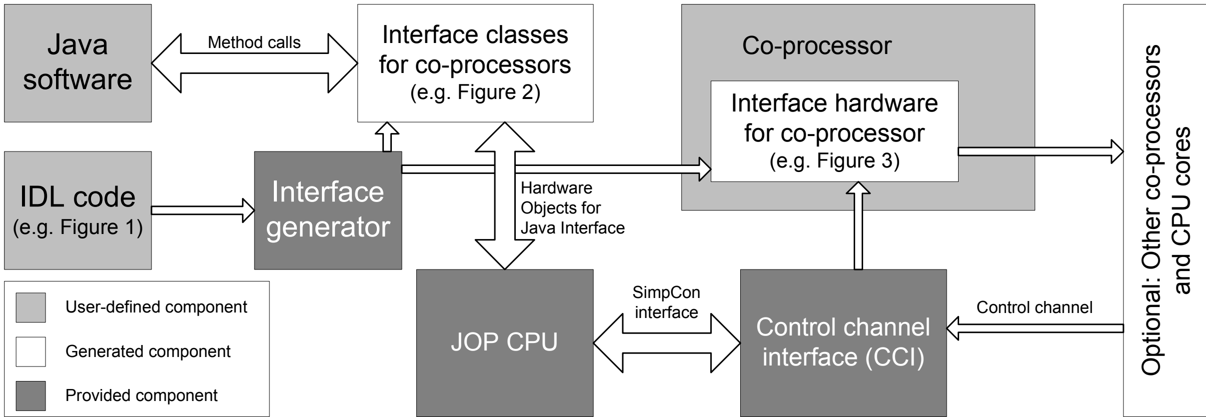


**Figure 4: Example system architecture showing both software and hardware components and illustrating the relationships between Java code, the JOP CPU and a co-processor.**

