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

Jack Whitham, Neil Audsley, **Martin Schoeberl**

University of York, Technical University of Vienna

Hardware Methods

- Lightweight, Java-friendly co-processors.
- A *hardware method* replaces *software* functionality with application-specific co-processor *hardware*.
- Benefits:
  - Higher performance
  - Time-predictable operation
  - Energy savings

Implementations

- Hardware methods have been implemented for JOP.
  - The JOP CPU is a WCET-friendly platform, good for demonstrating time-predictability advantages of co-processors.
  - The JOP CPU and the co-processors exist in the same FPGA.

- A second implementation of hardware methods for PC hardware is currently being developed.
  - Co-processors are implemented on a PCI Express FPGA card.

Co-processors and Java (1)

- Java isn't designed for direct hardware access, but it is possible, e.g. using:
  - `RawMemoryAccess` [13]
  - Hardware Objects for Java [29]

- These approaches allow memory-mapped registers to be read and written.

- This is a low-level interface that breaks Java abstractions such as "objects" and "methods".

Co-processors and Java (2)

- A Java co-processor interface should be more like the Java Native Interface (JNI).
  - It should *hide the low-level details* of software to hardware communication.
    - This helps with code maintenance, portability and reuse.
  - The interface should preserve Java abstractions as far as possible (methods, objects, variables…)
    - This makes the interface easy to use.
    - Just call a method to make use of a co-processor.

Issues

- How is the *data within an object* shared between hardware and software?
- How is the *structure of an object* shared between hardware and software?
- Should a co-processor be able to call software methods?

How is the *data within an object* shared between hardware and software?

- Most co-processors act on vectors, not scalar data; this needs to be shared between producer and consumer.
- Options include:
  - A single memory space is *shared* by both co-processors and CPUs.
  - The CPU memory space is accessed by the co-processors via a *bridge*.
  - Objects are *copied* to scratchpad memory local to each co-processor during setup.
- The JOP implementation of hardware methods uses a single memory space.

How is the *structure of an object* shared between hardware and software?

- In Java, the memory layout and location of an object is defined by the JVM.
- Options include:
  - Moving the JVM’s object management functionality into a co-processor, so that both hardware and software have a single point of reference [8].
  - Using JNI to translate objects into a format accessible from C, since the layout of C structures *is* well-defined [6].
  - Route all memory accesses via the JVM [30].

- The JOP implementation of hardware methods uses special bytecodes to determine the memory locations of objects.

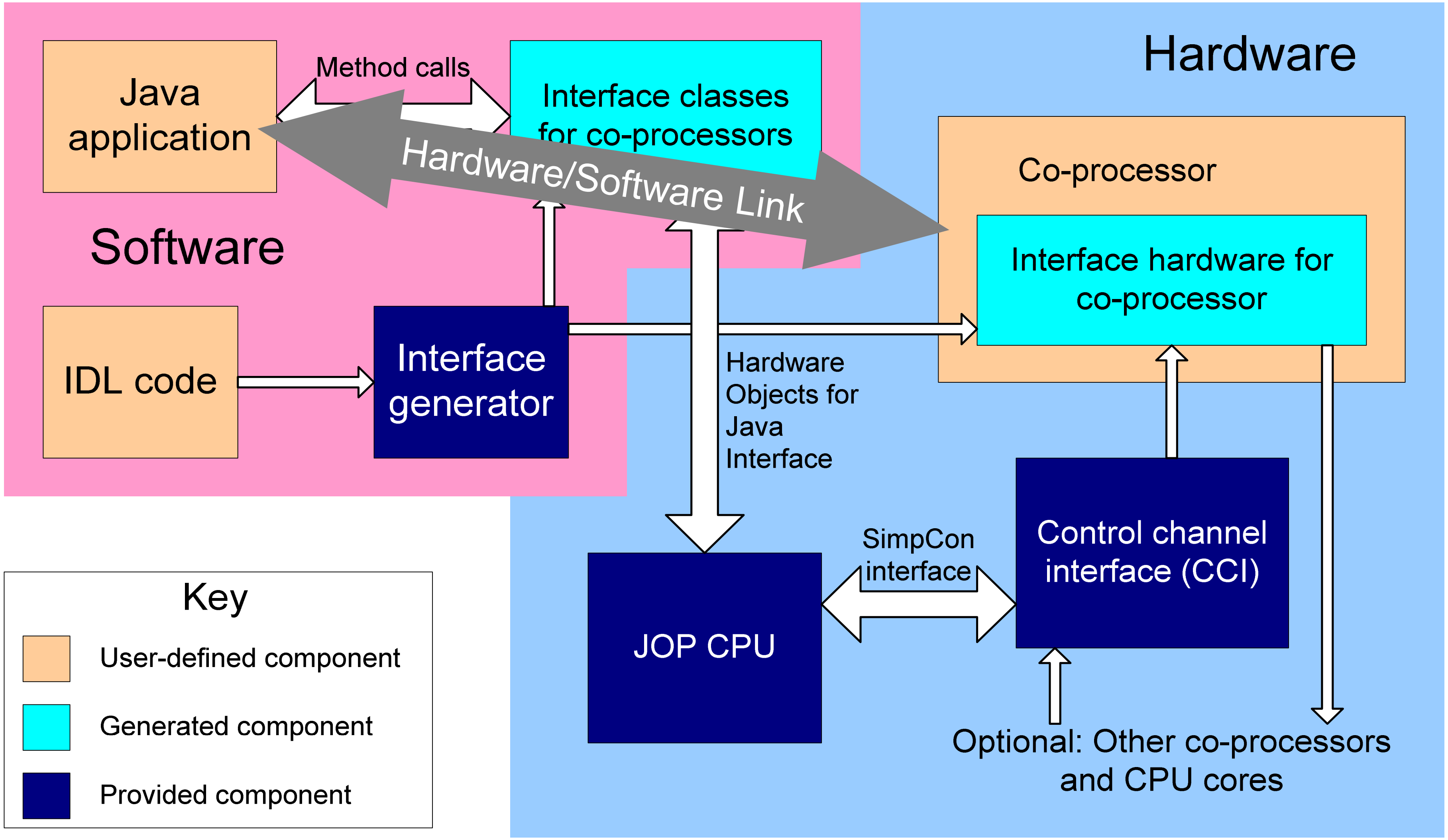
Should a co-processor be able to call software methods?

