FPGA-based hardware acceleration for Real-Time Big Data systems

Ian Gray, Yu Chan, Jamie Garside, Neil Audsley, Andy Wellings
Real-Time Systems Group
Department of Computer Science
University of York
{ian.gray, yc522, jamie.garside,
neil.audsley, andy.wellings}@york.ac.uk

***Abstract*—This paper discusses how FPGA acceleration is used within the JUNIPER platform. JUNIPER is a processing platform to enable the development of real-time, Big Data systems. Unlike existing Big Data approaches which are based on either batch processing, or streaming processing that is "fast enough", the JUNIPER platform integrates a range of technologies that increase the predictability of the system allowing for design time analysis and timing guarantees. One of these technologies is FPGA-based acceleration of Java.**

General purpose hardware translation of Java is a challenging problem. It is made possible in this work because of a number of features in the JUNIPER programming model. Rather than attempting to translate the entire application, the programmer can mark specific sections of their program (called Locales) as acceleratable. Communications in and out of Locales are restricted to use the MPI-based JUNIPER communications model.

Initial results show that the use of Java does not hamper hardware generation, and provides tight execution time estimates. This paper describes the work currently under way, the approach being developed, and presents some preliminary results that demonstrate the promise in the technique.

I. Introduction

Big Data is the term used for applications that cannot be developed using existing data processing techniques, because of either the sheer scale of the data being produced, or timing requirements that are placed on the data processing, filtering, and storage. The majority of existing Big Data systems are *batch-based* [1], meaning that processing is run periodically, and queries that trigger such processing may take significant time to complete. To avoid this, more recent systems [2], [3] have adopted a data streaming model in which input data is constantly being filtered and processed.

The JUNIPER project is an EU-funded project that is developing a framework for the development of *real-time* Big Data systems. In the context of Big Data, "real-time" is commonly used to mean "fast enough", or streaming-based. In this work we take a more strict definition of real-time to mean that the correctness of the data is dependent on both its value and the time by which it is delivered.

The JUNIPER framework is a Java 8 [4] API which includes a range of technologies to allow real-time, quality of service (QoS) guarantees to be provided, based on ahead-of-time scheduling analysis rather than relying on testing and profiling (as seen in figure 1). JUNIPER integrates the following technologies in a common API:

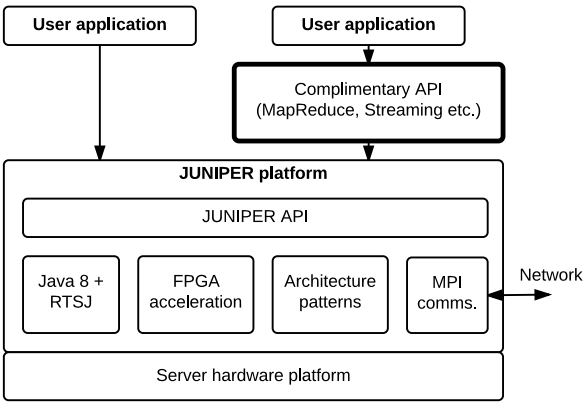

Fig. 1. The JUNIPER API unifies a range of real-time technologies for ease of development.

- The JUNIPER API adds many features to Java 8 to allow efficient distribution of data collections over a cluster environment.
- Features are also added to automate distributed streaming and map-reduce processing of these distributed collections. This allows easy implementation of both Hadoop and Spark/Storm-style algorithms.
- JUNIPER uses the Real-Time Specification for Java [5] to support the analysis of user code to determine its worst-case execution time, and integrates with existing scheduling analysis frameworks.
- The JUNIPER OS allows for disk and network *reservations*, allowing user-level processes to reserve bandwidth to ensure that other system activity cannot affect high-priority parts of the application.
- The JUNIPER API adds architecture awareness for efficient use of high-performance computing (HPC) and ccNUMA-style servers (more details in section III-A).
- JUNIPER also includes automatic translation of Java to FPGA hardware for reasons of both performance and predictability. This is the focus of this paper.

This paper covers the motivation for translation of Java to hardware (section II), and discusses the programming model

This work has received funding from the European Union's Seventh Framework Programme under grant agreement FP7-ICT-611731

exposed by the JUNIPER API (section III). Section IV then discusses the implementation strategy used, and preliminary results are reported in section VI.

II. Motivation and existing work

Due to Java's use of a Virtual Machine and its stated aim of being highly portable and not bound to a specific implementation target, hardware acceleration is not particularly well supported. The OpenACC standard supported by Cray, nVidia et. al. contains a pragma for use within C/C++ applications. 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 [6] which does not support the Real-Time Specification for Java. Project Sumatra [7] is a move to include accelerators in Java but it is still in the planning phase. All of this work targets GPU-style data parallel accelerators however. Whilst this is very useful for high performance work, the JUNIPER platform is also interested in predictability for more general code.

