Project Number 318763

D2.3 – Static Acceleration Design

Version 1.0
3 December 2013
Final

Public Distribution

University of York, aicas, SOFTEAM
Scuola Superiore Sant'Anna, University of Stuttgart

Project Partners: **aicas**, **HMI**, **petaFuel**, **SOFTEAM**, **Scuola Superiore Sant'Anna**, **The Open Group**, **University of Stuttgart**, **University of York**

Project Partner Contact Information

aicas Fridtjof Siebert Haid-und-Neue Strasse 18 76131 Karlsruhe Germany Tel: +49 721 66396823 E-mail: siebert@aicas.com	**HMI** Markus Schneider Im Breitspiel 11 C 69126 Heidelberg Germany Tel: +49 6221 7260 0 E-mail: schneider@hmi-tec.com
petaFuel Ludwig Adam Muenchnerstrasse 4 85354 Freising Germany Tel: +49 8161 40 60 202 E-mail: ludwig.adam@petafuel.de	**SOFTEAM** Andrey Sadovykh Avenue Victor Hugo 21 75016 Paris France Tel: +33 1 3012 1857 E-mail: andrey.sadovykh@softeam.fr
Scuola Superiore Sant'Anna Mauro Marinoni via Moruzzi 1 56124 Pisa Italy Tel: +39 050 882039 E-mail: m.marinoni@sssup.it	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of Stuttgart Bastian Koller Nobelstrasse 19 70569 Stuttgart Germany Tel: +49 711 68565891 E-mail: koller@hlrs.de	**University of York** Neil Audsley Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325571 E-mail: neil.audsley@cs.york.ac.uk

Contents

List of Figures

Document Control

Version	Status	Date
0.1	Document outline	4 November 2013
0.2	Complete First Draft	18 November 2013
1.0	QA for EC delivery	3 December 2013

Executive Summary

This document constitutes deliverable *D2.3 – Static Acceleration Design* of work package 2 of the JUNIPER project.

The purpose of this deliverable is to describe the design of the Static Acceleration facility for JUNIPER. Treating Java as a conventional language allows ahead-of-time compilation for better run-time performance. This deliverable focusses upon static acceleration; later deliverables will consider dynamic acceleration.

The purpose of Static Acceleration within JUNIPER is to speed up the execution of components of the Java application. The components selected are *locales*, identified manually in the source application by use of the new Java class `AcceleratableLocale`.

This deliverable contains a detailed design description of the Java acceleration approach. The approach builds upon the Java hardware methods approach [8] which allows easy wrapping of hardware components for access from Java as methods. Additions include handling of the features of locales, including Java threads and objects. Also, efficient mechanisms for communication between the FPGA from the JVM are established (related to work in Task 3.3). Finally the deliverable develops approaches for compiling Java directly to hardware (ie. FPGA), within the constraints of the Jamaica Java toolset used within JUNIPER.

1 Introduction

This document constitutes deliverable *D2.3 – Static Acceleration Design* of work package 2 of the JUNIPER project. The purpose of this deliverable is to describe the design of the Static Acceleration facility for JUNIPER. This is the first phase of acceleration work within JUNIPER – deliverables D2.4 and D2.5 will provide a description of the implementation and evaluation of the static acceleration design. Then, deliverables D2.6 and D2.7 will provide the design, implementation and evaluation of dynamic acceleration for JUNIPER. The intention is that the dynamic acceleration approach will build upon that for the static acceleration approach. The remainder of this document focusses upon static acceleration design.

Treating Java as a conventional language allows ahead-of-time compilation for better runtime performance. This deliverable focusses upon static acceleration, later deliverables (D2.6 and D2.7) will consider dynamic acceleration.

The deliverable considers the issues of:

- *what* components of the application should be accelerated statically.
- *when* components of the application are chosen for static acceleration.
- *how* to accelerate components of the application effectively.

This deliverable contains a detailed design description of the Java static acceleration approach. The approach builds upon the Java hardware methods approach which allows easy wrapping of hardware components for access from Java as methods. Additions include handling of the features of locales, including Java threads and objects. Also, efficient mechanisms for communication between the FPGA from the JVM are established (related to work in Task 3.3). Finally the deliverable develops approaches for compiling Java directly to hardware, within the constraints of the Jamaica Java toolset used within JUNIPER.

These areas are discussed further in the following sections:

- section 2 – The JUNIPER Conceptual and Run-Time Architecture
- section 3 – Programming Model
- section 4 – Static Acceleration Approach
- section 5 – Static Acceleration Tool Flow

2 The JUNIPER Conceptual and Run-Time Architectures

The JUNIPER tool flow, as illustrated in Figure 1, was introduced in D1.2 – the remainder of this section provides a summary to establish where static acceleration of Java application components sits within the overall JUNIPER project.