- This would be a powerful mechanism for sending data and messages between a co-processor and software.
- Implications:
  - The JVM must wait for messages from the co-processor, other than “completion”.
  - Co-processors need to be able to act as “masters” and cannot be simple reactive components.

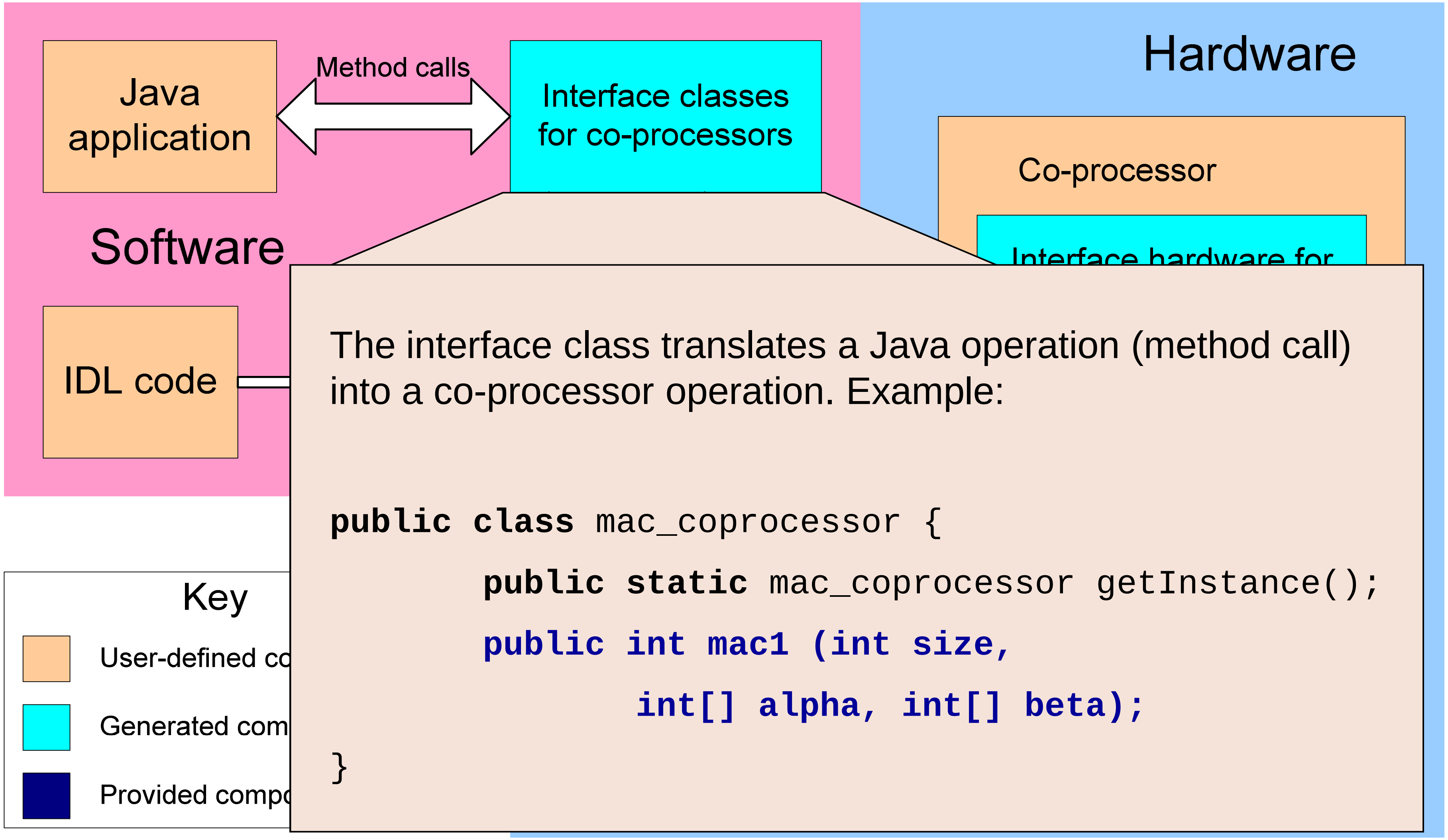
- The “hardware thread interface” mechanism uses a proxy thread for this purpose [30].
  - However, we are unconvinced that the extra complexity is worthwhile.