Classic work [8], [9], [10] has attempted to translate Java bytecodes directly into datapath hardware to moderate success. These approaches are limited by the fact that they cannot handle general-purpose Java and place limitation on aspects of the language. There has not been much recent work in this area.

The motivation for using Java at all for this, is that JUNIPER aims to provide simple acceleration of large-scale software and most of the major frameworks (Hadoop, Spark and Storm) are written in Java. Similarly, languages such as Clojure which are commonly used for this work still use the JVM for their implementation and so would be compatible with this approach.

Existing work [11] has focussed on offloading sections of software to dedicated accelerators on an FPGA. This work has obtained good results, and the rough approach serves as a base for this work. However it begins with C and involves a level of user interaction (pragmas etc.) which this work is attempting to minimise.

Equally, there are a wide range of approaches that focus on offloading of vector-based operations, such as Xilinx's SDAccel for offloading OpenCL workloads. These are likely to be better than the approach in this paper for scientific computing, in which existing compute kernels can be leveraged more easily.

JOP [12] is a processor that can natively execute Java bytecodes and is optimised towards predictable execution. It cannot obtain the highest possible execution speeds, but is very effective for predictable behaviour.

The approach described in this paper uses elements of all of these approaches. Through an expanded programming model, areas that are amenable to acceleration are identified. These areas are not translated wholly, partly down to the problems identified in the classic work cited above. Instead parts of the Java (such as memory allocators and the garbage collectors) remain on an embedded processor whilst methods are turned into datapath hardware. The rest of the paper details the approach taken.

III. Programming model

The JUNIPER API exposes a programming model which extends that of Java 8 to support large-scale computing environments, such as clusters ("cloud computing") and HPC. The full details of the JUNIPER model are outside of the scope of this paper and are detailed in existing work [13], [14]. Only the parts relating to acceleration will be detailed here.

A JUNIPER *application* is a logical concept, executing across the entire target platform (cluster or HPC), and is comprised of a set of interacting JUNIPER *programs*. Programs execute inside individual JVMs (limiting them to SMP or ccNUMA nodes) and they communicate using MPI. The JUNIPER API allows the programmer to write one application which is automatically decomposed into a set of communicating programs mapped to the specific hardware platform. The JUNIPER approach includes extensive model-driven development and code generation support to assist with deployment and code generation to achieve this.

Aside from providing mechanisms to allow a Java programmer to effectively use an entire cluster or HPC, the JUNIPER platform also provides APIs to maximise the performance use of individual servers. This is driven by the observation that software may execute in a range of places due dynamic mapping decisions made by the cloud or HPC management middleware. Most HPC and cloud environments are not entirely homogeneous, and so application software should react accordingly, scaling itself according to number of CPU cores and cache hierarchy. In JUNIPER, this is made easy through the use of *locales*.

A. Locales

Locales in the JUNIPER API provide an infrastructure with which the programmer can manage the *locality* of the code and data of their system. In existing systems, programmers are forced to place threads and data manually using affinities. This is error-prone, non-portable, and onerous on systems with many cores. Also, it is not the appropriate abstraction to use. The programmer does not want to express that two tightly-coupled threads should both exist on CPU core x, merely that wherever they are mapped they should be mapped 'close by'. Locales allow this.

A locale is a software-level element which is used to inform the JVM that the threads and data inside a locale will be tightly-coupled and so should be located as closely together as possible. These bundled threads and data items are then mapped to dynamically-discovered subsets of the target architecture, exposed as *architecture patterns* which describe types of architectures:

- NUMA: The NUMA architecture pattern provides few guarantees. It will contain a single address space, but caches may be incoherent and memory access times are unknown.
- ccNUMA: The cache-coherent NUMA architecture constrains the NUMA architecture with the guarantee that caches will be kept coherent from the point of view of the Java programmer. (Coherence may therefore be implemented in hardware or the OS.) Memory access speeds are still unknown and variable.

```
// Step through the patterns of the platform
// to find an SMP to create a Locale on
Platform p = Platform.getPlatform();
NUMA numa = p.getRootLocation();
CCNUMA ccnuma = numa.getChildren()[0];
SMP smp = ccnuma.getChildren()[0];
Locale locale = new Locale(smp);

//Create threads inside the new locale
int ncpu = smp.getNumCPUs();
for (int i = 0; i < ncpu; i++) {
  Thread th = locale.createJavaThread(() -> {
    //...
  });
  th.start();
}
```

Fig. 2. Simple locality and architecture mapping in the JUNIPER approach.

- SMP: The SMP pattern represents an architecture in which components are more tightly-coupled. Access times to memory are uniform within a reasonable error bound. Variation is only due to contention on a shared memory bus or cache effects, not because memory is a greater 'distance' from the processors.

The programmer must manually create locales, and map them to the patterns of their host architecture. An example of this is shown in figure 2. These patterns also allow architecture discovery, in which the programmer can request information about the hardware such as number of processors, cache size, etc. This information can be used to tailor the algorithm selected and amount of parallelism expressed.