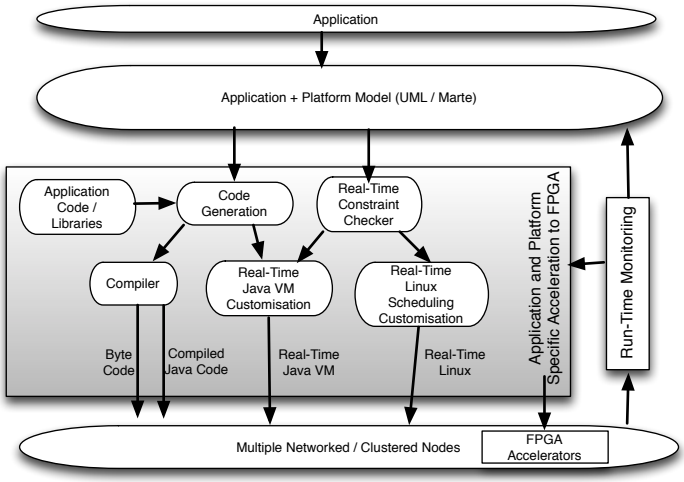

Figure 1: Conceptual Flow

Applications are modelled in UML, using the Marte subset to represent real-time and platform architecture aspects, allowing real-time constraints to be modelled and checked. The target language is Java, adopting the Real-Time Specification for Java extensions. To increase performance, we ensure that the parallel nature of the hardware platform is suitably abstracted to the application, and we accelerate key parts of the Java application, run-time and libraries.

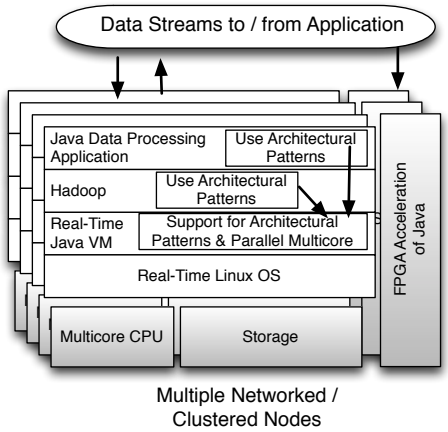

Figure 2: Run-Time Architecture

The run-time (software) architecture resulting from the flow is illustrated in Figure 2. The Real-Time Java application is executed on a Real-time Java Virtual Machine allowing for both interpreted and compiled Java (the latter for performance purposes). This forms a basis for building high performance real-time big data applications. At run-time, run-time monitoring and profiling of the developed application and system infrastructure occur. This permits further modification of run-time scheduling parameters (i.e., time and resource bandwidths allocated to individual applications and processes), and of the accelerated components (i.e. changing the components accelerated in the FPGA). Note that such dynamic acceleration will be discussed in a later deliverable (ie. D2.6 and D2.7).

3 Programming Model for Acceleration

Within this section, we address the issues of *when* decisions are made regarding which application components to accelerate and *what* application components to accelerate. The JUNIPER approach to acceleration is source code-based (i.e. from the Java source). Acceleration begins at the code level with identification of application candidate components for acceleration. Within JUNIPER, components are selected manually by the application developer, although this decision can be informed by any profiling information made available within the wider JUNIPER platform when the application is executed.

The remainder of this section discusses the parts of the JUNIPER Java programming model that can be accelerated.

3.1 JUNIPER Programming Model: *Locales*

The JUNIPER programming model (to be fully described in deliverable D2.1) is based on the forthcoming Java 8 and the Real-Time Specification for Java (RTSJ) [1]. We note that implementations of Java 8 now exist, with aicas currently working to implement Java 8 within the lifetime of JUNIPER.

In deliverable D1.2 the concept of *locales* is introduced for the JUNIPER Java programming model. A locale is a mechanism to group threads and their objects. Within JUNIPER the locale is used as the unit of the application that is acceleratable as a single component – hence is distributable and allocatable to the accelerator (i.e. FPGA).

A locale has the following properties (drawn from D1.2, updated with respect to hardware acceleration of locales):

- The threads and objects encapsulated in a locale are mapped by the JVM onto CPUs and into memories that form an SMP architecture pattern within the hosting platform; and also onto the FPGA acceleration platform.
- A locale is given a resource reservation that is the result of a negotiation between the JVM and the host operating system; where locales are allocated to the FPGA acceleration platform, negotiation will include consideration of resource allocation on the FPGA.
- A locale has a backing store which describes its local memory (i.e. for heap and stack allocation). This is allocated to the locale and not shared with any other. During acceleration, this memory will be physically located on the FPGA.
- References between acceleratable locales must be controlled. The current acceleration design does not permit the locale to contain references to other locales. Locales must communicate and share data through the JUNIPER Communications API (to be defined in D2.1), which uses bindings to MPI to implement this functionality. As the implementation develops, this restriction may be relaxed.
- The Java programming model requires either a garbage collector (GC) or a scoped memory scheme (such as found in the RTSJ) in order to reclaim dynamically allocated memory. The acceleration scheme will assume the use of scoped memories to remove the requirement for a full GC.

