Project Number 318763

D2.6 – Dynamic Acceleration Design

Version 1.0
2 June 2014
Final

Public Distribution

University of York

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	20 May 2014
0.2	Complete First Draft	29 May 2014
1.0	QA for EC delivery	2 June 2014

Executive Summary

This document constitutes deliverable *D2.6 – Dynamic Acceleration Design* of work package 2 of the JUNIPER project. It details the JUNIPER dynamic acceleration design. This system builds on previous deliverable D2.3, *Static Acceleration Design* in which a static FPGA acceleration system was defined. In the static system, sections of the input Java code are marked for implementation on an attached FPGA. These sections are compiled to hardware description language code and built into an FPGA configuration file which is programmed onto the FPGA before system boot.

The dynamic approach removes the limitation that the application code be compiled ahead of time. It instead uses a range of performance metrics that are made available by the JUNIPER framework to determine at runtime a suitable subset of the application to offload. The dynamic system runs periodically, constantly re-evaluating the state of the system to determine if a better offloading can be obtained.

This document defines the design of the dynamic acceleration system, and will be followed by deliverable D2.7, *Dynamic Acceleration Implementation and Assessment* in which the implementation of the system will be detailed and evaluated.

1 Introduction

Note: This document builds on the static acceleration design described in deliverable D2.3 and will assume that the reader is familiar with its contents.

The JUNIPER static acceleration design aims to accelerate key parts of the target application by compiling them to hardware and programming them onto an FPGA attached to the host computing node. In the static design, this is performed on the granularity of `Locales`. Locales are discussed in deliverable D2.1 and D2.2 which describe the JUNIPER programming model, but to recap, a Locale is a software-level object in the JUNIPER programming API. Locales are created explicitly by the programmer and can be used to create threads and allocate data storage. Locales can also be bound to parts of the target architecture (such as individual CPUs, or groups of CPUs). This gives *locality* to the threads and data of the Locale, and means that threads will only be scheduled on the CPUs of the Locale, and data will be allocated in the memories of the Locale.

Locales are therefore a programmer-defined group of tightly-coupled threads and data items which are likely to receive a large benefit from acceleration (due to the relatively low amount of inter-locale communication when compared with intra-locale communication).

Not all Locales are suitable for offloading to the FPGA. For this purpose `AcceleratableLocale` was defined, which is a restricted subset of `Locale`. Only Locales that are actually an `AcceleratableLocale` can be compiled to the FPGA. This remains true in the dynamic approach described in this document. The actual process by which a Locale is converted to hardware description language code for implementation on the FPGA is described previously in deliverable D2.3. This document builds on this work to discuss the need for a dynamic approach to FPGA acceleration.

1.1 Dynamic Acceleration

Due to space constraints on the FPGA, most of the time it will not be possible to offload all `AcceleratableLocales` to the FPGA simultaneously. The crucial question, therefore, is *which* Locales should be offloaded and *when*. The static acceleration approach defines this ahead of time. The chosen set of Locales remains static throughout the execution of the system. This has the following benefits and drawbacks.

- Benefits
  - The behaviour of the system is more predictable due to the fact that allocation is performed ahead of time.
  - The system is less complex as less supporting infrastructure is required.
  - The time taken to configure the FPGA does not need to be considered when performing scheduling analysis.
- 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 tradeoffs.
- 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.

The dynamic acceleration approach attempts 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 an optimum selection of `AcceleratableLocale`s to place on the target FPGA without programmer intervention. This is done through a combination of online performance monitoring and online FPGA compilation.

Section 2 briefly outlines the potential design options that are available, before describing the chosen design in section 2.2 and the final toolflow in section 2.6.