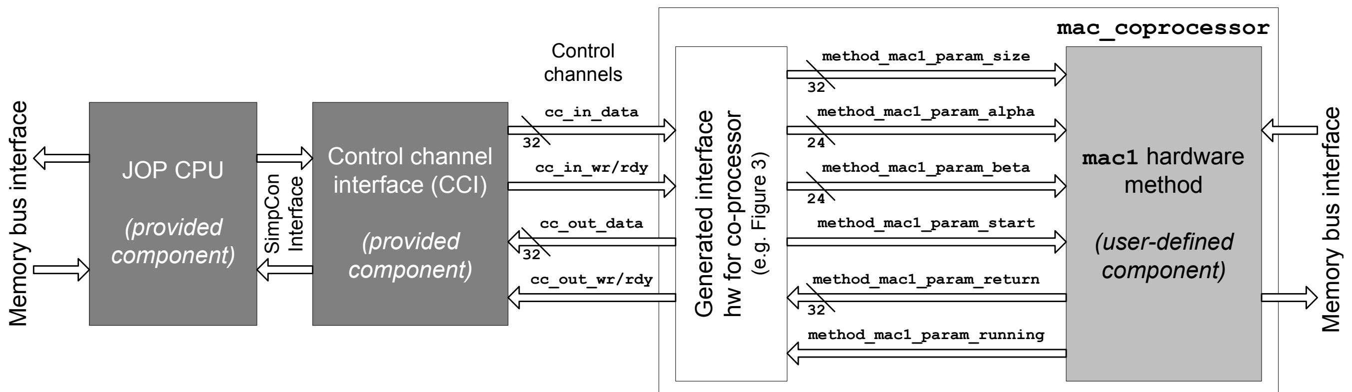


**Figure 5: Detailed view of the wiring needed to implement `mac_coprocessor` in hardware (Figures 1-3).**

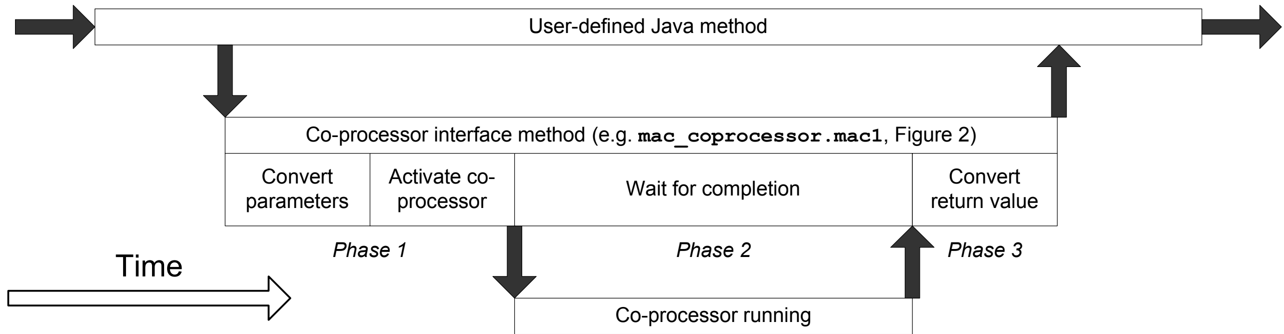


**Figure 6: Time line of events as co-processor functionality is activated.**

```
 byte  →  vector(7 downto 0)
short  →  vector(15 downto 0)
  int  →  vector(31 downto 0)
int[]  →  Pointer to base address of array:
          vector(23 downto 0)
```

**Figure 7: Translation of primitive Java variables into a form that can be used by co-processor hardware.**

ery platform, just as the external interface of the Java class (Figure 2) is the same on every supported JVM. The wires named `cc_in...` and `cc_out...` are control channels carrying messages to/from the co-processor. Messages are received via `cc_in...`; replies are sent to `cc_out...`. The CCI is responsible for sending and receiving these messages on behalf of the CPU.

## 4.3 Hardware Method Call

Figure 6 shows the sequence of events as co-processor functionality is activated. The first step is a method call from Java. Within this method (a software interface for the hardware method), parameters are translated into a form that can be used by the hardware (Figure 7). In this form, the data is transferred via the control channel (*Phase 1*). A start signal is activated (e.g., `method_mac1_start`, Figures 3 and 5). The Java method then blocks while the running signal (e.g., `method_mac1_running`) is asserted (*Phase 2*). Finally, the return value is copied from the co-processor, translated and passed to Java (*Phase 3*).

This arrangement reflects the semantics that a Java programmer would expect from a method, while allowing the co-processor to operate concurrently with Java code. This can be arranged through multithreading, or by changing the way that the running signal is asserted by a method. For example, it would be easy to implement "start" and "stop" methods within a co-processor: these would return immediately, but enable or disable some background process. This arrangement also means that there is no need for a proxy thread for each co-processor, minimizing resource usage.

# 5. WCET ANALYSIS FOR HW METHODS

In an embedded real-time system, time-predictable operation is as important as performance and correctness [27]. Java CPUs such as JOP facilitate WCET analysis by ensuring that every bytecode is executed in a predictable way (see Section 3). Hardware methods must provide the same feature. WCET analysis relies on concrete knowledge of both the application and the hardware architecture [24], so this section is based on an FPGA implementation of hardware methods using the JOP CPU.

Section 5.1 describes the necessary additions to the JOP architecture. Section 5.2 explains how the WCET of the Java code is captured, and Section 5.3 discusses methods to obtain the WCET of a co-processor function.

## 5.1 JOP-based Architecture

Within this implementation, the control channel is organized as a ring network (Figure 8), since this topology allows it to accommodate a large number of co-processors (or even CPUs, in a multiprocessor environment). Each device on the network (co-processor or CPU) forwards control messages onwards if they are not recognized.

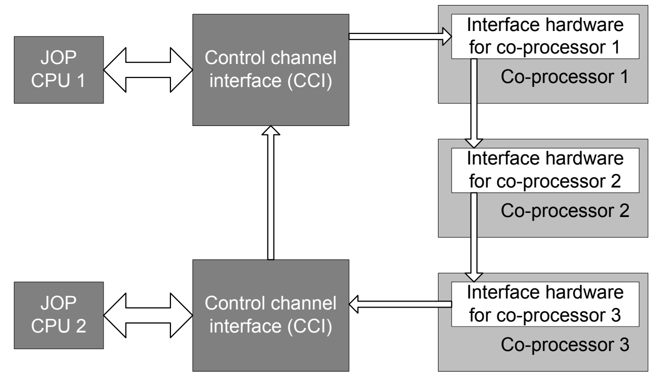


**Figure 8: The control channel acts as a ring network, able to accommodate large numbers of co-processors and CPUs. The protocol used on the network is lock-free so that there is no requirement for a global lock.**