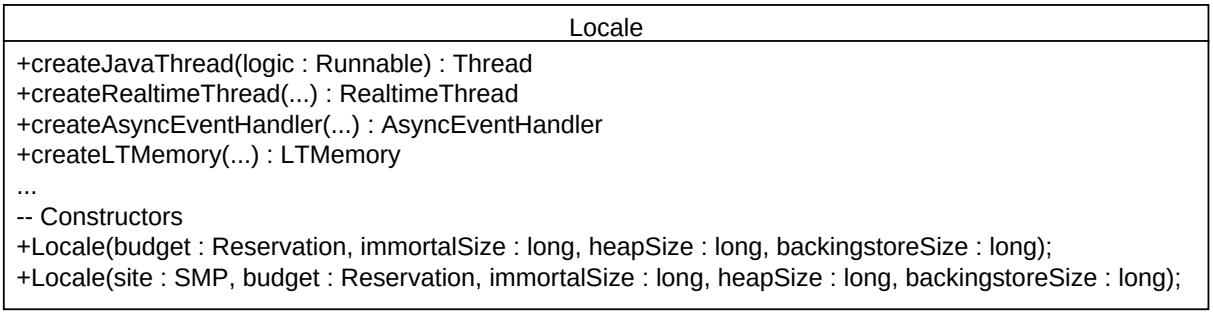

Figure 3: The Application Model

An abridged version of the Locale class is show in Figure 3. The full details are described in WP 2.2.

The approach taken is to provide factory methods to create the main thread (threads, real-time threads, and asynchronous event handlers) and memory (scoped, physical – including a physical heap) types in the RTSJ. Creation of these objects outside of these factory methods has no locality defined and can be located at the JVM's discretion. However, threads can dynamically have their affinities set (for example, when part of an executor's thread pool).

The goal of the JUNIPER model is to restrict the impact caused by adding locality to a few well-defined places in the Java libraries. So for example, rather than adding new constructors for real-time threads which contain a locale parameter, we provide factory methods in the Locale class.

Whilst locales facilitate collocation of data and threads, they do not directly facilitate distribution of collections or thread pools *across multiple locales*. This is discussed below. Locales also do not facilitate across-locale distribution of arrays of primitive types and parallel *for* loops. These will initially not be supported by the JUNIPER programming model as the same effect can be achieved using the Juniper collection framework.

3.1.1 Locale Example

The following example shows how locales can be used to keep a set of created threads together by allocating them inside the same locale. The JVM can use this information to know that these threads will observe faster performance if they are kept close together. Other parts of the application may be placed elsewhere in the system.

```
public class PrimeGenerator {
    public static void main(String[] argv) {
        long hi = /* get from user */;
        Platform p = Platform.getPlatform();
        NUMA numa = p.getRootLocation();
        CCNUMA ccnuma = numa.getChildren()[0];
        SMP smp = ccnuma.getChildren()[0]; // Get first SMP
        Locale locale = new Locale(smp);
        int ncpu = smp.getNumCPUs();
```

```
        for (int i = 0; i < ncpu; i++) {
            final long min = ((long) i) * hi / (long) ncpu;
            final long max = ((long) i + 1) * hi / (long) ncpu;
            Thread th = locale.createJavaThread(() -> {
                // Search from min to max to find a prime
                //...detail skipped...
                if (found) System.out.println(n);
                //...
                }
            });
            th.start();
        }
    }
}
```

In the example above, a set of threads will be created in order to parallelise the search for prime numbers. The example begins by creating a thread for each CPU of the current SMP platform. This shows a simple example of the JUNIPER API being used for architecture discovery. Once the target number of threads is determined, the total search space is split evenly between the threads spawned. The threads are created using a locale factory method. This informs the JVM that the threads are tightly-coupled and should be kept close together (for example in the same SMP of a NUMA system).

3.2 JUNIPER Programming Model: *Acceleratable Locales*

The developer is required to identify the locales within the application that are amenable to static acceleration on the FPGA. The OpenACC standard supported by Cray, nVidia *et. al.* contains a pragma for use within C/C++ applications. We note that there is no direct Java binding to OpenACC, and so targeting GPU languages such as CUDA and OpenCL requires the use of third party compilers such as Rootbeer [1] which does not support the RTSJ or real-time Java in general. Project Sumatra [2] is a move to include accelerators in Java but it is still in the planning phase.

The approach taken within JUNIPER is to introduce a subclass of the `Locale` class called `AcceleratableLocale`. `AcceleratableLocale` includes an abstract method called `initialise` which creates all of the threads and data that will be allocated inside that locale. (Normal locales can allocate threads and data freely, subject to the normal RTSJ restrictions). A `Locale` cannot be created inside `AcceleratableLocale.initialise()` but an `AcceleratableLocale` can be.