Locales communicate using the JUNIPER communications API. This API provides a range of high-level communication features to facilitate automatic deployment, group communications, and code portability, but the details are outside of the scope of this paper. Communications in JUNIPER are implemented using MPI, and the lowest level features of the API are similar to MPI calls.

Use of locales and patterns has been shown to both slightly increase average performance, but also to increase the predictability of execution times (due to the greater control over placement and therefore lower pessimism of memory latency and cache usage) [14].

In the context of this work, locales are important because they are the unit of FPGA acceleration in the JUNIPER approach.

B. Acceleratable Locales

A common problem with general-purpose acceleration of a high-level language such as Java is that it can be difficult to determine which parts of the application should be accelerated for the largest gain. In the JUNIPER platform the programmer has already expressed their application in terms of groups of closely-coupled threads and data (locales) and so it is these which are focussed upon.

The developer is required to identify the locales within the application that are amenable to static acceleration

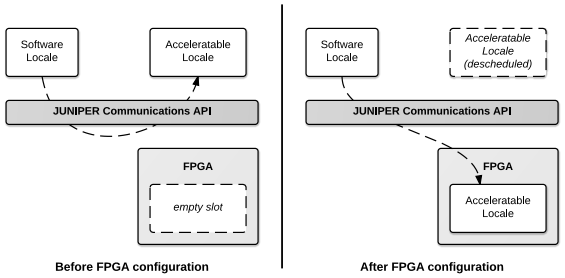

Fig. 3. The JUNIPER API performs communication redirection.

on the FPGA. 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 that will be allocated inside that locale. (Normal locales can allocate freely, subject to the normal RTSJ restrictions). A `Locale` cannot be created inside `AcceleratableLocale.initialise()` but an `AcceleratableLocale` may be. This restriction ensures that the accelerated locale can have its structure analysed to aid the hardware tool flow. General memory allocation (`new`) is still supported but should be avoided. (This is discussed further in section IV-C).

`AcceleratableLocales` *may* be moved to the FPGA co-processor, but this is not required. In systems without FPGAs they execute as normal Java locales. Equally, an `AcceleratableLocales` may migrate between hardware and software dynamically as the system executes (see section V).

The second restriction on `AcceleratableLocales` is that all communications with the `AcceleratableLocale` must use the JUNIPER communications API. In the general purpose JUNIPER model this is not strictly required. Locales are normal Java code and so they may simply share heap data, but if they do this then the sharing locales are restricted to existing within a ccNUMA or SMP pattern, and must be present on the same JVM. With `AcceleratableLocales` this is not permitted. All communications must be via the MPI-like JUNIPER API. This allows Locales on remote nodes to communicate with an accelerated locale, and also allows the transparent migration of `AcceleratableLocales` between software and hardware, as shown in figure 3.

IV. Implementation Strategy

This implementation section covers three main issues. First, the architecture of the FPGA system is detailed in section IV-A. Then the way that this architecture interacts with the host server is described in section IV-B. Finally, the actual task of translating Java code to an FPGA component is covered in section IV-C.

A. FPGA architecture

The structure of the FPGA architecture is shown in figure 4. The structure is as follows:

- *Static section*: The same for all JUNIPER designs, contains:

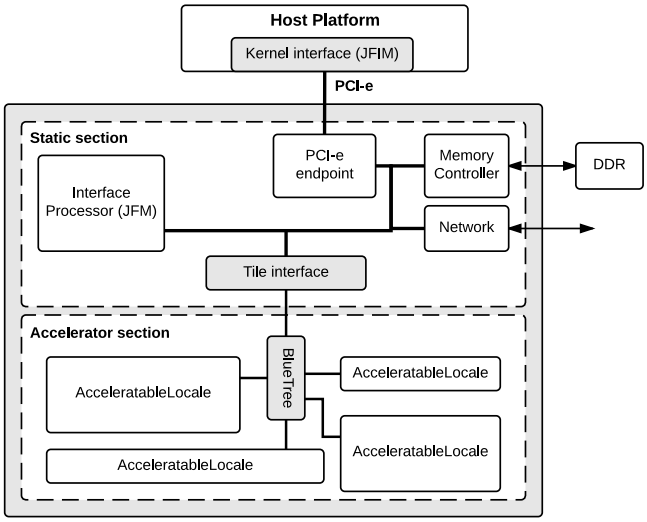

Fig. 4. The structure of the FPGA design.