- The JOP implementation omits this functionality.

Hardware Methods for JOP (1)



Hardware Methods for JOP (2)



The interface class translates a Java operation (method call) into a co-processor operation. Example:

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

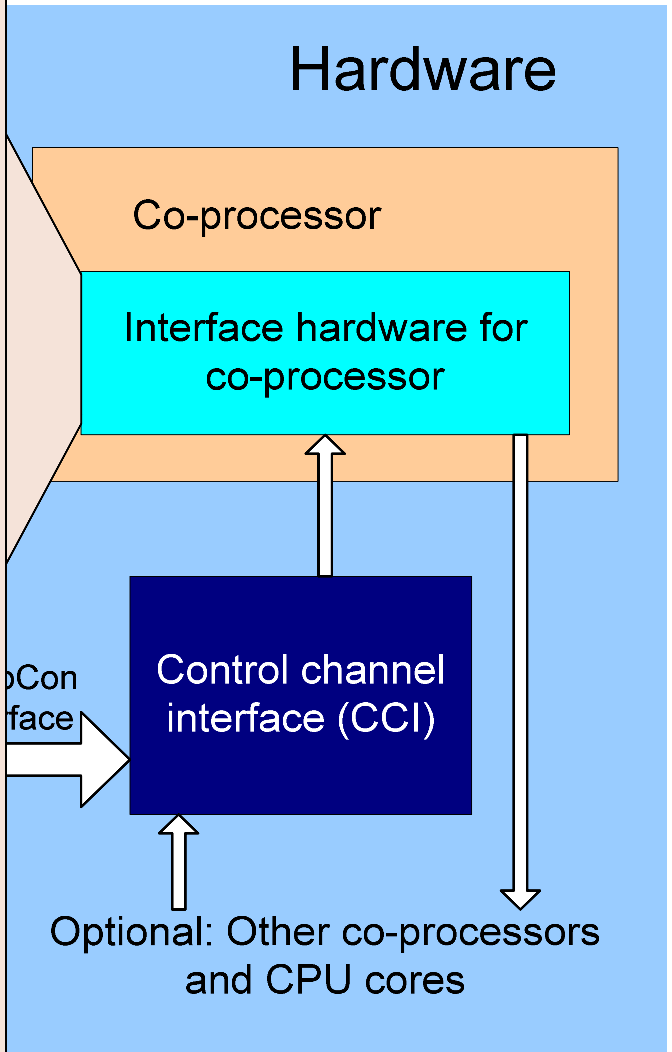
Hardware Methods for JOP (3)

The interface hardware tells the co-processor what to do, via a series of VHDL/Verilog wires. The wire values are derived from the parameters given to the method. Example:

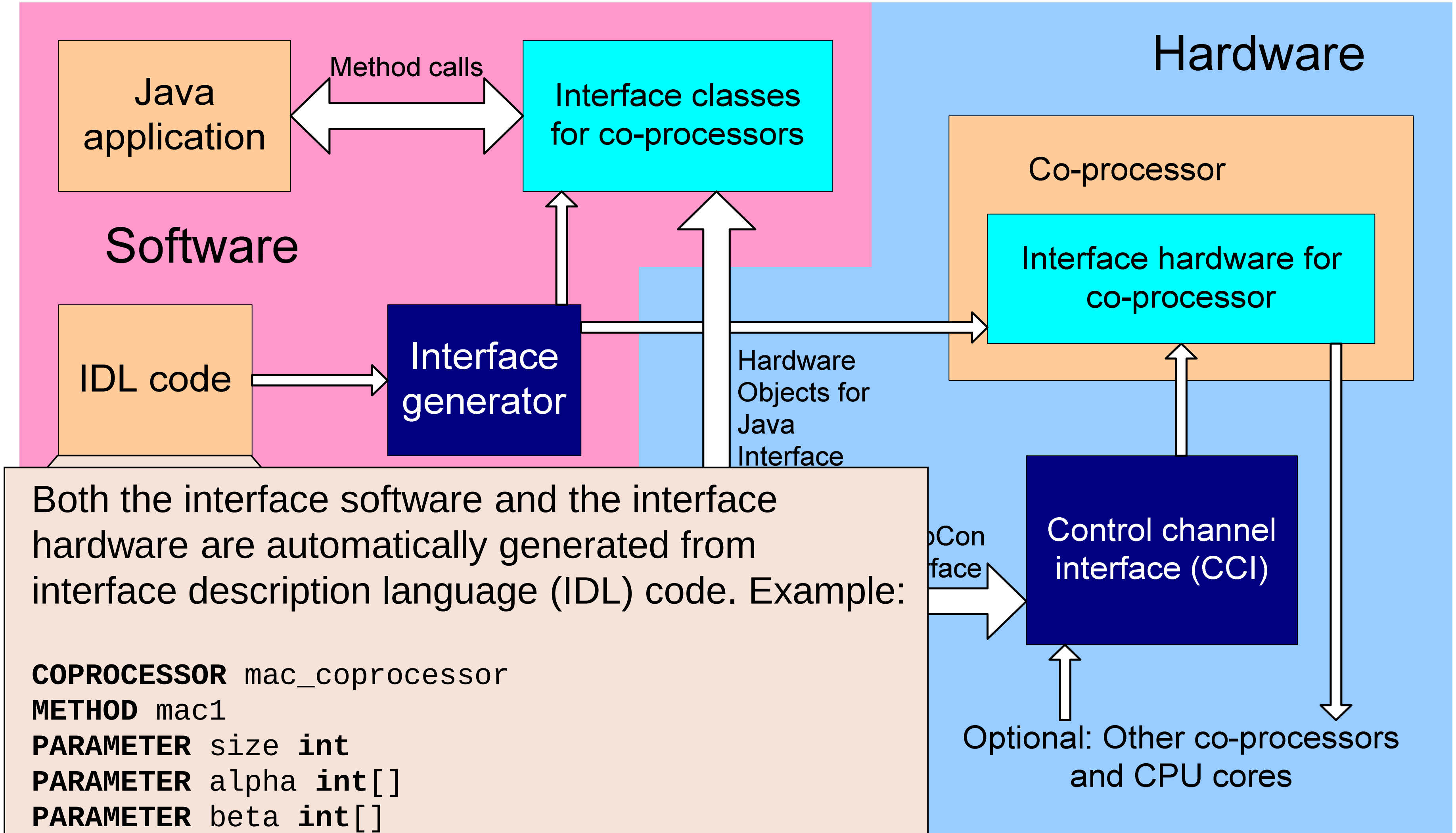
```
entity mac_coprocessor_if is port (
clk                        : in std_logic;
reset                      : in std_logic;

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;
```



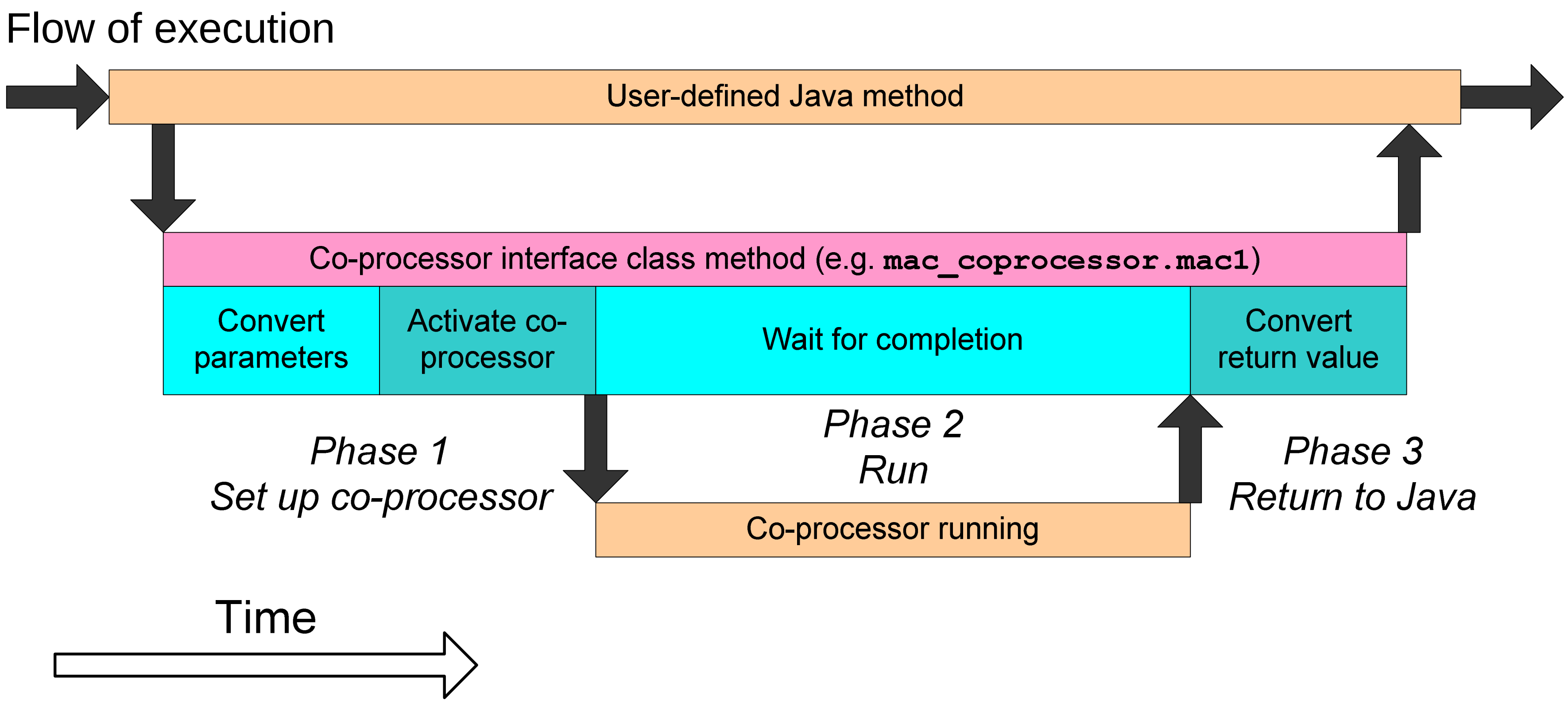
Hardware Methods for JOP (4)