The preliminary structure of the `AcceleratableLocale` class is shown below.

```
public class AcceleratableLocale extends Locale {
        abstract public void initialise(Map<String, Object> args);

        public AcceleratableLocale(SMP site);
        public AcceleratableLocale();
        public Locale(SMP site, Reservation reserv);
```

[1] http://rbcompiler.com/index.html
[2] http://openjdk.java.net/projects/sumatra/

```
        public Thread createJavaThread(Runnable logic);
        public Thread createRealtimeThread(Runnable logic);

        public void bindCurrentThread();
        public void bindThread(Thread th);

        public Object createLTMemory(long size);
}
```

When an `AcceleratableLocale` is created it is assigned to a `Location` (see the example above), which is a physical CPU (or FPGA coprocessor). This is analysed ahead of time during compilation so that the code from the locale can be compiled for the FPGA. As a result, the `initialise` method must be amenable to static analysis. The chosen strategy is that the compiler will execute the `initialise` method and observe the threads and data that it instantiates. Therefore the `initialise` method must not be non-terminating and there are restrictions on what actions it can take. The precise restrictions that this implies will be defined during implementation and described in deliverable D2.4. The rest of the locale can be standard Java, although again some restrictions may also be imposed by the final implementation details.

It will be possible for an `AcceleratableLocale` to use MPI (via the JUNIPER API) to interact with locales on the host or other places in the system. Java synchronization and remote method invocation will not be used directly and should be wrapped in the JUNIPER API.

3.2.1 Acceleratable Example

The example from section 3.1.1 can be edited to support an `AcceleratableLocale` as follows:

```
public class PrimeGenerator {
        private long hi;
        private class SieveAcceleratableLocale extends AcceleratableLocale {
                @Override
                public void initialise(Map<String, Object> args) {
                    for (int i = 0; i < 4; i++) {
                        final long min = ((long) i) * hi / 4;
                        final long max = ((long) i + 1) * hi / 4;
                        ManagedThread th = new ManagedThread(new PriorityParamaters(1),
                                    new StorageParameters(...), new Runnable() {
                                     public void run() {
                                         // Search from min to max to find a prime
                                         //...detail skipped...
                                         if (found) System.out.println(n);
                                         //...
                                     });
                        th.register();
                    }
                }
        }

    public static void main(String[] argv) {
        hi = /* get from user */;
```

```
        Platform p = Platform.getPlatform();
        ...
        SieveAcceleratableLocale locale = new SieveAcceleratableLocale();
    }
}
```

Note that the locale used is now an `AcceleratableLocale` which creates and registers `ManagedThreads` in its `initialise()` method. It also does not use the `getNumCPUs()` method and instead requests a specific number of threads.

3.3 Java Programming Model Issues

There are a number of issues that must be considered when accelerating a locale by moving its functionality to FPGA. Implementation issues are considered in section 4. The remainder of this section considers issues at the Java language level.

3.3.1 Inter-Locale Communication

Within the programming model of D1.2 (to be finalised within D2.1), a Java program is executed on a single JVM, with one JVM per platform (i.e. CPU or multicore). A single Java program may contain multiple locales. The JUNIPER inter-locale communication model is based on a Java binding to MPI [3]. This will maintain consistency with assumed platforms, which are cluster and supercomputers using MPI.

Within a program there can be many locales. D2.1 will define the communication between locales upon the same JVM – there are advantages in using MPI communication between all locales (e.g. the ability to migrate), but at the potential cost of increased overheads. The approach to static acceleration given later in this deliverable is amenable to either choice.

In the RTSJ, the programmer may use the `RawMemory` class to describe additional memory spaces in the host platform which are not used by the JVM normally for the stack, heap, or immortal memory. For example, an instance of the `RawMemory` class may describe a section of flash memory which the programmer may then use to explicitly write data into and read data from. `RawMemory` does not store Java object instances, only primitives and arrays of primitives.

Inside a JVM which supports offloading to an FPGA, the JUNIPER API will maintain an instance of `RawMemory` which represents the local memory of the FPGA. This allows the JVM to move and share data between the normal Java code executing as software in the JVM and accelerated code executing as hardware in the FPGA. All acceleratable locales will, when initialised on the FPGA, draw their memory requirements from the FPGA's memory.

Passing data via shared memory between software locales is handled by MPI's normal mechanisms for doing this. MPI automatically makes use of shared memory for data transfers when it is available.

[3] NB Using MPI implementations such as MPICH will allow multiple programs to be integrated within a single distributed MPI framework.

4 Static Acceleration Approach

Within this section we address the issue of *how* Java locale components can be accelerated within the context of the overall JUNIPER platform. Initially, key requirements for the acceleration approach are derived from the programming model (section 3); subsequent sub-sections describe the acceleration approach in detail, indicating how the requirements will be met.

4.1 Java Programming Model Acceleration Requirements

From the JUNIPER Java programming model acceleration restrictions discussed in section 3, and the wider behaviour of the JUNIPER platform (described in D1.2), specific requirements for the static acceleration approach can be derived. The static acceleration approach *SHOULD*:

- support multiple locale components accelerated on an FPGA simultaneously;
- provide communications between Java software locales and locales on an FPGA;
- allow simple data (primitives and primitive arrays) to be shared and communicated between locales on a CPU and those on an FPGA;
- provide a separate physical memory space for locales on an FPGA;
- allow static (offline) choice of which locales will be allocated to an FPGA.