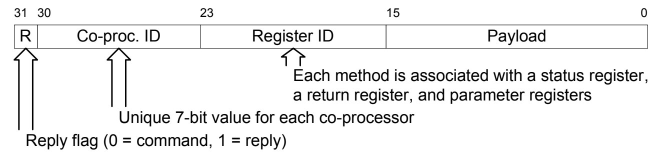


**Figure 9: The structure of a message.**

The control channel must support variable-length messages, since the number of parameters given to a hardware method is not fixed. It must also use a lock-free protocol, because (1) thread pre-emption could otherwise cause a deadlock, and (2) locks would otherwise need to be held across multiple CPUs in a CMP environment. An additional challenge is that atomic transactions (e.g., uninterruptable write-then-read) are not currently possible on JOP without breaking the requirements of the *Real-Time Specification for Java* (RTSJ), which does not permit disabling interrupts within a thread. The *control channel interface* (CCI) works around this problem and avoids any need for a global lock by tolerating pre-emption at any time.

Messages are 32-bit words with the structure shown in Figure 9. The CCI contains a single message register used for sending and receiving. Messages written to this register are guaranteed to traverse the ring network until they reach a co-processor with a matching identifier. The co-processor may generate a message in reply; this will have the same identifier, but the reply bit will be 1. Replies are captured by the CCI and stored in the message register until the software reads them. Figure 10 shows how this protocol is used for (a) write operations and (b) read operations. The behavior in the event of pre-emption is particularly interesting: the loop in the transaction subroutine *repeats* the read operation if the original reply is lost, as it would if (1) pre-emption occurred between the write and read operation, and (2) the pre-empting thread also used the CCI.

The Java code for a hardware method with $n$ parameters must send $2n + 1$ messages (two for each parameter, since payloads are 16-bits wide, plus one "start" message). Then,

```
// (a) write message to co-processor
public void _write(int message) {
    control_channel.data = message;
}

// (b) read message from co-processor
public int _read(int message) {
    reply_identifier = (message >> 16) | 0x8000;
    reply = 0;
    while (reply_identifier != (reply >> 16)) {
        // send request: control_channel.data
        // represents a hardware register; each
        // read/write access is a command for the CCI.
        control_channel.data = reply_identifier << 16;

        // receive a reply:
        reply = control_channel.data & 0x7fffffff;
        // "while" condition checks that we received
        // the correct reply for our request.
    }
    return reply;
}
```

**Figure 10: Java code to access co-processor registers via the CCI for (a) writes, and (b) reads.**

```
public void _wait_completed(int start_message) {
    reply_identifier = (start_message >> 16) | 0x8000;
    reply = 0;

    while ((( reply & 1 ) == 0 )
    || (reply_identifier != (reply >> 16))) {
        control_channel.data = reply_identifier << 16;
        reply = control_channel.data & 0x7fffffff;
    }
}
```

**Figure 11: Java code to poll for co-processor completion; completion is signaled when bit 1 of the status register is set.**

the Java code waits for completion, polling a co-processor register with repeated read operations (Figure 11). The number of iterations of this loop depends on the co-processor functionality. Following completion, further code reads back a return value (if any). This requires two read operations (Figure 10). Provided that the thread is not pre-empted, these two "while" loops will execute only once. Figure 12 shows the process used for the `mac1` method; this code is automatically generated from the IDL description.

The polling process is used for compatibility with the RTSJ, where CPUs are required to execute the highest-priority threads that are runnable. To avoid problems caused by lower-priority threads executing during hardware method execution, the hardware method is notionally "executed" on the CPU that calls it.