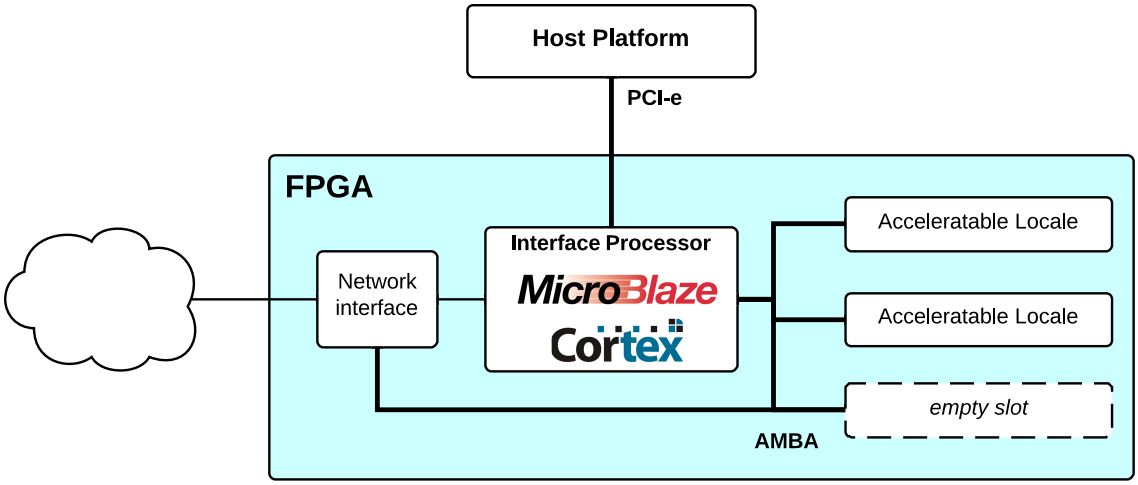

Figure 1: The layout of *AcceleratableLocale*s on the target FPGA.

2 Design

2.1 FPGA Layout

Before describing the available design choices, it is useful to recall the structure of the FPGA designs used in the JUNIPER project. An overview is shown in figure 1. As can be seen, much of the FPGA design is fixed. A PCI-e endpoint hardware module is instantiated to connect to the host computer, and interface processor is used to receive commands over this bus. Recall from deliverable D3.4, that a kernel module in the host operating system called the JUNIPER FPGA Interface Module (JFIM) is responsible for managing this interface, sending programming commands and transferring data into and out of the FPGA. On the FPGA, the interface processor that listens for commands from the JFIM is called the JUNIPER FPGA Manager (JFM). This may be either a hard processing core (as with the ARM Cores on the Xilinx Zynq-7000 platform [14]) or a soft processing core (the Xilinx Microblaze [13] on all other FPGAs).[1] The JFM uses an internal AMBA bus to communicate with a set of 'tiles'. These tiles are the `AcceleratableLocales` of the input application that have been selected for offload to the FPGA.

D3.4 goes into greater detail about how communications are implemented between Locales on the host computer and on the FPGA, this document will instead focus on how the FPGA configuration files (called bitfiles) are produced and when.

2.1.1 Partial Dynamic Reconfiguration

FPGAs have the unique ability to be reprogrammed 'in the field' to change their function. Commonly the entire device is reprogrammed, but modern FPGAs allow reprogramming at much finer tile-based granularities. It is possible to change small parts of the the FPGA's design whilst the rest of the system is still operating, with a technique called Partial Dynamic Reconfiguration (PDR). PDR is not

[1] The JUNIPER project uses Xilinx FPGAs because of the availability of high-level synthesis tools.

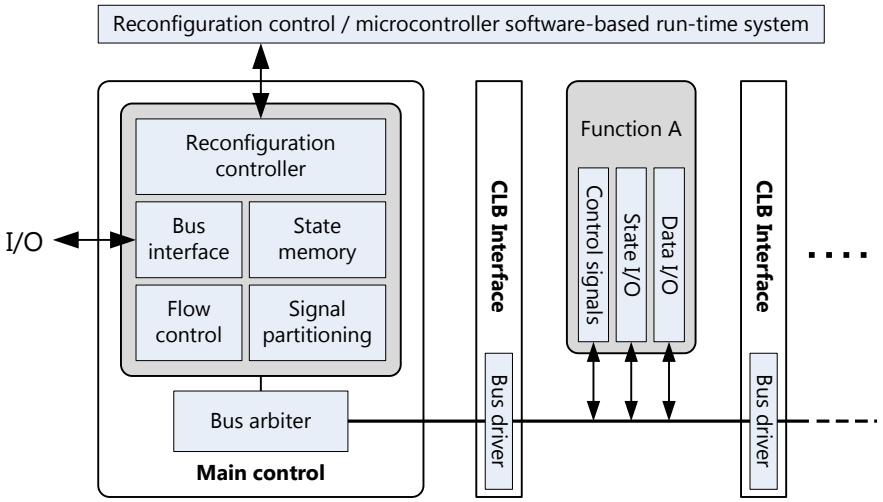

Figure 2: Partial dynamic reconfiguration - tiles of the design (Function A) can be swapped in and out of the running design. Tiles are connected to a static bus so that they can be communicated with after configuration is complete.

commonly used in industry due to poor tool support and the vastly increased complexity of the final design, but it is a powerful technique which has been shown to provide some unique benefits. It is discussed here, and used in the JUNIPER approach in a constrained way.