The emphasis within the static acceleration approach in JUNIPER is a static choice of which locales can be accelerated (see section 3.2), and which of those locales are actually accelerated at run-time – ie., the application development process identifies which locales will be placed on the FPGA. We note that the dynamic acceleration approach (D2.6 and D2.7) will relax these restrictions.

In terms of the provision of communications between the Java application running on the CPU and the accelerated parts running on the FPGA, this is partially dependent upon the underlying software stack, ie. the JVM and OS. Hence the static acceleration approach is supported by work in WP3 Task 3.3 (FPGA support within Linux) and WP4 Task 4.1 (JVM extensions).

4.2 Acceleration Platform Overview

In section 2 the role of acceleration was presented within the context of conceptual flow (figure 1) and run-time architecture (figure 2). Detailed description of the acceleration part of the platform is shown in figure 4. We note:

- An FPGA is provided as part of physical architecture to give a platform for hardware acceleration – the FPGA has known physical characteristics (ie. size, memory size etc.).
- A parallel Java application is accelerated by placing statically identified and selected locales within the FPGA (locales placed on the FPGA remain notionally under the control of the JVM on the host CPU):
    - Data Filter – part of the Java application will be responsible for processing incoming data (for storage on disk), this is placed on the FPGA (alongside the network input) to allow fast processing;

  - Java Application Locales – locales within the application (statically) selected for acceleration on the FPGA. Communication of structured data between locales on the CPU and those on the FPGA is achieved via the JUNIPER API.
- Integration of the FPGA with the OS to enable efficient access to/from the FPGA for loading Java components to the FPGA; also for monitoring etc. (see WP3 Task 3.3).
- Potentially, key components of the OS itself, particularly the filesystem, can be placed onto the FPGA in order to accelerate the storage of incoming data and also access of data stored by the application. This is illustrated in figure 5.

We note that accelerating parts of the OS (ie. filesystem) lies outside the Description of Work for the JUNIPER project, but is included as indication of further improvement and use of FPGA acceleration for real-time big data applications.

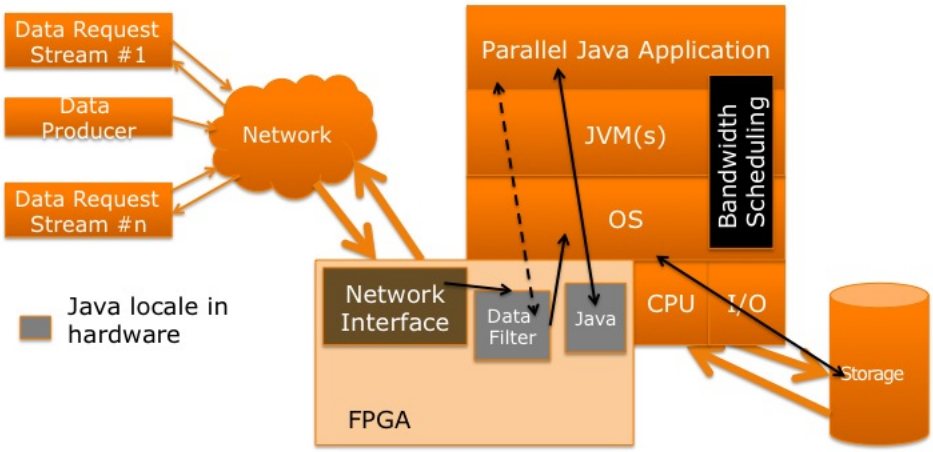

Figure 4: Acceleration of the Programming Model using Java Components Synthesised to FPGA

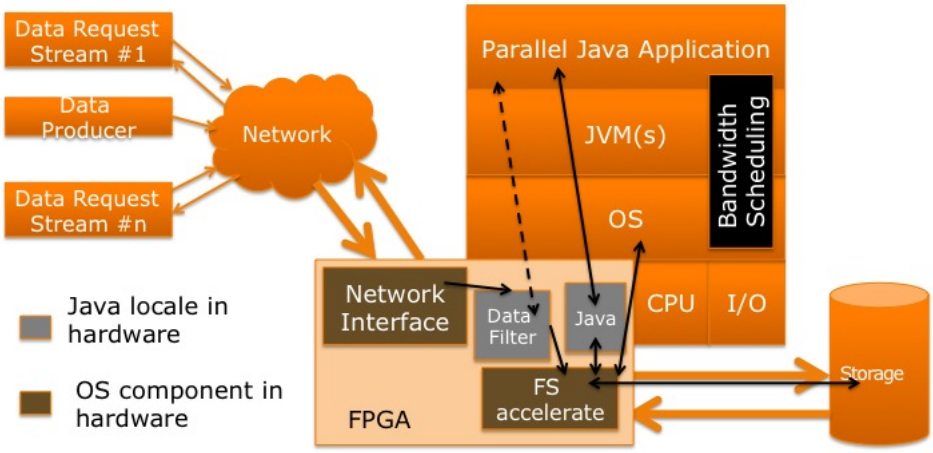

Figure 5: Acceleration of the Programming Model using Java Components Synthesised to FPGA with Accelerated Filesystem

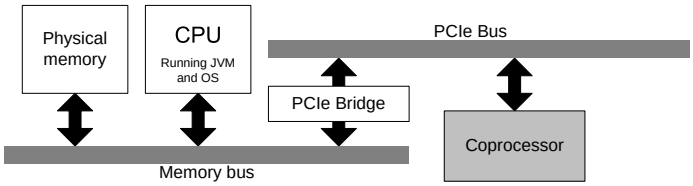

Figure 6: Simplified PC or server architecture – the coprocessor is found on an expansion bus such as PCI Express. The coprocessor may contain private memory.