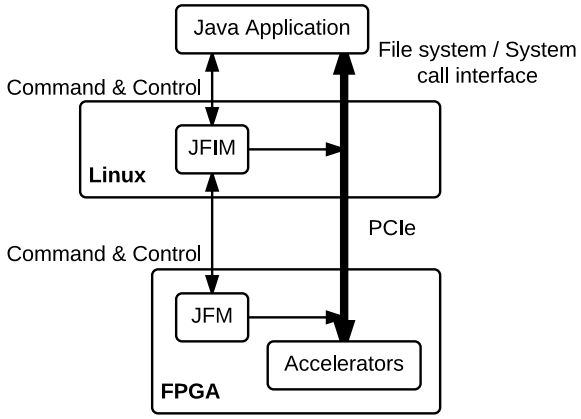

Fig. 5. The OS support for FPGA acceleration.

- A PCI-express endpoint to implement communication between the host platform and the FPGA (more details in section IV-B).
- An external memory controller to connect to external DDR memory, into which the host platform will DMA copy data for the accelerators to process.
- An interface processor called the JUNIPER FPGA Manager (JFM) which performs on-FPGA management.

- *Accelerator section*: This section varies depending on the accelerators located in the design. It consists of a set of translated `AcceleratableLocale`s (section IV-C) connected via a unique tree-based NoC called BlueTree [15]. BlueTree is a configurable NoC interconnect with a tree-based structure that is designed to optimise the connection of many clients to external memory in a real-time or safety-critical system. It supports the use of worse-case aware prefetching [16], and memory and bandwidth partitioning to assist in the development of mixed-criticality systems [17].

The accelerator section is treated as a single large tile, rather than individual tiles for each accelerator. This simplifies the design and reduces the overhead (both area and communication time) of interconnect standards between the tiles, at the cost of having to create an accelerator tile with a fixed set of accelerators for the given application. Section V discusses a more dynamic acceleration design.

The JUNIPER platform is currently being developed on Xilinx Spartan 6, Virtex 7, and Zynq FPGA devices, so the JFM is the Xilinx Microblaze softcore processor [18].

B. Interactions with host

The host server uses the accelerators on the FPGA with two key infrastructure components:

- A JUNIPER kernel module called the JUNIPER FPGA Interface Module (JFIM) implements zero-copy transfers between the Java application and the FPGA. It is required because the application (resident in user address space) cannot directly access the FPGA hardware, nor directly call drivers within the Linux kernel that access hardware.
- As discussed in section III, the JUNIPER programming model mandates the use of a communications API to send data to and from an accelerator. The JUNIPER library implementation of this API uses the JFIM to move data as efficiently as possible.

This is shown in figure 5. The JFIM offers a high level driver for the Java application (in user space), with the JFIM leveraging the lower level (hardware dependent) PCIe drivers within the host Linux OS to communicate with PCIe support on the FPGA board, which in turn interfaces with the JFM on the FPGA board.

The approach described is not specific to Java as it provides generic facilities that can be used by any user level process on the host, however programmed. The purpose of the command and control interface (CCI) is to provide an interface to the JFIM that is accessible from the application process in user memory space. The Linux device driver architecture allows application processes to access devices via the file system (i.e. `/dev` and `/sys` directories, `open()`, `read()`, `write()`, and `ioctl()` system calls).

A summarised list of the features exposed by the JFIM are:

- `JFIM_get_resource_map()`: returns the resource map of the FPGA as negotiated between the JFIM and JFM. This information is exposed to the application through the JUNIPER API.
- `JFIM_load_design(file)`: load of an accelerator tile into the FPGA.
- `JFIM_reset_accelerator(to)`: reset an accelerator on the FPGA.
- `JFIM_start_accelerator(to)`: start an accelerator in the FPGA.
- `JFIM_stop_accelerator(to)`: stop an accelerator running.
- `JFIM_write_memory(from, to, offset, length)`: effect a write to FPGA memory.
- `JFIM_read_memory(from, to, offset, length)`: effect a read of FPGA memory.
- `JFIM_control(to, addr, len)`: send a control message to the target accelerator.