## 5.2 WCET Analysis of Java Code

Phases 1 and 3 of Figure 6 are easily analyzed by a WCET tool, such as the WCA software distributed with JOP [28]. This code is linear and executes in constant time. Even the code that obtains return values (in Phase 3) executes in constant time if there is no pre-emption, because pre-emption is the only effect that could cause the loss of a control channel message, so each read message loop executes only once. WCET analysis conventionally assumes that each thread ex-

```
public int mac1(int size, int[]alpha, int[]beta) {
    // convert parameters
    int_hw_size=size;
    int_hw_alpha=Native.rdMem(Native.toInt(alpha));
    int_hw_beta=Native.rdMem(Native.toInt(alpha));
    // load parameters
    _write(0x1010000|(((_hw_size)>>0)&0xffff));
    _write(0x1810000|(((_hw_size)>>16)&0xffff));
    _write(0x1020000|(((_hw_alpha)>>0)&0xffff));
    _write(0x1820000|(((_hw_alpha)>>16)&0xffff));
    _write(0x1030000|(((_hw_beta)>>0)&0xffff));
    _write(0x1830000|(((_hw_beta)>>16)&0xffff));
    // start
    _write(0x1000001);
    _wait_completed(0x1000000);
    // get result
    int _hw_ret = (_read(0x1840000) << 16)
                | _read(0x1040000);
    // convert result
    int _ret = _hw_ret;
    return _ret;
}
```

**Figure 12: Java code for hardware method `mac1`, showing the control channel message sequence.**

ecutes without pre-emption, since higher-level analysis accounts for this overhead [24]. The single iteration of the read operation loop is expressed by the loop bound annotation "`// @WCA loop=1`".

The only part of the Java code with a non-constant execution time is Phase 2: the completion loop (Figure 11). This requires another loop bound annotation, which must bound the number of iterations according to the worst-case completion time of the co-processor. Equation 1 can be used to calculate the upper bound. Let $i$ be the WCET of a single iteration of Figure 11, $c$ the WCET of the co-processor operation, and $b$ the upper bound on the number of iterations:

$$b = \lceil \frac{c}{i} \rceil \tag{1}$$

$i$ depends on JOP. It is a constant value because the completion loop can be implemented without any bytecodes that access external memory. On current JOP CPUs, $i = 49$. $c$ depends on the co-processor functionality, the control channel architecture and the memory subsystem. For example, a co-processor that requires $c = 1000$ clock cycles will cause Figure 11 to iterate at most $b = 21$ times. The loop bound annotation "`// @WCA loop<=21`" would be used.

## 5.3 WCET Analysis of Co-processors

The WCET $c$ is highly dependent on the co-processor function, and there is no standardized way to compute it. By analogy with a software WCET, it is the time required by the function to do a task, as measured from initiation (e.g. assertion of `method_mac1_start`) to completion (e.g. `method_mac1_running` is cleared). This time is distinct from the propagation delay through the co-processor logic gates, which determines the maximum frequency of the hardware, but provides no information about the number of clock cycles required to execute a function. Two options exist: (1) the WCET can be calculated using knowledge about the co-processor implementation and the surrounding architecture, or (2) the WCET can be determined by measurement.

The former approach is likely to be time-consuming, and must be repeated if the architecture is changed. The latter

approach would not be suitable for a general program because of the difficulty of determining the input conditions that maximize execution time; it is only possible in the special case of a *single-path program* [23], where the program behavior is the same for all input conditions. However, co-processors are similar to this special case. Their architecture can ensure that timing is either independent of the input data ($c$ is constant), or dependent only on the size of the input needing to be processed. An example of the first category would be a co-processor that acts on fixed-size data, such as *discrete cosine transform* (DCT) co-processors for image compression [33]. An example of the second category would be a co-processor that acts on a variable-sized vector of data, such as a symmetric encryption co-processor that encrypts a data buffer. For this case, $c$ is defined as:

$$c = k_1 + k_2 s \tag{2}$$

In equation 2, $k_1$ and $k_2$ are constant, and $s$ is the size of the input. Naturally, it is possible to create co-processors that do not exhibit this model of behavior: a hardware implementation of a non-linear-time algorithm such as Quicksort would qualify. These are outside the scope of this paper.