4.3 JVM to Hardware Locale Communication

Application locales executing as software on the JVM require communications to hardware locales on the FPGA. This requires the ability to pass and share structured data between software and hardware locales, and for the software locales to call methods within the hardware locale. Given the discussion regarding the programming model (section 3.3.1), the main approaches to achieving software locale to hardware locale communications are:

- *via MPI and JNI*:
  The Java JNI (Java Native Interface [5]) is used to call an MPI library outside the JVM (ie. C based MPI library); a (minimal) MPI interface is provided on the FPGA for the hardware locales.
- *via MPI*:
  A direct Java-MPI binding is used (eg. OpenMPI); a (minimal) MPI interface is provided on the FPGA for the hardware locales.
- *via JNI*:
  The Java JNI (Java Native Interface) is used to call the hardware locales; this requires a wrapper for the hardware locales on the FPGA to transpose incoming JNI calls appropriately for the hardware locale.

We note that resolution of the best approach will become clear after the D2.1 has been produced containing the complete programming model. However, clearly there is a cost in including an MPI interface as part of a hardware locale in terms of hardware area and overheads (it would remain for any implementation to quantify overheads).

Shared memory between software and hardware locales is also required in order to transfer bulk data between the two. Physically, typical FPGA boards have separate memory to the CPU – but DMA transfers can be set up to move data between the boards. This will support the `RawMemory` approach in the programming model (section 3.3.1) where effectively the two areas of physical memory are both known by the (software) JVM.

4.4 On-FPGA Infrastructure

Conventionally, an FPGA accelerator is placed on an expansion bus, such as PCI Express (PCIe), USB or HyperTransport [4]. This leads to a configuration similar to Figure 6. This structure dictates part of the on-FPGA infrastructure needed by the static acceleration approach – ie. the presence of

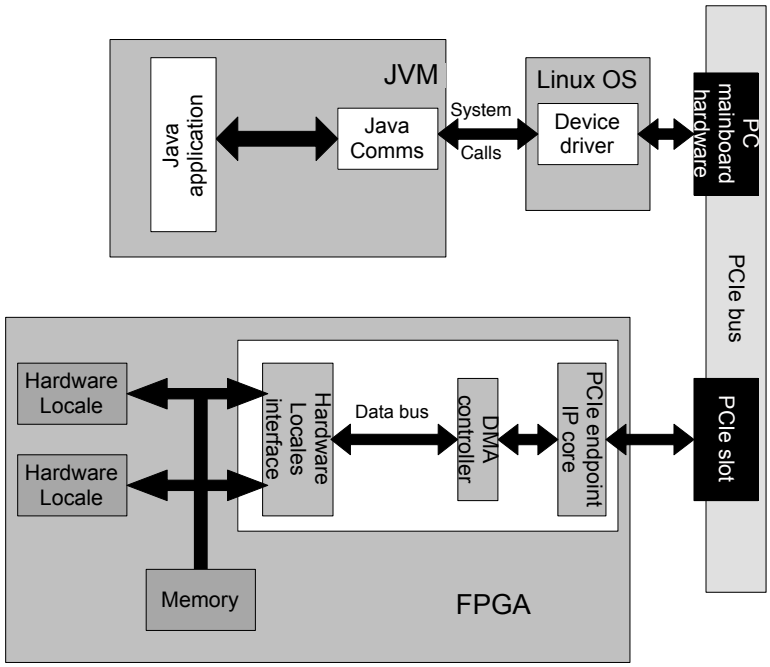

Figure 7: Logical Infrastructure.

a PCIe endpoint on the FPGA to route communications to/from the PCIe bridge, and hence to/from the CPU (and its memory).

The logical infrastructure of the acceleration approach is shown in Figure 7. Here a Java application communicates to hardware locale via PCIe, using one of the communications approaches identified above (ie. JNI, JNI-MPI or MPI). Key infrastructure on the FPGA includes the ability to communicate with the CPU using the communications approach selected (ie. within the Hardware Locales Interface component). Also included is the ability to interface to PCIe, and to use DMA transfer requests to move data between FPGA memory (ie. hardware locale memory) and memory directly connected to the CPU (software locales).

The physical infrastructure needs to support a number of hardware locales on a single FPGA, and incorporate the communications components illustrated in Figure 7 (ie. Hardware Locales Interface, DMA and PCI controllers). To support a number of hardware locales with connections to the communications components and memory, a simple topology would involve a single shared bus. As the number of hardware components scales, it is unclear what the best topology will be, given that there are (inevitably) bottlenecks for communication across PCIe, and also to memory. During implementation, JUNIPER partners will consider both shared bus and more scalable topologies such as Network-on-Chip (via a framework developed at York [2, 3, 6]).