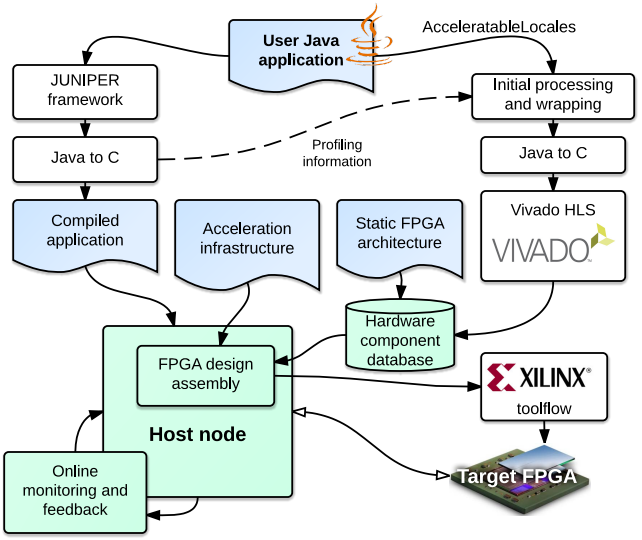

Fig. 6. The hardware translation toolflow.

```
jamaica_int32 jam_comp_data_method(
    jamaica_thread *ct,jamaica_ref r1) {
 jamaica_int32 tp=ct->m.cli+0;
 if(tp > ct->javaStackSize) {goto LABEL_tSOE;}
 n4=0;
 ...

/* becomes... */

#include <xparameters.h>
#include <xhls.h>
XHls acc;
XHls_Initialize(&acc, XPAR_HLS_0_DEVICE_ID);

jamaica_int32 jam_comp_data_method(
    jamaica_thread *ct,jamaica_ref r1) {
 XHls_SetBase(r1); //Pass in the base address
 XHls_Start(&acc);
 while(!XHls_IsDone(&acc));
 return XHls_GetReturn(&acc);
}
```

Fig. 7. The software implementation is translated to use the accelerated method.

C. Translation of Java to HDL

The JUNIPER toolflow translates Java code into a hardware description language (HDL) representation for implementation on the FPGA. The toolflow used is shown in figure 6.

For normal Java code in the JUNIPER approach, the input Java is translated to C for native compilation (or interpretation) by a real-time JVM called JamaicaVM [19]. This approach supports both standard Java and real-time Java, and allows for more predictable real-time behaviour (including real-time garbage collection).

JamaicaVM, as part of its compilation process (and similar to the standard Java HotSpot JIT compiler [20]), profiles the target application to determine frequently-used code. This information is used to inform which `AcceleratableLocales` should be focussed upon for hardware implementation. This is because it has been observed by both the developers of Java and of JamaicaVM that it is actually fastest not to natively compile the entire of the application's bytecode. Bytecode tends to be more compact than machine code, so interpreting low frequency methods results in less pressure on the instruction cache. High frequency methods are compiled. The same effect is observed when translating to hardware so JamaicaVM's profiling can be reused.

Locales that make up the bulk of the application's computation time are translated from C to HDL using a Xilinx tool called Vivado HLS. This produces an archive of hardware components, one for each `AcceleratableLocale` in the system. A separate JUNIPER tool then takes a set of these components and inserts then into the static FPGA architecture (discussed in section IV-A) to create a final FPGA design.

When translating a Java locale to hardware, the following issues must be considered:

- Unlike many existing systems, the tool flow does not attempt to turn all features of Java into hardware. Tasks like memory allocation, garbage collection, and low-frequency methods are left as software and executed on a processor on the FPGA. This could either be a soft processor, or an embedded processor such as an ARM core in the Zynq. Methods which perform a lot of allocation are unlikely to benefit from acceleration.
- Translation uses the profiling information from JamaicaVM to translate on a method-by-method basis, starting with the most frequently-used methods.
- VM calls (native methods) are not translated.
- The target method is wrapped in a top level C function that implements an AXI bus interface. This creates an IP core which can be placed on an AXI bus, can communicate with memory (via the BlueTree NoC, see section IV-A), and be communicated with from the JFM. These directives are shown in figure 8 and the resulting hardware in figure 9.
- Once a method has been successfully translated, it should be used rather than the software implementation (which is C, translated from the input Java). To do this, the C implementation of the translated method is replaced with driver calls to execute the accelerator. This is shown in figure 7. The IP drivers are automatically generated by Vivado HLS.
- Exceptions are currently not supported inside translated methods. JamaicaVM implements exception handling with `setjmp` and `longjmp` which are not available in VivadoHLS. General exceptions are inefficient in hardware as they add an exponential number of potential state transitions to the state machine of the hardware. Limited schemes have been implemented, but this remains further work.
- Communications from accelerated methods are not accelerated and must be executed as software.

The JamaicaVM memory model has been extensively adjusted for hardware implementation. JamaicaVM models memory as a set of *blocks*, which are the unit of work for its real-time memory allocator and garbage collector. These blocks are variable size, but may be accessed as appropriately-

```
volatile jamaica_data32 *__juniper_ram_master;
jamaica_thread t;

void hls(volatile jamaica_data32 *master,
  int *baseaddr) {
 #pragma HLS RESOURCE variable=master
   core=AXI4M
 #pragma HLS RESOURCE core=AXI4LiteS
   metadata="-bus_bundle slv0"
   variable=baseaddr
 #pragma HLS RESOURCE core=AXI4LiteS
   metadata="-bus_bundle slv0"
   variable=return
 #pragma HLS INTERFACE ap_bus port=master
 #pragma HLS INTERFACE ap_none
   port=baseaddr register
 create_jamaica_thread(&t);

 __juniper_ram_master = master;
 jam_comp_dataProcessing_fir(&t,
  (jamaica_ref) baseaddr)
}
```

Fig. 8. The top level function uses HLS directives to define the accelerator's interface

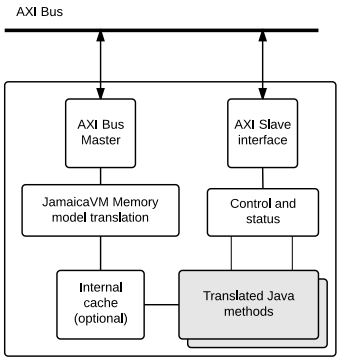

Fig. 9. The resulting IP core from the top level in figure 8.

sized arrays of any of the base C types (e.g. as 4 `int`s, or 8 `short`s etc.). Unfortunately this approach is not compatible with Vivado HLS because this requires pointer reinterpretation (an `int *` pointer to be cast to a `short *` for example) and this is not supported.