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' functional units in and out of the device. This has been used in an automotive project to fit a range of automotive control units into a single device, as shown in figure 2. [12, 3]

A difficulty of such systems is that traditional on-chip buses fail when parts of them are swapped in and out of the FPGA. Metastability is a large problem, as the state of the FPGA is unpredictable as it is undergoing reconfiguration, and it is very easy for the entire state of the running system to become corrupt. Network on Chip-based PDR designs have been proposed [8, 7] to avoid this. These systems work by 'hooking' rectangular reconfigurable tiles onto a static network layout as they are configured into the device. Essentially, a static grid-based network of interconnect is laid down on the chip with router components placed at regular intervals. This may be in one or two dimensions. This part of the design is static and is not affected by run-time reconfiguration. Most of the routers are not connected to a tile and so sit idle, but when the system wants to swap in a hardware tile it can select a free router and 'hook' the incoming tile off the selected router. If the tile is very large it may overwrite a number of other routers in the surrounding area, but the grid layout will be maintained. This way, incoming tiles can always communicate with each other, as shown in figure 3.

In all of this work, there is one very large problem. FPGA partial bitfiles are not 'relocatable', and must be regenerated for every possible target location on the device. In a large device this may be hundreds of targets. Unfortunately Xilinx's bitfile format is both proprietary and encrypted, and so all of the cited work was only made possible through direct collaboration with Xilinx. Recent work [4] has considered this problem to attempt a 'bitstream filter' to give the illusion of relocatable bitfiles, but the work is not mature enough for use.

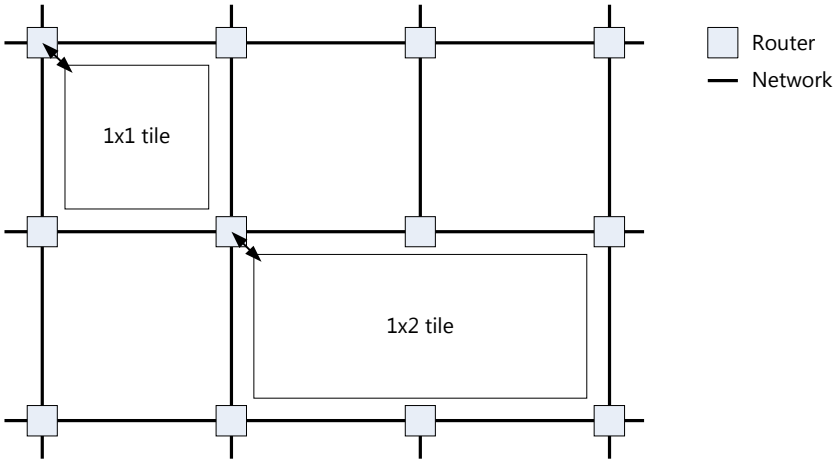

Figure 3: Network-based designs have been proposed to avoid some of the limitations of standard PDR. A traditional network maintains routing and connectivity whilst tiles are swapped in and out.

The advantages of a PDR-based system are clear. Given the FPGA architecture detailed in section 2.1, PDR could be used to swap individual Locales in and out of the FPGA, which allows for far greater responsiveness than reconfiguring the entire FPGA. (Reconfiguration times of FPGAs are proportional to the area being reconfigured.) This would make the design more responsive to changing demands from the software environment. However, PDR also imposes a very high cost. The complexity of a PDR design is many times greater than a static one, and there are still a range of unsolved problems that must be overcome (primarily the lack of relocatable bitfiles). Finally, reconfiguring the entire design is problematic as this will disrupt the operation of the I/O and RAM controllers in the device. DDR memory is not 'hotswappable', so disconnecting and reconnecting it (which is the effect of reprogramming the FPGA) can result in undefined behaviour or error states. Similar effects have been observed with PCI-e, even though it is supposed to be resilient to this.

2.2 Selected Design

The chosen solution is to use a partially-dynamic system which can obtain many of the benefits of PDR without imposing too much of a complexity penalty. As shown in figure 4, I/O and other interfaces are statically placed and routed, and remain fixed throughout the lifespan of the application. The rest of the FPGA is configured using PDR as a single reconfigurable tile. Locales are not individually swapped in and out (because of the complexity of bitfiles lacking relocation). Instead when the system wants to change the offloaded locales, it must select a new set and rebuild an entire new tile. This tile is then swapped for the previous tile, replacing all locales simultaneously, but leaving the I/O and other static elements unaffected.