5 Static Acceleration Tool Flow

Within this section we continue consideration of *how* Java locale components can be accelerated within the context of the overall JUNIPER platform. This section focusses upon a potential tool flow.

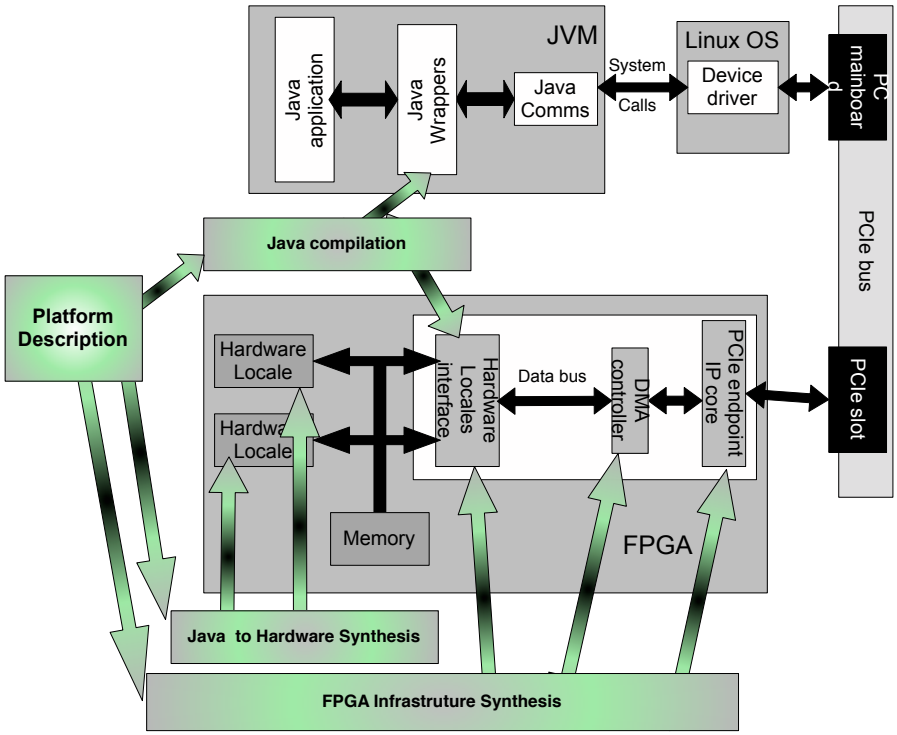

Figure 8: Abstract Tool Flow

5.1 Abstract Tool Flow

Figure 8 shows the Logical Infrastructure of the acceleration approach (Figure 7) supplemented by aspects of the tool flow, namely Java compilation of the application, and synthesis of the hardware locales. A "Platform Description" is assumed which contains information including:

- details of hardware architecture, eg. FPGA board, size, memory layout etc.
- details of communications stack, eg. specifics regarding MPI (if used) and / or JNI implementations that may impact the on-FPGA infrastructure.
- details of host platform (ie. PC) required for FPGA infrastructure synthesis.

In general, the approach taken is similar in design to that used in the EU FP7 project JEOPARD [7], which accelerated selected methods from host Java code by converting them to components for an attached FPGA. However, we note that synthesis of these 'hardware methods' merely wrapped existing hardware components to be callable from Java – synthesising active hardware from Java with threading was not part of the JEOPARD approach. A potential tool flow for achieving this is discussed in the following section.

5.2 Tool Flow for Hardware Implementation of Java Locales

The following stages are required in order to convert the input Java for acceleration. Note that the static acceleration process assumes that the designer will choose which `AcceleratableLocales` should be synthesised for implementation on the FPGA and which will remain in software. This will be described in the Platform Description (see Figure 8). Deliverable D2.6 which considers dynamic acceleration will revisit this assumption.

1. For each `AcceleratableLocale` in the software that is to be compiled to hardware, the flow will statically analyse its `initialise` method to determine the structure of the locale. Recall from section 3.2 that acceleratable locales may not dynamically allocate threads or data – they are restricted to allocations in their `initialise` method. The exact information required will be determined by the implementation of the tool flow, but will include the threads and schedulable objects of the locale, their entry points, and JUNIPER API calls that are made in order to assist the hardware translation to create the structure of the final component.

2. The `AcceleratableLocale` will interact with its host through the JUNIPER API in order to fetch data, communicate with other nodes in the system, and synchronize with Java executing in software. The implementation of the JUNIPER API calls that implement this host-to-FPGA communication will be automatically generated according to the Platform Description. There are two sides of this autogeneration:
   - The Java side (host-to-FPGA) involves the implementation of native methods for the JamaicaVM to link as required.
   - The hardware side (FPGA-to-host) involves the generation of C for later translation to HDL.