The solution implemented is to view the entire memory space as an `int *` pointer and add access functions (in the wrapper C) to read sub-words accordingly. This slightly duplicates normal bus control logic (because AXI can natively support such sub-word reads) but it avoids the limitation of Vivado HLS and allows the JamaicaVM output code to remain unaffected.

V. Dynamic Acceleration

Due to space constraints on the FPGA, most of the time it will not be possible to offload all `AcceleratableLocale`s to the FPGA simultaneously. The question becomes, therefore, *which* Locales should be offloaded and *when*? The static approach defines this ahead of time and the chosen set of Locales remains static throughout the execution of the system. This means that the behaviour of the system is more predictable due to the fact that allocation and configuration is performed ahead of time, and the system is less complex as less supporting infrastructure is required. However it has drawbacks:

- Only a fixed subset of the application can be accelerated, which can limit the applicability of the approach.
- The designer must analyse the system ahead-of-time to determine how to best use the FPGA to get the highest benefit.
- The static approach is not transparent. The designer is required to use FPGA development tools and to understand relevant tools, techniques, and design trade-offs.
- The static approach cannot deal with unseen code (no dynamic code loading).
- Static offloading requires a fixed deployment to FPGA-equipped nodes. Applications cannot merely use FPGAs if they exist and ignore them if they are not available.

JUNIPER is therefore developing a dynamic acceleration approach to make the acceleration transparent to the developer by removing the requirement for ahead-of-time system analysis. It aims to allow the system to dynamically discover at runtime a suitable selection of `AcceleratableLocale`s to place on the target FPGA without programmer intervention, at the cost of some level of real-time guarantees. This is done through a combination of online performance monitoring (already part of the JUNIPER framework) and online FPGA compilation.

A. Partial Dynamic Reconfiguration

In the dynamic approach, rather than reprogramming the entire target FPGA, Partial Dynamic Reconfiguration (PDR) is used. PDR allows *partial bitfiles* to reprogram a section of the running device without interrupting the rest of the system. This is crucial for the JUNIPER system, because the JFM and bus interfaces must remain uninterrupted. A common use of PDR is to time-slice a large amount of FPGA logic onto a device that would not otherwise fit by 'swapping' tiles of functional units in and out of the device. This has been demonstrated in an automotive project to fit a range of automotive control units into a single device. [21]

When using the vendor-supported tools, FPGA partial bitfiles are not 'relocatable' and must be regenerated for every possible target location on the device. In simple devices (such as the Virtex 2) it was quite simple to relocate modules but each successive FPGA generation made this problem harder. In modern FPGAs (Virtex 7 and Ultrascale) the bitfile format is a proprietary and largely undocumented 'black box', and the architecture itself is highly irregular. Some papers have proposed schemes to work around this restriction [22] and these approaches show promise. Given the target domain of the JUNIPER project, we avoid the necessity for such schemes because we argue that we can perform the reconfiguration on-demand using a node from the cluster.

Accordingly, the JUNIPER dynamic approach regenerates the whole accelerator section of the FPGA architecture (figure 4) as a single tile. A set of accelerators is selected, they are packaged with the BlueTiles NoC into a tile design, and the FPGA vendor tools executed online to create the partial bitfile. This design is shown in figure 10.

TABLE I. COMPARISON BETWEEN JAMAICAVM AND HAND-DEVELOPED C WHEN SYNTHESISED WITH VIVADO HLS

Function	Hand-developed C + HLS		Java + JamaicaVM + HLS	
	LUTs	Latency	LUTs	Latency
Vector sum (500 items)	113	507	175	511
Collatz evaluation (fixed input)	293	278	383	282
MD5 hash evaluation	1675	3463	272	676
FIR filter	298	183	283	121

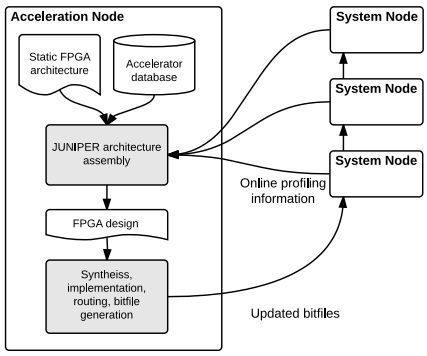

Fig. 10. The dynamic acceleration flow.