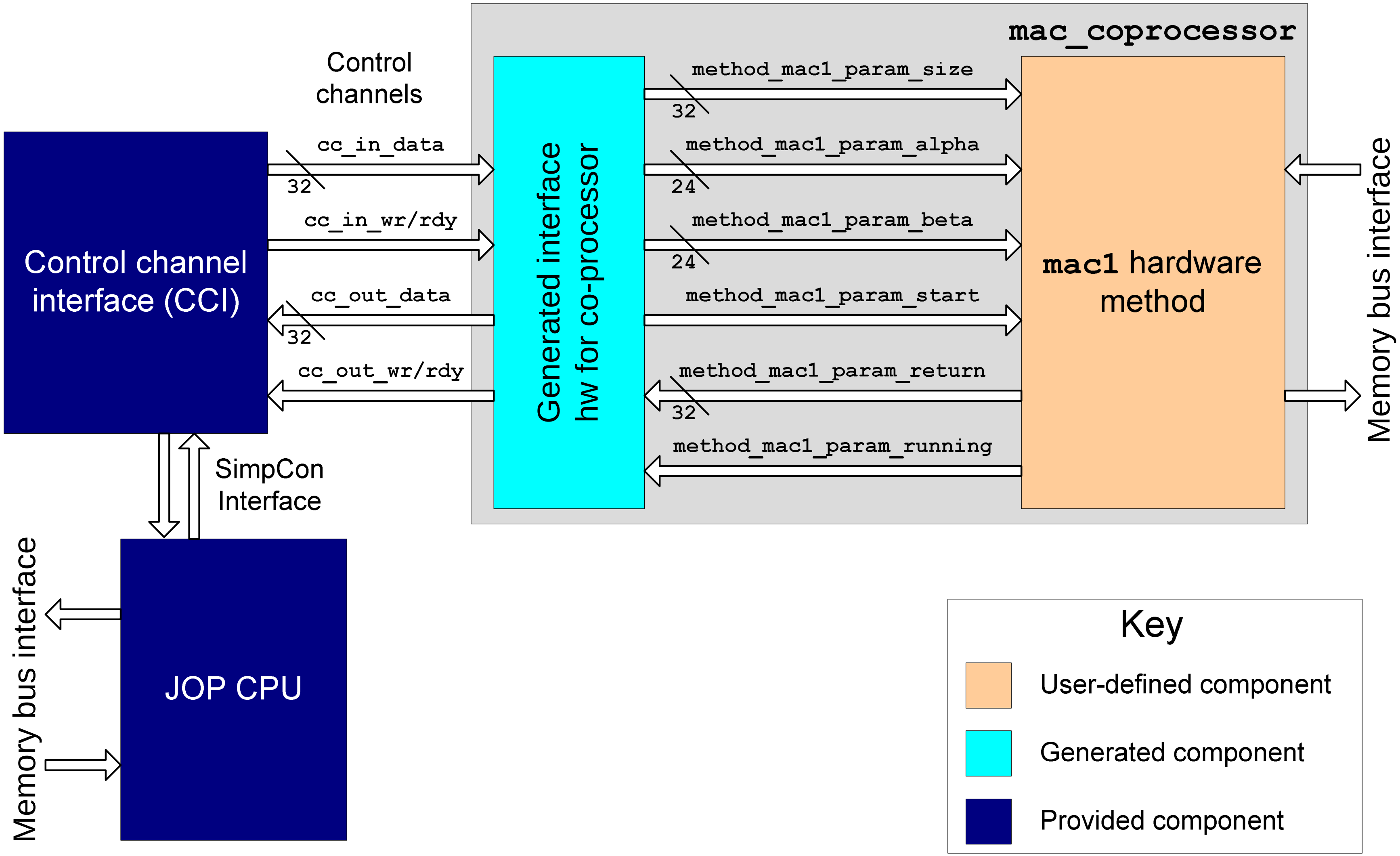
Both the interface software and the interface hardware are automatically generated from interface description language (IDL) code. Example:

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

Calling a hardware method



Implementing a hardware method



Features

- Details of the hardware/software interface are hidden by the interface generator.