As described previously in section 1, the purpose of the dynamic acceleration design is to remove the requirement for direct developer involvement in the FPGA acceleration, but it retains the requirement that suitable parts of the input software are marked as acceleratable through the use of `AcceleratableLocale`. The system therefore uses metrics to determine which locales to offload and which to not, which are discussed later in section 2.5.

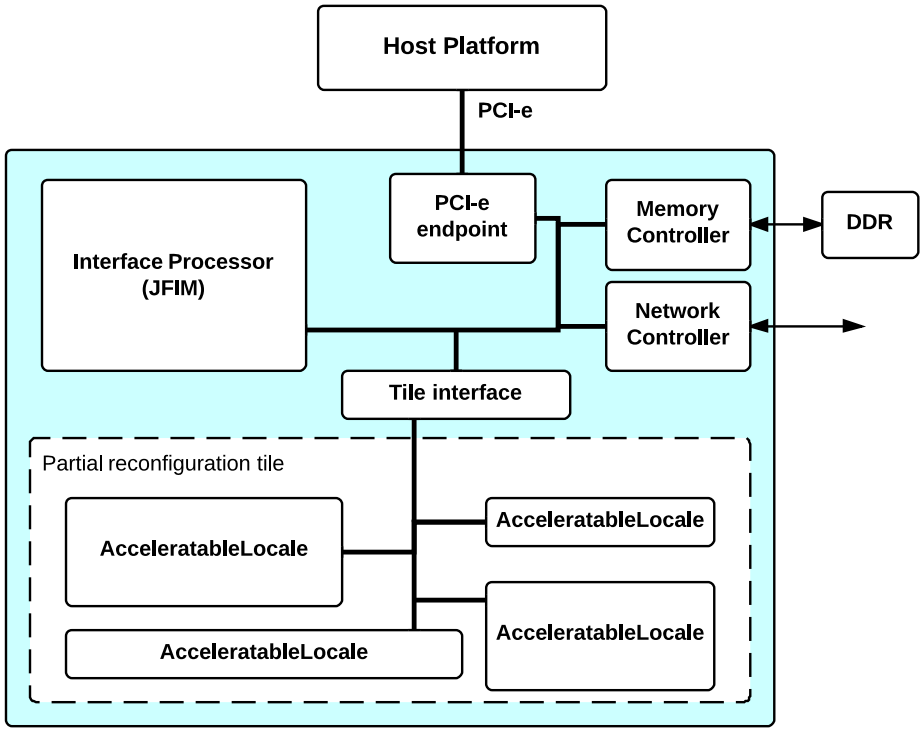

Figure 4: The chosen solution is partially reconfigurable. I/O and other interfaces remain static whilst the rest of the design is reconfigured using partial dynamic reconfiguration.

The high-level overview of the approach is as follows:

1. Before the system begins, all `AcceleratableLocales` are converted from Java into hardware components. The process of converting the Java code for a Locale into hardware is called *high-level synthesis* (HLS).
2. The HLS process involves the following stages (and is the same as for the static acceleration design):
   (a) Java Code Preparation: The Java code must be transformed into a format which is suitable for passing through JamaicaVM [1] (the JUNIPER Java to C tools).
   (b) Java to C: JamaicaVM converts the Java code into C code.
   (c) Java Code Preparation: The output of JamaicaVM is not directly suitable for HLS, and so must again be transformed.
   (d) High Level Synthesis: The Xilinx tool Vivado HLS is used to convert the transformed C code into a hardware description language implementation.
3. The system begins executing as a software (RTSJ) application. All Locales execute in their bound locations as described by the programmer.
4. After system start up, metrics are collected (see section 2.5) which record which `AcceleratableLocales` are being used most commonly.

5. From this information, the migration infrastructure begins constructing an reconfiguration tile for the FPGA design. This consists of a set of hardware implementations of `AcceleratableLocale`s, prepared earlier.

6. The prepared tile must be run through the FPGA design flow. This involves synthesis, mapping, place and route, and bitfile generation. This requires FPGA vendor-specific tools. Note that it is most appropriate for these tools to run on a single 'tool server' in the cloud, rather than attempting to support the tools on every cloud node. The specific details of this are dependent on the specific cloud deployment.

7. Once the bitfile is prepared, it is dynamically programmed on to the FPGA, replacing the old locales that were present but leaving I/O, the JFM, and other static parts of the design unaffected.

8. The JUNIPER communications library can then redirect all communications bound for an offloaded Locale to the FPGA (through the JFIM and JFM) transparently.