For a co-processor with a linear dependence on the size of the input $s$, a combination of the two approaches is likely to be most successful. Consider the WCET of a hardware method as $E(s)$ for $s$ input items. This is composed of two parts: (1) a software overhead $k_3$, used within phases 1 and 3 of Figure 6, and (2) the hardware execution time $b$ (equation 1). Hence:

$$E(s) = k_3 + b \tag{3}$$

Substituting the definition of $b$ (equation 1) and further substituting the definition of $c$ (equation 2) gives:

$$E(s) = k_3 + i\lceil \frac{k_1 + k_2 s}{i} \rceil \tag{4}$$

A combination of measurement and analysis gives $k_1$, $k_2$ and $k_3$. $k_2$ can be calculated by inspection of the internal state machine of the co-processor: how many clock cycles are required to process each data element? $k_1$ can be calculated by measuring the associated Java code and substituting the measured time $E(s)$ (for various $s$) into equation 4.

# 6. EVALUATION

In this section, the hardware methods scheme is evaluated using WCET analysis. The evaluation is a comparison of the WCETs that can be estimated for (1) software-only and (2) combined software and co-processor implementations of three different tasks. In each case, Java software is executed on the JOP CPU. The examples in this section have been chosen to illustrate cases where a hardware implementation can substantially improve performance (Section 6.1), time-predictability (Section 6.2), or both (Section 6.3). Additionally, Section 6.4 evaluates the process of implementing software functionality using hardware methods.

## 6.1 Array Operations

A multiply-accumulate (MAC) function acts on two arrays of numerical data as shown in Figure 13. WCET analysis of this function is trivial in both software and hardware form: the execution time has a linear dependence on the array size. In this case, the advantage of a co-processor implementation is a substantial reduction in WCET. A co-processor can pipeline the tasks of fetching input data from memory, multiplying data, and adding it to an accumulator.

```
public int mac(int size, int[]alpha, int[]beta) {
    int out = 0;
    for ( int i = 0 ; i < size ; i ++ )
    {
      out += alpha [ i ] * beta [ i ];
    }
    return out;
}
```

**Figure 13: Multiply-accumulate code.**

```
public int search_max(int size, int[]data) {
    int max = 0;
    for ( int i = 0 ; i < size ; i ++ )
    {
      int d = data [ i ];
      if ( d > max ) max = d;
    }
    return max;
}
```

**Figure 14: Search maximum code.**

This is demonstrated on JOP, where Figure 13 requires 730,334 clock cycles to process an array of size 10,000: 73 clock cycles per iteration, with a software overhead of 334 clock cycles. The same array requires 60,916 clock cycles when the hardware method implementation of `mac1` (Figures 1-3) is used instead; a twelve-fold reduction in WCET.

The co-processor behavior can be modeled by equation 4. In this model, the co-processor overhead $k_1 = 28$, the iteration cost $k_2 = 6$, and the software overhead $k_3 = 842$. The overhead of the hardware method is almost three times higher than that of the software implementation because of the time taken to send parameters and receive return values, but this cost is insignificant in relation to the time saved by the lower iteration cost. The WCET is reduced by the co-processor whenever the following inequality holds:

$$\begin{aligned} E(s) &< 73s + 334 \\ k_3 + i\lceil \frac{k_1 + k_2 s}{i} \rceil &< 73s + 334 \\ 842 + 49\lceil \frac{28 + 6s}{49} \rceil &< 73s + 334 \end{aligned} \tag{5}$$

This simplifies to $s > 8$; a MAC operation involving as few as nine items will have a lower WCET if implemented using a co-processor, because the time saved by the hardware method will make up for the additional overhead.

## 6.2 Infeasible Paths

WCET estimates can become inaccurate in the presence of *infeasible paths*, which are paths through the program that are possible according to the structure of the program, but not feasible when its inputs and semantics are taken into account [12].