Work on the JUNIPER dynamic approach is in the early stages, with the aim to deliver at the end of the project.

VI. PRELIMINARY RESULTS

As this represents work in progress it is not possible to test entire applications yet, only relatively small filters and methods. This will be addressed as development of the JUNIPER platform matures. Consequentially, this section focusses on the areas that can currently be evaluated.

An important area of concern for the project is whether the use of Java introduces unacceptable overhead or not. Table I shows a comparison of a range of functions implemented as C code and then synthesised to hardware with Vivado HLS, against Java method implementations of the same functions that are translated through JamaicaVM first, before using HLS. In the table, *LUTs* (lookup tables) is a measure of FPGA area used. The toolflow also reports an estimate for equivalent flip flop usage but the ratios are the same as for LUTs. For comparison, the Microblaze processor uses between 770–1154 LUTs on the Spartan 6 and Virtex 6 FPGA fabrics, depending on how it is configured. *Latency* is measured in clock cycles of the target FPGA, and is the speed for a single data item to complete a pass through the function. The FPGA runs at 200MHz in these tests.

The surprising point of note is that in some cases the version of the hardware that has passed through JamaicaVM first is smaller and faster. This is the case because all of these numbers are before any hand-optimisation of HLS directives. HLS's default behaviour can frequently be improved by adding directives to unroll loops (at the cost of LUTs) or similar. In the case of MD5, the largest such discrepancy, hand unrolling and function inlining can reduce the hand-developed C version to be similar in size and speed to the JamaicaVM version, but this requires specialist knowledge. A lot of the extra LUTs are in multiplexors which are added because of the C version's use of pointers, something which is removed when the same code has passed through `javac` and JamaicaVM.

It appears that passing code first through JamaicaVM can have a range of advantages:

- The Java code is subjected to the Java compiler's extensive static analysis to produce bytecode, and then to JamaicaVM's range of code optimisations.
- JamaicaVM translates Java bytecode, which is a smaller input language than the largely unrestricted C that Vivado HLS normally uses.
- The Jamaica VM output C code is very simple as a result. It is a direct state machine representation of the JVM operations for the target method, which translates into hardware very easily.
- When using hand-developed C code, developers tend to make use of techniques which are very efficient in software but not as amenable for hardware. The MD5 code, for example, uses a lot of pointer arithmetic and buffer slicing. Vivado HLS can work with this, but it clearly doesn't produce optimal hardware without further intervention.

Other tests which have simpler input C code in the hand-developed version show a slight improvement in the C version, which is more what was expected. In these cases however the difference is only small, demonstrating that the use of Java does not appear to be an impediment to the quality of the generated hardware, and may in fact help as the code becomes more complex.

In all of these results we can see that the generated hardware has specific latency value, rather than a range. Clearly depending on the algorithm the inputs can change its execution time, but with fixed inputs we can be certain down to the clock cycle about how long a piece of hardware will take to execute. Uncertainty can be introduced through memory latency or bus/network latency as with software implementations. Hardware has the advantage that when execution time accuracy is vital, data can be cached into the hardware to remove this uncertainty, at the cost of FPGA memory resources. This has not been discussed in this paper and will be expanded upon in future work.

These results show that there is potential for the JUNIPER acceleration approach. More evaluation will be performed as the platform develops.

VII. FUTURE WORK

As this represents ongoing work, there are a range of issues that are still to be resolved. In general C to HDL translation the end results can be hugely variable. This problem is ameliorated

in JUNIPER because we are not working with unknown C, but rather the output of JamaicaVM. However, there is still design-space exploration that needs to be performed to obtain the best quality hardware. Specifically, the treatment of memory is very important. Vivado HLS uses a default implementation but this can be overridden using directives. A simple approach could try different combinations of directives, at the cost of accordingly increased compilation time.

Related to the above point, the accelerators can operate even faster if, instead of copying their input data to DDR, the data is copied directly into memories inside the accelerator itself. This takes up a large amount of valuable on-FPGA memory, but would be a useful optimisation to implement for high-priority accelerators.

It would be useful to be able to feed back information on expected performance and space tradeoffs when using the acceleration process. Unfortunately this is a common problem when using FPGAs because the synthesis and implementation of an FPGA design can take a very long time. Given the more restricted subset of designs that will be used in the JUNIPER approach, it might be possible to generate approximate metrics ahead of time.

VIII. Conclusion

The JUNIPER platform is an approach to building the next generation of Big Data systems which can provide design-time guarantees about their response times and performance metrics. To do this, the platform includes a range of real-time technologies. This paper focusses on the acceleration of application-level Java code to FPGAs.