9. This process is then repeated after a given period of time.

The allocation loop must run repeatedly because it is not in general possible to, ahead of time, determine that a given offloading pattern will be the most beneficial possible. Instead heuristics are used to search for an allocation which is good enough (the application meets all timing requirements as defined in the application model, described in deliverables D5.1 and D5.2).

The primary differences therefore between the static and dynamic acceleration design are:

- The migration infrastructure must be able to change the running design during runtime. This requires migrating all running locales back from the FPGA to the processor, and then reprogramming the FGPA with a new partial bitfile. In general, this requires checkpointing a running Locale and saving its entire state, something which is very hard to do with general purpose software and even harder with dedicated hardware. This issue is discussed in section 2.3.
- The HLS process runs online.
- The Locales selected for offload are chosen by feedback metrics rather than a fixed deployment.

It is important to remember that FPGA acceleration is concerned only with the software that is running on a single target node (cloud server). Locales are not moved from one node to another for acceleration and so the acceleration can consider only a single JUNIPER program, rather than the entire JUNIPER application.

2.3 Migration

Migrating an active object (like a thread) between hardware and software implementations is much more challenging than migrating a thread between cores of an SMP system. This requires the implementation of a *heterogeneous process migration* framework. The main purpose of such a framework

is to define a *state correspondence* between execution states of the hardware and software implementations of the same thread. With such a correspondence in place, *state translation* can then be used to move between one and the other.

The process of transferring a running thread from one processor to another requires the following steps:

- First the thread must be stopped so that its execution does not interfere with the migration. It is not possible to halt execution at any arbitrary point because the hardware and software implementations have very different execution paths. For example, a CPU program counter has no meaning in hardware.
- This problem is solved using a checkpointing system in which checkpoints are added at frequent points in the source code or bytecode. Threads may only be migrated once they reach a checkpoint. The checkpoints are identical in the source and target implementations, thereby allowing a program counter correspondence across different ISAs and implementation formats.
- The execution context of the thread must be extracted. This consists of the state of the processor's registers and the thread's stack. This can be obtained from the kernel on the source processor, normally named a process control block (PCB). Extracting the state of a hardware implementation is harder and requires dedicated 'read out' hardware.
- The context must be translated so that it fits the architecture of the target implementation. This requires converting register contents appropriately and marshalling between different data representations.
- The translated context is transferred to the target implementation and started.

There are a number of existing systems which implement heterogeneous process migration systems. Tui [11], Dome [2], Gantel et. al. [5], and Ramkumar et. al. [9] all use a version of the algorithm above to migrate tasks between processors with differing instruction sets and architectures. These approaches differ by the manner and frequency at which checkpoints are added, displaying different run-time properties accordingly. Systems based on Java or other interpreted languages have the advantage that threads are compiled to bytecodes so portability is guaranteed. Examples of such systems are JESSICA2 [15] and the work by Sakamoto et. al. [10].

Unfortunately, systems that can translate between hardware and software implementations are still in their infancy and not yet ready for industrial use, due to the vastly increased complexity of the problem. Accordingly, the JUNIPER approach will use a variation on software checkpointing described above in the literature. First, a simple user-driven version will be created for the initial dynamic acceleration prototype. This version will rely on the user to implement the migration of an `AcceleratableLocale` manually. We define an interface, `Migratable`, which is implemented by the `AcceleratableLocale` class. `Migratable` contains the following methods:

`public void` `initiateMigration();`
Called by the migration infrastructure when it wishes to migrate a locale from its current location. After this call, `isMigrating()` will start returning `true`.

`public boolean` `isMigrating();`
Returns `true` if `initiateMigration()` has been called by the dynamic acceleration infrastructure. User code should periodically check the return value of this method and respond accordingly.

`public byte[] requestMigrationBuffer(int size);`
The user code will use this method to request a buffer of `size` bytes that can be used to transfer state information during the migration.

`public void readyToMigrate(byte[] data);`
Called by the user code to signify that they have performed whatever housekeeping is required and they are now ready to be terminated by the dynamic acceleration infrastructure. `data` is a reference to a `byte` array, created with `requestMigrationBuffer()` which will be passed to `resumingFromMigration()` after the migration is completed.

`public void resumingFromMigration(byte[] data);`
Called by the migration infrastructure after this locale has been migrated. `data` is a reference to a `byte` array which was created with `requestMigrationBuffer()` and passed into `readyToMigrate()` by the user code.

`void releaseMigrationBuffer(byte[] data);`
Called by user code to release a buffer that was allocated by `requestMigrationBuffer()`.