Figure 14 gives a function to find the maximum integer within an array. In this function, an array that *never* triggers the condition `d > max` will lead to best-case timing, while an array that triggers the condition *on every iteration* (e.g., [1, 2, 3, ...]) will lead to worst-case timing. A

```
public int bit_count1(int size, int[]data) {
    int count = 0;
    for ( int i = 0 ; i < size ; i ++ ) {
        int d = data [ i ];
        for ( int j = 0 ; j < 32 ; j ++ ) {
            if (( d & 1 ) == 1 ) count ++;
            d = d >> 1;
        }
    }
    return count;
}
```

**Figure 15: Bitcount: naïve implementation.**

```
public int bit_count2(int size, int[]data) {
    int count = 0;
    for ( int i = 0 ; i < size ; i ++ ) {
        int d = data [ i ];
        for ( int j = 0 ; j < 4 ; j ++ ) {
            count += lut [ d & 255 ];
            d = d >> 8;
        }
    }
    return count;
}
```

**Figure 16: Bitcount: improved implementation.**

WCET tool must assume the second condition, leading to a WCET of 450,308 when the function is executed with a 10,000 element array on JOP. However, this execution path may be infeasible if certain properties of the array hold. For instance, if every element of the array is less than 10, then the condition can be `true` on at most 9 occasions (for values 1 through 9). In that case, the true WCET is 420,184, but it is not trivial for a WCET tool to make use of this semantic information, so the overestimate of 450,308 must be used. This is a trivial example of an infeasible path; the implications would be more severe if the loop contained more conditional operations, or if the loop was nested within another.

WCET analysis can attempt to detect infeasible paths [12] and some infeasible path information can be specified by annotations. These techniques do not cover all cases: for example, it is not possible to specify that every array element is less than 10. Hardware implementations provide a better solution because multiple paths can be executed in parallel so that there is no timing difference between `true` and `false` conditions. A hardware method implementation of Figure 14 has an execution time of 30,765 clock cycles for 10,000 elements, regardless of the values of those elements. In addition to a fourteen-fold reduction in WCET, the infeasibility problem is entirely solved. This approach could scale up to support complex conditional structures with no difficulty from higher levels of loop nesting.

## 6.3 Naturally Parallel Operations

Figure 15 gives a function that counts the number of non-zero bits in an array. This naïve implementation iterates through each bit of each word; the WCET for a 10,000 element array is 12,300,308. The execution time also varies according to the number of non-zero bits because of the `d & 1` condition, from a minimum of 1102 to a maximum of 1230 clock cycles per iteration.

In this case, a more efficient and predictable implementation is possible in Java (Figure 16). This uses a lookup table (`lut`) to obtain the number of non-zero bits in each byte. Here, the WCET for a 10,000 element array is 2,650,308: 265 clock cycles per iteration, independent of the array contents, with an overhead of 308 clock cycles per call. However, the improved implementation uses five memory accesses per iteration instead of one, which could significantly increase the cost of execution in some systems [34].

A hardware method would be a better choice, since the lookup table can be replaced by an adder tree capable of counting the number of non-zero bits in a single clock cycle. A hardware method implementation of `bitcount` has a WCET of 30,765 for 10,000 elements: a WCET reduction of over 86 times in comparison to Figure 16. This can be modeled by equation 4, with co-processor overhead $k_1 = 22$, iteration cost $k_2 = 3$, and software overhead $k_3 = 728$. The WCET is still reduced by the co-processor if the following inequality holds:

$$\begin{aligned} E(s) &< 265s + 308 \\ k_3 + i\lceil \frac{k_1 + k_2 s}{i} \rceil &< 265s + 308 \\ 728 + 49\lceil \frac{22 + 3s}{49} \rceil &< 265s + 308 \end{aligned} \quad (6)$$

This simplifies to $s > 1$. Although the overhead of the hardware method is higher than a pure software implementation, arrays holding more than one element are still processed with a lower WCET using the hardware method.

## 6.4 Discussion

The examples given in Sections 6.1-6.3 show that hardware methods permit embedded real-time programs to take advantage of the speed and time-predictable operation of hardware. The cost of accessing Java variables is minimal, being limited to a fixed "software overhead" (incurred by phases 1 and 3 of Figure 6). Memory accesses can be performed at the speed permitted by the FPGA hardware (3 clock cycles each in this environment [35]).

The annotation of the bound for the polling loop is not an ideal solution for WCET analysis. The user must determine the correct bound knowledge of the execution time of a single loop iteration and the execution time of the hardware method is needed. Both values depend on the actual implementation of the processor and the hardware method. To be independent of the processor implementation, a new annotation for the hardware method's execution time in clock cycles would be preferable.