Rather than attempt to accelerate general purpose Java, only specific sections of the application (called locales) are translated. Locales are expressed by the programmer using the JUNIPER programming model. This eases the problem because locales are tightly-coupled collections of threads and data. This means that the complex task of determining what to offload is passed to the programmer, but in a way which is not onerous. Locales also provide other benefits in the JUNIPER platform, allowing greater control over scheduling, resource allocation, and deployment though the target cloud or supercomputer. Finally, communications in and out of locales are restricted to use the MPI-based JUNIPER communications model meaning the programmer does not need to worry about manually implementing Java to FPGA communications.

Initial results show that the use of Java to accelerate software does not add significant overheads, and in fact when code becomes more complex and 'C-like' the JUNIPER toolflow can give better results unless manual expertise is then applied. It also provides tighter execution time estimates. This paper describes the work currently under way, the approach being developed, and presents some preliminary results that demonstrate the promise in the technique.

References

[1] "Apache Hadoop Website," http://hadoop.apache.org/, 2011.

[2] Apache Software Foundation, "Apache Storm – Distributed and fault-tolerant realtime computation," http://storm.incubator.apache.org/.

[3] Apache Software Foundation, "Apache Spark – Lightning-Fast Cluster Computing," http://spark.apache.org/.

[4] Oracle Corporation, "JDK 8," http://openjdk.java.net/projects/jdk8/, September 2013.

[5] J. Gosling and G. Bollella, *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[6] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch, "Rootbeer: Seamlessly using gpus from java," in *High Performance Computing and Communication 2012, IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012*. IEEE, 2012, pp. 375–380.

[7] The HotSpot Group, "Project sumatra," http://openjdk.java.net/projects/sumatra/, November 2014.

[8] J. M. Cardoso and H. C. Neto, "Towards an automatic path from java tm bytecodes to hardware through high-level synthesis," in *Electronics, Circuits and Systems, 1998 IEEE International Conference on*, vol. 1. IEEE, 1998, pp. 85–88.

[9] T. Kuhn and W. Rosenstiel, "Java based object oriented hardware specification and synthesis," in *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '00. New York, NY, USA: ACM, 2000, pp. 579–582. [Online]. Available: http://doi.acm.org/10.1145/368434.368809

[10] M. Wirthlin, B. Hutchings, and C. Worth, "Synthesizing rtl hardware from java byte codes," in *Field-Programmable Logic and Applications*, ser. Lecture Notes in Computer Science, G. Brebner and R. Woods, Eds. Springer Berlin Heidelberg, 2001, vol. 2147, pp. 123–132. [Online]. Available: http://dx.doi.org/10.1007/3-540-44687-7_13

[11] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Panainte, "The molen polymorphic processor," *Computers, IEEE Transactions on*, vol. 53, no. 11, pp. 1363–1375, Nov 2004.

[12] M. Schoeberl, "JOP: A Java Optimized Processor for Embedded Real-Time Systems," Master's thesis, Technischen Universitat Wien, 2005.

[13] I. Gray, Y. Chan, N. C. Audsley, and A. Wellings, "Architecture-awareness for real-time big data systems," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: ACM, 2014, pp. 151:151–151:156. [Online]. Available: http://doi.acm.org/10.1145/2642769.2642798

[14] Y. Chan, A. Wellings, I. Gray, and N. Audsley, "On the locality of java 8 streams in real-time big data applications," in *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, ser. JTRES '14. New York, NY, USA: ACM, 2014, pp. 20:20–20:28. [Online]. Available: http://doi.acm.org/10.1145/2661020.2661028

[15] G. Plumbridge, J. Whitham, and N. Audsley, "Blueshell: A platform for rapid prototyping of multiprocessor nocs and accelerators," *SIGARCH Comput. Archit. News*, vol. 41, no. 5, pp. 107–117, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2641361.2641379

[16] J. Garside and N. C. Audsley, "Wcet preserving hardware prefetch for many-core real-time systems," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. ACM, 2014, p. 193.

[17] N. C. Audsley, "Memory architectures for noc-based real-time mixed criticality systems," *Proc. WMC, RTSS*, pp. 37–42, 2013.

[18] Xilinx Corporation, "Microblaze Processor Reference Guide," vol. UG081 v13.2, 2011.

[19] F. Siebert, "Realtime garbage collection in the jamaicavm 3.0," in *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems*, ser. JTRES '07. New York, NY, USA: ACM, 2007, pp. 94–103.

[20] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, "Design of the java hotspot; client compiler for java 6," *ACM Trans. Archit. Code Optim.*, vol. 5, no. 1, pp. 7:1–7:32, 2008.

[21] J. Becker, M. Hubner, K. D. Muller-Glaser, R. Constapel, J. Luka, and J. Eisenmann, "Automotive control unit optimization perspectives: Body functions on-demand by dynamic reconfiguration," in *Design, Automation and Test Eur. Conf. Exhibition (DATE 2005)*, 2005.

[22] C. Beckhoff, D. Koch, and J. Torresen, "Portable module relocation and bitstream compression for xilinx fpgas," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, Sept 2014, pp. 1–8.