This API allows for a robust and efficient checkpointing system to be implemented, with the downside that the user software must be involved and the process is not transparent. After this initial prototype has been developed, the research partners will investigate the possibility of performing this migration automatically. An automatic approach can use the fact that the input software is written in the RTSJ to simplify the problem. RTSJ programs do not contain free-running threads but instead implement their functionality using periodic and aperiodic event handlers. The automatic system will rely on this and only migrate at an idle tick, in between event handler releases, so therefore no running thread needs to be migrated. Only `static` variables in the Locale need to be copied during the migration.

2.4 Communication Redirection

A stated limitation in the static approach was that all Locales that communicate with an `AcceleratableLocale` must communicate through the JUNIPER communications API. Described in D2.1, this API is a wrapper around MPI [6]. The reason for this can now be seen when considering the dynamic case.

In the dynamic system, it is not known ahead of time whether an `AcceleratableLocale` is currently residing on the FPGA or in software. Consequentially, Locales communicating with it would not know where to send messages. In the JUNIPER system, the communications API solves this problem by abstracting away from direct inter-locale communications. As shown in figure 5, communications can be transparently routed according to the current state of the target locale.

2.5 Offload Metrics

The JUNIPER project provides many potential repositories of profiling data that can be used by the dynamic acceleration design. The specific policies and metrics that will be used to decide which Locales should be offloaded are not known ahead of time, and so are to be researched through the development of the JUNIPER project, specifically in conjunction with the Scheduling Advisor work

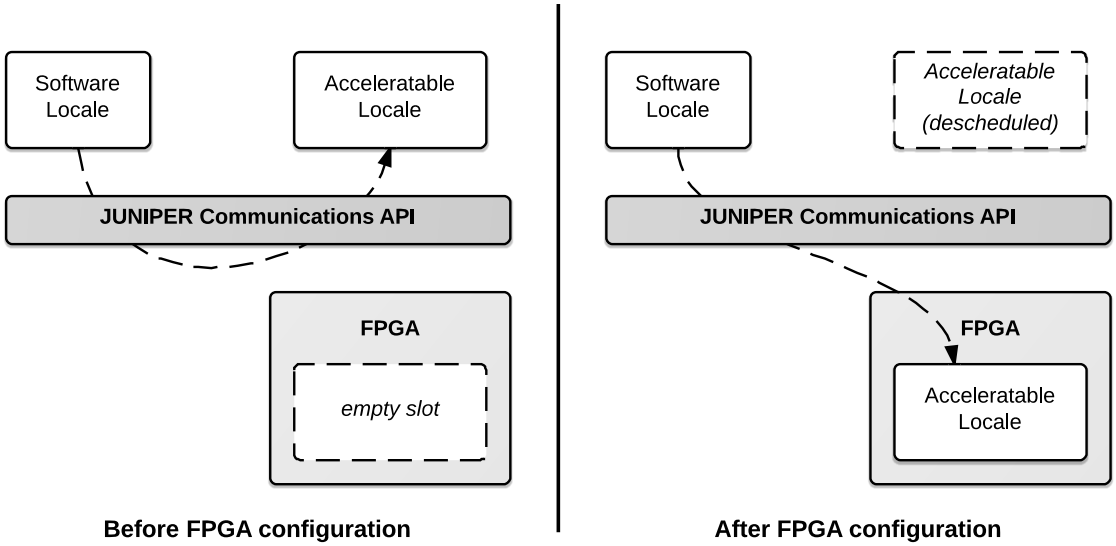

Figure 5: Communication redirection occurs inside the JUNIPER Communications API

in workpackage 3. The results of this will be described in the upcoming deliverable D2.7, *Dynamic Acceleration Implementation and Assessment*.

The metrics can be obtained from the following locations:

- The application itself: It is possible for the migration infrastructure to profile the running application directly to determine which Locales are being called the most frequently. This provides a coarse metric for which parts of the system are likely to be worth offloading, but does not provide more advanced information such as data rates or memory and heap usage.
- The JUNIPER monitoring framework: The JUNIPER project defines the use of monitoring frameworks to observe the performance of the entire application as it executes on the target cluster. This information is available from a service node in the cluster and can be queried like any other web service. The advantage of such an approach is that rather than the offloading decisions being completely opaque, if required the developer can also gain access to this information through the same mechanism.
- The JUNIPER Scheduling Advisor: As described in deliverable D3.8, the JUNIPER project defines a Scheduling Advisor which monitors the use of each node and determines the optimal deployment of the JUNIPER application throughout the target cluster. This information can perhaps also be leveraged to make offloading decisions.