Unfortunately, the quantitative comparisons between pure software and hardware method WCETs do not give the whole picture, because they provide no representation of the engineering process. Three points should be made about this. Firstly, the hardware methods for `mac`, `search_max` and `bit_count` can be used as plug-in replacements for the methods shown in Figures 13, 14 and 16, because they have exactly the same interface. That is, no special procedure is needed to call a hardware method.

Secondly, it is easy to add new hardware methods to a system, because the control channel can be expanded to include up to 127 co-processors just by adding new co-processors to the ring network (Figure 8). In this case, a JOP system was extended to first include `mac_coprocessor`, and

then extended further to include `bitcount_searchmax`, a co-processor implementing two hardware methods. Each extension amounted to (1) an insertion in the ring network, and (2) an increase in the number of ports provided by a memory arbiter.

Finally, co-processors are not necessarily difficult to implement, as they can be generated by user-friendly languages and tools such as Handel-C [1] and Simulink [21]. A hardware method with no array parameters does not access memory; this is particularly simple to create, because it acts only on the signals produced by the generated interface component (Figure 5). Co-processors that do access memory are only slightly more complex, and it is easy to extend a template (e.g., `mac_coprocessor`) with whatever functionality is required to implement a new hardware method (e.g., `bitcount`).

# 7. CONCLUSION

Hardware methods provide a straightforward, flexible and easy-to-use mechanism for extending embedded real-time Java code with co-processors. The evaluation has highlighted both the performance and time-predictability benefits of hardware implementations of Java functions. Hardware methods can be used to reduce the WCET of code, but also to improve the quality of WCET estimates by solving the infeasible path problem and making time-predictable use of the natural parallelism in code.

Hardware methods appear to be Java methods, so they can be easily used from any code. The process of extending a system with new hardware methods is simple, given VHDL or Verilog implementation skills. The process is facilitated by the use of an interface component generated from a platform-independent interface description language.

Currently, a second implementation of hardware methods is being written for FPGAs connected to a PCI Express interface. This will allow hardware methods to be used by Java software running on a conventional PC. The new implementation will use the IDL described in this paper, demonstrating that the paradigm is portable to other Java environments.

# Acknowledgment

This work has been supported by the JEOPARD project under grant agreement number 216682 which is funded by the European Commission Seventh Framework Programme.

# 8. REFERENCES


[1] Agility DS. Handel-C Language Reference Manual. `http://agilityds.com/literature/HandelC_Language_Reference_Manual.pdf`.

[2] J. Agron, W. Peck, E. Anderson, D. Andrews, E. Komp, R. Sass, F. Baijot, and J. Stevens. Run-Time Services for Hybrid CPU/FPGA Systems on Chip. In *Proc. RTSS*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[3] P. J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[4] N. Audsley and I. Bate. Synthesis of Legacy Real-Time Ada Software to FPGA. In *Proc. RTCSA*, pages 21–40, 2004.

[5] N. Audsley, I. Bate, and M. Ward. Mapping Concurrent Real-Time Software to FPGA. In *Proceedings of the 3rd U.K. ACM SIGDA Workshop on Electronic Design Automation*, 2003.

[6] C. Austin and M. Pawlan. JNI Technology. In *Advanced Programming for the Java 2 Platform*, pages 207–230, 2000. `http://java.sun.com/developer/Books/j2ee/advancedprogramming/jni.pdf`.

[7] D. M. Beazley. Swig: an easy to use tool for integrating scripting languages with c and c++. In *Proc. TCLTK*, page 15. USENIX Association, 1996.

[8] A. Borg, R. Gao, and N. Audsley. A co-design strategy for embedded java applications based on a hardware interface with invocation semantics. In *Proc. JTRES*, pages 58–67, 2006.

[9] A. Filippov. Building an Ogg Theora camera using an FPGA and embedded Linux (accessed 26 April 07). `http://www.linuxdevices.com/articles/AT3888835064.html`, 2002.

[10] Git. Version control system. `http://git-scm.com/`.