3. The `AcceleratableLocales` will be converted to C code using the JamaicaVM from aicas. This will link to the generated implementation of the JUNIPER API. Note that an initial transformation pass of the code for the `AcceleratableLocales` will be required to ensure that suitable C code is produced. Few C to hardware tools support threads, so threading will be factored out of the input Java using information extracted at the start of the flow. This factoring will turn a single multi-threaded Java program into multiple single-threaded programs that can be converted individually. This will depend on implementation details and will be detailed in deliverable D2.4.

4. The C code will be converted to an FPGA component using Xilinx's AutoESL, or a similar tool. This translation will include the FPGA-to-host code generated above. The resulting hardware component will not be the entire FPGA design but a self-contained subcomponent.

5. HDL code will be generated which wraps the generated hardware components of all accelerated locales in an FPGA top-level design.

5.2.1 Illustrative example

It is not possible to get a precise example yet as the implementation work is under way. Instead, this section shows a small example for illustrative purposes. Details may change in the final tool flow.

The code in section 3.2.1 uses threads to search for prime numbers. The acceleration flow will analyse this code and determine that 4 threads are created with the following bodies:

```
long n;
for (n = min; n < max; n++) {
        if ((n & 1) == 0) continue;
        boolean skip = false;
        for (long d = 3; d < n; d += 2)
                if (n % d == 0) {
                        skip = true;
                        break;
                }
        if (!skip) System.out.println(n);
}
```

This is the code which will be wrapped and passed to JamaicaVM for translation to C and then HDL. Further auto-generated code must be created to start these thread bodies and handle any coordination between them. Note that these thread bodies return values to the host (by printing to the screen using `System.out.println`). This and similar methods which involve JVM *native* methods will be replaced with new implementations. In the case of `println()` the new implementation will send strings back from the FPGA to the host and the host will echo them to its terminal, as expected.

JamaicaVM creates C code similar to the following example output:

```
{
        ...
        jamaica_ref new_ref=r2;
        if(new_ref!=JAMAICA_NULL) {
                if(JAMAICA_BLOCK_GET_R(new_ref,0)==gc->white) {
                        JAMAICA_BLOCK_GET_R(new_ref,0)=gc->greyList;gc->greyList=new_ref;
                }
                l[1]=new_ref;
        }

        if (r2==JAMAICA_NULL) {
                goto LABEL_tNPE;
        }
        n0=JAMAICA_BLOCK_GET_DATA32(r2,3).i;
        n1=((n0>>4) &  ((jamaica_int32)268435455));
        r3=r0;
        ...
```

As can be seen, this includes Jamaica functions that will have to be reimplemented so that they work in the C-to-HDL tool. This can then be passed to C-to-HDL translators to create HDL for wrapping an implementation, as described above.

6 Conclusions

This document has described the Static Acceleration Design for the JUNIPER project. Essentially, the approach taken is to accelerate restricted parts of the Java application. In common with the Programming Model (being detailed in D2.1), the part of the Java application selected is the *Locale*, a collection of methods and objects identified by means of an API call to the JVM run-time. Selected locales are statically (offline) synthesised to FPGA hardware circuit, with suitable infrastructure (software and hardware) generated to ensure communication and data transfer can occur between software and hardware domains.

The Static Acceleration Design will be implemented and evaluated during the next phase of the JUNIPER project. The intended implementation toolflow has been described in this document. Currently, the intention is to leverage the Java toolset used within JUNIPER to produce a C output, with this transformed to be suitable for a commercial C to FPGA tool. Clearly, this intended implementation path may change as JUNIPER moves into the implementation phase for static acceleration, particularly if perceived improvements in performance are not seen.

Also, the Static Acceleration approach described in this document will form the basis of the Dynamic Acceleration approach to be defined within WP2. The fundamental difference between the static and dynamic approaches is the need to allow dynamic selection of the Java locales to be accelerated at run-time.

References

[1] Greg Bollella and James Gosling. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[2] Jamie Garside and Neil C. Audsley. Investigating Shared Memory Tree Prefetching within NoC Architectures. In *Proceedings Memory Architecture and Organization Workshop MEAOW*, 2013.

[3] Jamie Garside and Neil C. Audsley. Prefetching Across a Shared Memory Tree within a Network-on-Chip Architecture. In *Proceedings International Symposium on System-on-Chip*, 2013.

[4] Lattice Semiconductor Corporation. LatticeSC/M HTX Evaluation Board and Reference Design. `http://www.hypertransport.org/default.cfm?page=ProductsViewProduct&ProductID=94`, 2010.

[5] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[6] Gary Plumbridge, Jack Whitham, and Neil C. Audsley. Blueshell : A Platform for Rapid Prototyping of Multiprocessor NoCs and Accelerators. In *Proceedings HEART Workshop*, 2013.

[7] The JEOPARD consortium. JEOPARD - Java Environment for Parallel Realtime Development. `http://www.jeopard.org/`, 2010.

[8] Jack Whitham, Neil Audsley, and Martin Schoeberl. Using Hardware Methods to Improve Time-predictable Performance in Real-time Java Systems. In *Proc. JTRES*, pages 130–139, 2009.