The initial prototype of the dynamic acceleration framework will use simple profiling to count the CPU time spent within each `AcceleratableLocale`. The Locales with the highest amount of time spent will be selected for offloading until the FPGA is full. This system is likely to result in local minima (for example, if a single large Locale with a high CPU usage takes the space of two smaller Locales with slightly lower individual usages but a larger combined usage). Later versions of the framework will then extend these metrics to include information collected at runtime from the JUNIPER monitoring framework. The following metrics will be considered:

- Per thread CPU usage

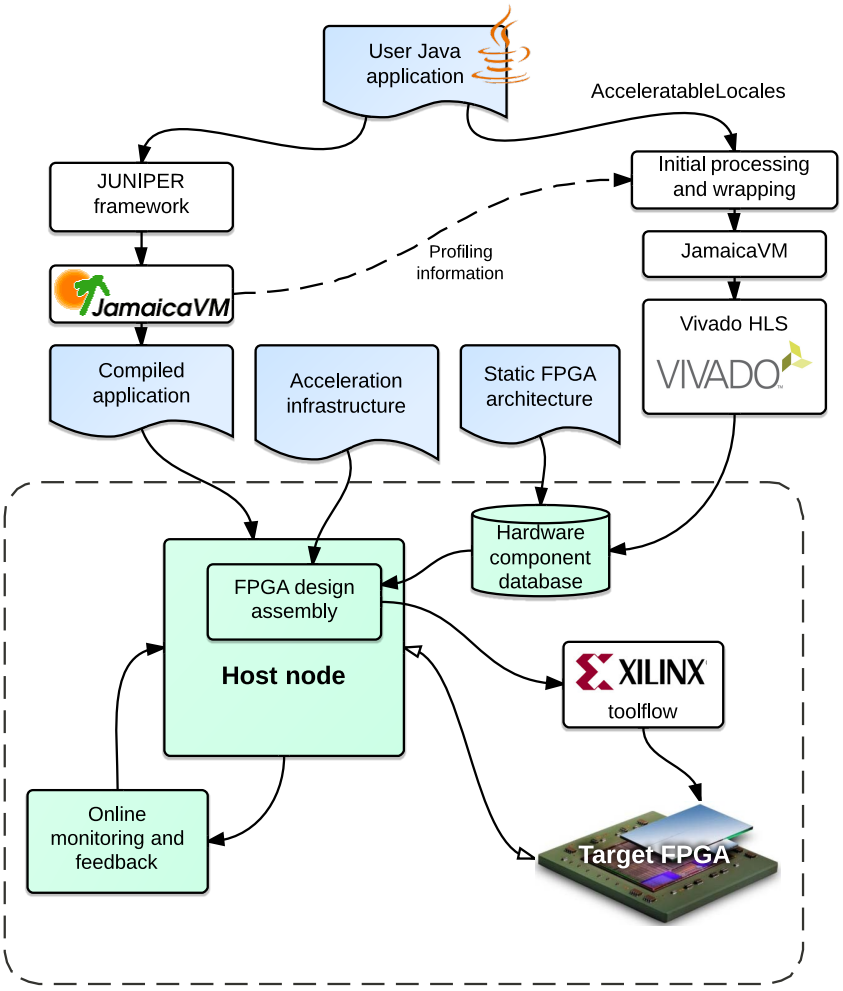

Figure 6: The JUNIPER Dynamic Acceleration Toolflow

- Per thread Memory usage
- Per thread Disk bandwidth
- Per thread Network data rate
- Node CPU usage
- Node Disk bandwidth

The per (Java) thread metrics are used to determine which parts of the application to offload, the per node metrics are used to determine when to do so.

2.6 Toolflow

The designed toolflow is shown in figure 6. The input application (in RTSJ-compliant Java) runs through the normal JUNIPER development processes as detailed in the other deliverables of the project. The result of this is a (set of) binaries for execution on the target node.

The figure also shows the static acceleration design. The `AcceleratableLocales` of the software are passed through a processing transformation and compiled to C using JamaicaVM. This C is then compiled to hardware description language (VHDL) using Xilinx Vivado HLS. The resulting components are stored in a database that is accessible by the running system.

The lower half of the figure shows that as the system is executing, online monitoring and feedback is used to determine what parts of the system may be suitable for dynamic acceleration. This information feeds an assembly stage which runs on the target node, and constructs an FPGA design from the static section of the design and selected transformed Locale components. The resulting design is built using the FPGA vendor toolflow and programmed to the FPGA.

2.7 Limitations

There are a number of limitations in the designed system that should be considered.