- The user only needs to:
  - Specify the interface using IDL code.
  - Write a co-processor that receives parameters (as VHDL/Verilog signals).

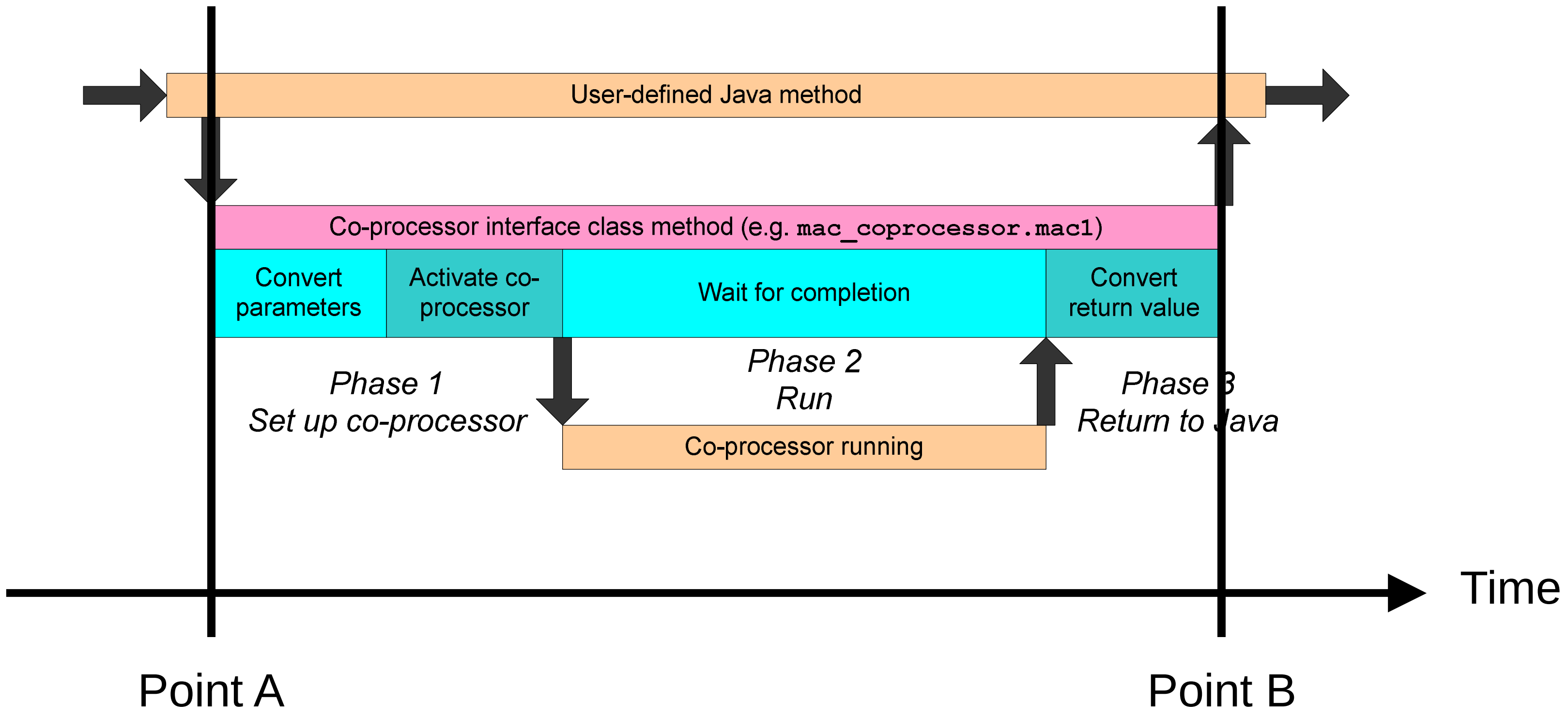
- Using a co-processor is as simple as it could possibly be.

WCET Analysis for Hardware Methods (1)

- WCET = worst case execution time
  - Maximum possible execution time for a program.
  - JOP includes the WCA tool, which computes a *safe* and *tight* WCET estimate.