[11] F. Gruian, P. Andersson, K. Kuchcinski, and M. Schoeberl. Automatic generation of application-specific systems based on a micro-programmed java core. In *Proceedings of the 20th ACM Symposium on Applied Computing, Embedded Systems track*, Santa Fee, New Mexico, March 2005.

[12] J. Gustaffson, A. Ermedahl, and B. Lisper. Algorithms for infeasible path calculation. In *Proc. WCET*, pages 1–6, 2006.

[13] D. Hardin, M. Frerking, P. Wiley, and G. Bollella. Getting down and dirty: Device-level programming using the real-time specification for Java. In *Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, pages 457–464, 2002.

[14] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proc. IEEE*, 91(7):1038–1054, 2003.

[15] T. Henties, J. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, Mar. 2009.

[16] B. Huber. Worst-case execution time analysis for real-time Java. Master's thesis, Vienna University of Technology, Austria, 2009.

[17] M. Hubner and J. Becker. Exploiting dynamic and partial reconfiguration for FPGAs: toolflow, architecture and system integration. In *Proc. SBCCI*, pages 1–4, 2006.

[18] Ingo Proetel. Re: Use of JNI as part of a Java/co-processor interface. Email on JEOPARD mailing list, 30 May 2008.

[19] I. H. Kazi, H. H. Chen, B. Stanley, and D. J. Lilja. Techniques for obtaining high performance in Java programs. *ACM Comput. Surv.*, 32(3):213–240, 2000.

[20] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proc. RTSS*, page 12, 1999.

[21] Mathworks. Simulink - Simulation and Model-Based Design. http://www.mathworks.com/products/simulink/.

[22] C. Pitter. *Time-Predictable Java Chip-Multiprocessor*. PhD thesis, Vienna University of Technology, Austria, 2009.

[23] P. Puschner. Is worst-case execution-time analysis a non-problem? – towards new software and hardware architectures. In *Proc. WCET*, 2002.

[24] P. Puschner and A. Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Syst.*, 18(2-3):115–128, 2000.

[25] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[26] M. Schoeberl. *JOP Reference Handbook*. Number ISBN 978-1438239699. CreateSpace, 2009. Available at http://www.jopdesign.com/doc/handbook.pdf.

[27] M. Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.

[28] M. Schoeberl and R. Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press.

[29] M. Schoeberl, C. Thalinger, S. Korsholm, and A. Ravn. Hardware Objects for Java. In *Proc. ISORC*, pages 445–452, 2008.

[30] E. T. Silva, D. Andrews, C. E. Pereira, and F. R. Wagner. An Infrastructure for Hardware-Software Co-Design of Embedded Real-Time Java Applications. In *Proc. ISORC*, pages 273–280, 2008.

[31] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The Molen Polymorphic Processor. *IEEE Trans. Comput.*, pages 1363–1375, November 2004.

[32] Verilog.com. Verilog Resources. http://www.verilog.com/.

[33] G. K. Wallace. The JPEG still picture compression standard. *Commun. ACM*, 34(4):30–44, 1991.

[34] J. Whitham and N. Audsley. The Scratchpad Memory Management Unit for Microblaze: Implementation, Testing, and Case Study. Technical Report YCS-2009-439, University of York, 2009.

[35] Xilinx. Virtex-4 Family Overview. Datasheet DS112, Xilinx Corporation, 2007.

## Appendix

The described design is open-source under the GNU GPL. The source can be downloaded with *git* [10]:

```
git clone git://www.soc.tuwien.ac.at/jop.git
```

The build process of JOP and the accompanying tools is described in detail in Chapter 2 of the JOP Reference Handbook [26].

Hardware method support can be found within the `vhdl/jeopard` directory. Files ending in `.def` within this directory are processed as IDL code (e.g. Figure 1) by the `build.py` program, which generates both VHDL and Java code in appropriate places within the JOP source tree. A uni-processor or CMP build for the Xilinx ML401 board can be carried out using the `build` program in `xilinx/ml401a`; the number of CPUs can be set in `vhdl/top/jop_ml401cmp.vhd` and the arbiter type can be selected in `xilinx/ml401a/ml401.prj`. The Quartus project `jpd_cyc12` contains a configuration of a single JOP with two MAC hardware methods for an Altera Cyclone EP1C12 based board (`dspio`).