First, as the HLS process must now run online, this requires the Xilinx development tools to execute on a cloud-connected server. For commercial use this poses potential licensing issues. This will not affect the development of the system through the lifetime of the project because the responsible research partners have the required licenses. Moving forward, however, it is a factor that should be considered. One possible way to avoid this is to architect the system so that rather than rebuilding the bitfiles online, the system instead selects from a set of pre-generated bitfiles that are constructed in bulk offline and then stored on the node. The system can then change bitfiles as it enters different modes. Mode changes are a well-studied aspect of real-time systems and are supported by existing analysis tools and the RTSJ. This gives rise to 'partially-dynamic' approach, but does not require online FPGA tool use.

Due to the manner in which the dynamic approach is built on the static, the limitations of the static approach still apply. Primarily that only `AcceleratableLocales` may be offloaded, and that other Locales must use the JUNIPER communications libraries to communicate with the offloadable Locales.

As a necessary side-effect of the dynamic nature of this approach, a system that uses the JUNIPER dynamic acceleration approach will not be as predictable as one that uses the static approach. In some real-time systems this unpredictability may be unacceptable. This approach is better used for soft real-time applications.

3 Conclusions

This document has detailed the JUNIPER dynamic acceleration design. This system builds on previous deliverables which describe a static acceleration system. In the static system, sections of the input Java code could be marked for implementation on FPGA. These sections were compiled to a hardware description language and implemented on an attached FPGA accelerator.

The dynamic approach removes the limitation that the application code be marked ahead of time. It instead uses a range of performance metrics that are made available by the JUNIPER framework to determine at runtime a suitable subset of the application to offload. This is a search problem, and so the dynamic system runs periodically, constantly re-evaluating the state of the system to determine if a better offloading can be obtained.

References

[1] aicas GmbH. The JamaicaVM 6.0 User Manual. http://www.aicas.com/jamaica.html, July 2010.

[2] Jose N Arabe, Adam Beguelin, Bruce Lowekamp, Erik Seligman, Mike Starkey, and Peter Stephan. Dome: Parallel Programming in a Heterogeneous Multi-User Environment. Technical report, Pittsburgh, PA, USA, 1995.

[3] 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.

[4] Simone Corbetta, Massimo Morandi, Marco Novati, Marco Domenico Santambrogio, Donatella Sciuto, and Paola Spoletini. Internal and External Bitstream Relocation for Partial Dynamic Reconfiguration. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17-11:1650–1654, 2009.

[5] Laurent Gantel, Salah Layouni, Mohamed El Amine Benkhelifa, F. Verdier, and StÃ©phanie Chauvet. Multiprocessor Task Migration Implementation in a Reconfigurable Platform. In *International conference on ReConFigurables Computing and FPGAs (ReConFig)*. IEEE Computer Society, 2009.

[6] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1994.

[7] Michael Huebner, Tobias Becker, and Juergen Becker. Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration. In *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*, pages 28–32, New York, NY, USA, 2004. ACM Press.

[8] T. Marescaux, J. Mignolet, A. Bartic, W. Moffat, D. Verkest, S. Vernalde, and R. Lauwereins. Networks on Chip as Hardware Components of an OS for Reconfigurable Systems. In *Proceedings of the 13th International Conference on Field Programmable Logic and Applications, Lisbon*, 2003.

[9] Balkrishna Ramkumar and Volker Strumpen. Portable Checkpointing for Heterogeneous Archtitectures. In *FTCS '97: Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, page 58, Washington, DC, USA, 1997. IEEE Computer Society.

[10] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. In David Kotz and Friedemann Mattern, editors, *Agent Systems, Mobile Agents, and Applications*, volume 1882 of *Lecture Notes in Computer Science*, pages 443–481. Springer Berlin / Heidelberg, 2000.

[11] Peter Smith and Norman C. Hutchinson. Heterogeneous Process Migration: The Tui System. Technical report, Vancouver, BC, Canada, Canada, 1996.

[12] M. Ullmann, M. Huebner, B. Grimm, and J. Becker. An FPGA run-time system for dynamical on-demand reconfiguration. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 135–, April 2004.

[13] Xilinx Corporation. Microblaze Processor Reference Guide. UG081 v13.2, 2011.

[14] Xilinx Corporation. Zynq-7000 All Programmable SoC. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/, April 2014.

[15] Wenzhang Zhu, Cho-Li Wang, and F.C.M. Lau. JESSICA2: a distributed Java Virtual Machine with transparent thread migration support. pages 381 – 388, 2002.