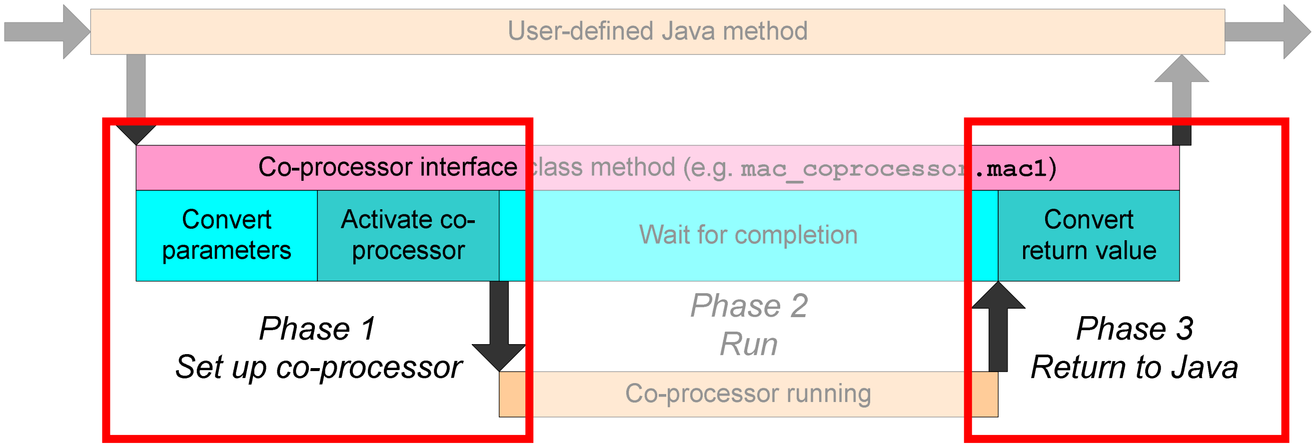
- In software, improved performance often comes at the cost of time-predictability.
  - e.g. Less accurate WCET estimates, *or* reduced average execution time, but increased WCET.
  - This does not apply to co-processors!

WCET Analysis for Hardware Methods (2)



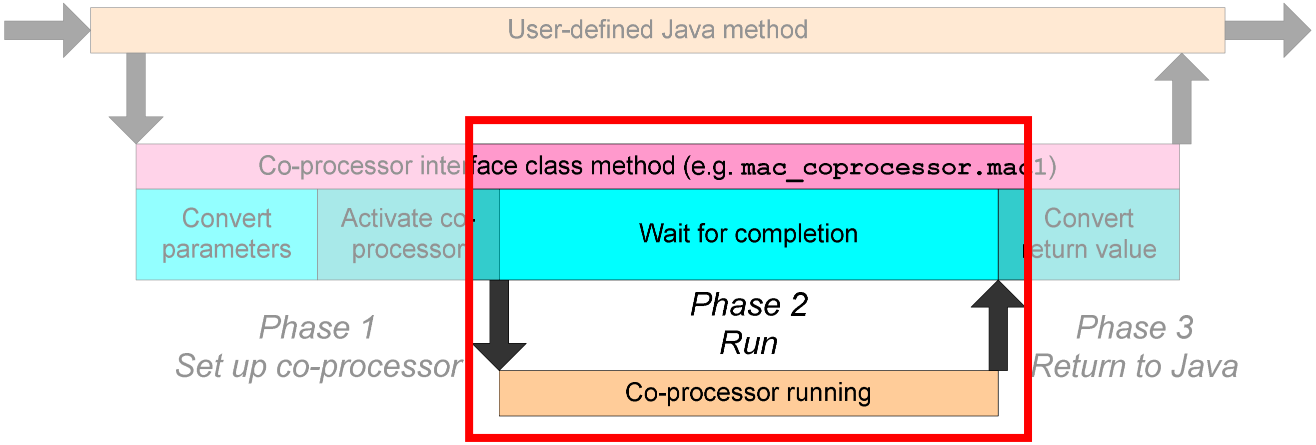
- Goal of WCET analysis for hardware methods: compute maximum time between point A and point B.

WCET Analysis for Hardware Methods (3)



- Phases 1 and 3 are easily analysed.
- WCET depends only on software operations.
- The existing WCA tool for JOP has all the required features.

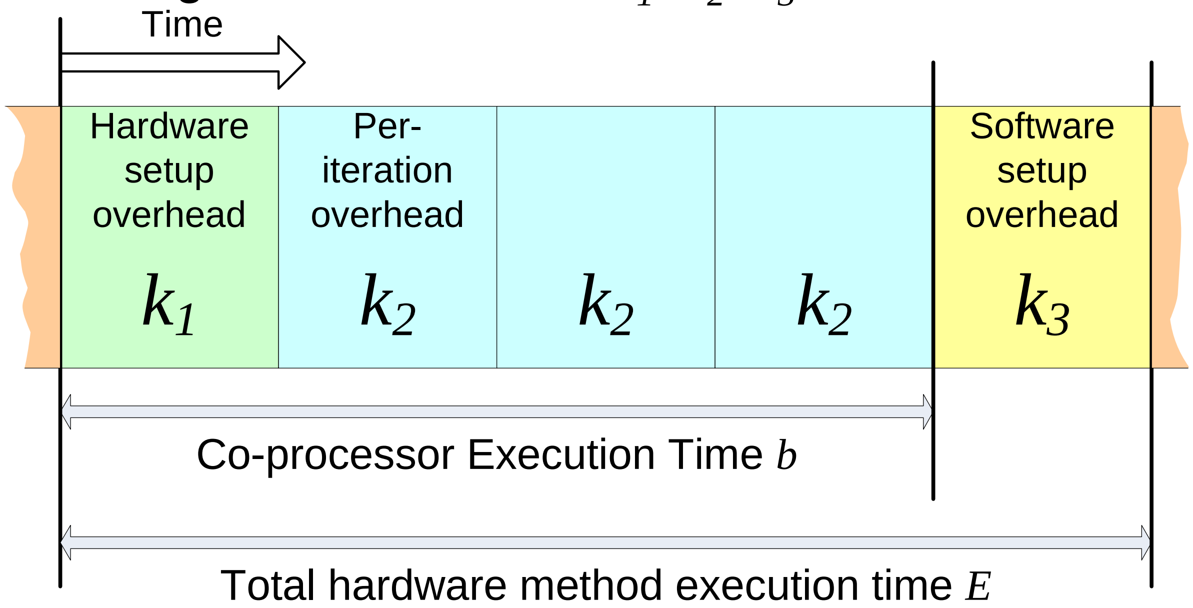
WCET Analysis for Hardware Methods (4)



- Phase 2 depends on the hardware execution time.
- In software, a `while` loop polls for completion.

WCET of Co-processor Hardware

- Assume the co-processor has a linear (i.e. $O(n)$) execution time.
- Model it using three constants, k_1, k_2, k_3:



k_3 is the cost of phases 1 and 3 (computed by WCA).

k_2 is derived by looking at the co-processor's state machine; how long does it operate on each data item?

k_1 is whatever remains.

WCET of Software

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

    while ((( reply & 1 ) == 1 )                    // @WCA loop<=s
    || (reply_identifier != (reply >> 16))) {

        control_channel.data = start_message;// ask: is done?
        reply = control_channel.data;        // reply: yes/no
    }
}
```

- Let i be the per-iteration cost of the `while` loop.
- Let E be the total hardware method WCET.
- The maximum number of loop iterations s is determined using an equation *(right)*.

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

Hardware Methods Evaluation

- Goal: compare the WCET of various functions on JOP, when implemented as:
  - Software (in pure Java)
  - Co-processors (using hardware methods)

- The evaluation considers the following:
  - Functions that process arrays.
  - Functions that may contain infeasible paths.
  - Functions that are naturally parallelisable.

Array Processing (1)

- Example: multiply/accumulate:

```
public int mac1(int size, int[]alpha, int[]beta) {
    int out = 0;
    for (int i = 0; i < size; i++)
    {
        out += alpha[i] * beta[i];
    }
    return out;
}
```

- Benefit of hardware methods: improved average and worst-case performance.

Array Processing (2)

Implementation of `mac1`	WCET (10,000 MACs)	Overhead $k_1 + k_3$	Per-iteration cost k_2
Pure Java	730,334	334	73
Hardware Method	60,916	916	6

- On the test JOP platform with one CPU and one hardware method, MAC is 12 times faster in hardware - *in the worst case*.

Infeasible Paths (1)

- Example: search an array for a maximum value:

```
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;         // how often?
    }
    return max;
}
```

- How often is the **`if`** condition true?
- Pessimistic assumption: *always*.
- Optimistic assumption: *once*.
- With a hardware method: *it doesn’t matter*.

Infeasible Paths (2)

Implementation of `search_max`	WCET (10,000 items)	Overhead $k_1 + k_3$	Per-iteration cost k_2
Pure Java (optimistic)	420,184	184	42
Pure Java (pessimistic)	450,308	308	45
Hardware Method	30,765	765	3

- The per-iteration cost is much smaller *and* it's the same in the best and worst case.
- Infeasible paths are not important.

Parallel Operations (1)

- Example: counting the number of bits that are 1:

```
public int bit_count(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;
}
```

- Benefit of hardware methods: do all operations within the inner loop *in parallel*.

Parallel Operations (2)

- Basic improvement using a lookup table:

```
public int bit_count(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;
}
```

- This provides *some* degree of parallelism…
- But hardware methods allow even more.

Parallel Operations (3)

Implementation of `bit_count`	WCET (10,000 items)	Overhead $k_1 + k_3$	Per-iteration cost k_2
Pure Java (naive)	12,300,308	308	1230
Pure Java (lookup table)	2,650,308	308	265
Hardware Method	30,765	765	3

- A substantial improvement!

Conclusion

- Hardware methods can be used to replace Java methods in embedded real-time systems:
  - They improve average and worst-case performance.
  - They act as plug-in replacements for software methods, abstracting the details of hardware access.

- Currently implemented for the JOP platform.
  - An implementation for the PC platform is in progress.

Thank You

- Questions?