Architecture-Awareness for Real-Time Big Data Systems

Ian Gray
Dept. of Computer Science
University of York, York, U.K.
ian.gray@york.ac.uk

Neil C. Audsley
Dept. of Computer Science
University of York, York, U.K.
neil.audsley@york.ac.uk

Yu Chan
Dept. of Computer Science
University of York, York, U.K.
yc522@york.ac.uk

Andy Wellings
Dept. of Computer Science
University of York, York, U.K.
andy.wellings@york.ac.uk

ABSTRACT

Existing programming models for distributed and cloud-based systems tend to abstract away from the architectures of individual target nodes, concentrating instead on higher-level issues of algorithm representation (MapReduce etc.). However, as programmers begin to tackle the issue of Big Data, increasing data volumes are forcing developers to reconsider this approach and to optimise their software heavily. JUNIPER is an EU-funded project which assists Big Data developers to create architecture-aware software in a way that is suitable for the target domain, and provides higher performance, portability, and real-time guarantees.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

General Terms

Real-Time, Big Data, JUNIPER, Real-Time, High Performance Architectures, Cloud-Computing, FPGAs

1. INTRODUCTION

The volume of data being processed and stored by internet applications is increasing exponentially. This issue has been termed 'Big Data', referring to the observation that the main challenge for such systems is the volume of data that they must handle [9], rather than the processing that is performed on individual data elements. Due to the scale of these systems they display a large degree of heterogeneity in the servers upon which they are deployed. Systems are built from a range of commodity hardware up to high-performance architectures and supercomputers. Finally the deployment platforms for such systems are generally shared between multiple users through hypervisors and virtual machines, causing further heterogeneity and design-time architectural uncertainty.

JUNIPER [2] is an EU Seventh Framework project which believes that the unique challenges created by Big Data are currently not well-addressed. Existing programming models are based on standard desktop programming languages and so attempt to abstract hardware details (of both the target node and the inter-node communications) in a way which makes it difficult for the programmer to exploit the full power of the underlying hardware. The hardware information that is available is presented at the wrong granularity. As a solution, JUNIPER defines a new programming model based around Java 8 [12] with the following core principles:

- It is not possible to express an entire Big Data system at the source-code level, so elements of model-driven engineering (MDE) are employed to ease development, portability, and deployment.
- Big Data programmers want the ability to optimise their software to reduce latency and increase throughput. It is necessary to provide access to architectural features (CPUs, memory layout, caches, communications, and accelerators) in a portable way which is suited towards the target domain.
- Part of this optimisation should include real-time requirements and guarantees. The JUNIPER framework allows the developer to reserve system resources (CPU time, bandwidth) for high priority threads of their software to maximise quality-of-service when the system is under high demand.

This paper will introduce the main aspects of the JUNIPER programming model in section 2 and discuss how this is augmented by MDE in section 2.3. Section 3 describes the ways in which the JUNIPER API aims to accelerate Big Data applications through exposing appropriate architectural features, and shows some preliminary results in this area. Finally, section 5 concludes.

1.1 Related work

Many Java-based, Big Data applications make use of the Apache Hadoop framework [1], which enables processing

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

of large data sets across clusters of off-the-shelf computers using the MapReduce programming model [6]. However, Hadoop does not make it easy to exploit the full power available on a single node. For example, each Hadoop map or reduce task executes in a separate Java Virtual Machine (JVM) by default, thus incurring a significant overhead for every task start-up. The cloud infrastructure may invoke Hadoop tasks at an unpredictable location in the cloud, and so it is important for these tasks to react to the architecture of their current host node. Support for this is not provided by Hadoop. Finally, Hadoop requires manual configuration files to describe its target cluster, which makes reconfiguration and retargeting difficult.

Apache Spark [3] is a framework developed in response to Hadoop's reliance on the batch processing, two-stage, MapReduce paradigm. Spark is based on the same file system as Hadoop (HDFS) but expands its computational model to include more general computation stages, including streaming data processing. Spark provides a computational framework for other tools to be built on top of.

Storm [4] also attempts to extend past Hadoop's limited model by allowing the definition of a directed acyclic graph of event-triggered transformations which describes the processing required. This can then be deployed onto the target cluster. Storm focuses on 'realtime computation', but in this context 'realtime' means that a streaming (rather than batch) processing model is used. Real-time guarantees are not considered.

The JUNIPER project is based on version 8 of the Java programming language. Java has enjoyed much acceptance in the industry through the prevalence of Hadoop and bindings to Spark and Storm. JUNIPER also makes use of the Real-Time Specification for Java [8] to enable the development of systems which can provide timing and response guarantees. The volume of data in a Big Data system places real-time constraints on the implementation, because it is not possible to store all data in memory. Incoming data must be processed and stored at a rate which can keep up with the incoming data rate.

Java 8 [12] is the latest release of the Java programming language, available since 2014. It adds a range of features designed to make Big Data programming easier in the Java ecosystem. Lambda expressions are the most visible addition to Java 8. They provide a concise way to express functional programming concepts in Java [13]. A lambda can be specified in place of a value whose type is a functional interface (an interface with exactly one abstract method). This can aid parallel programming because their limited interface encourages a functional style of programming, in which data dependencies are minimised and work is decomposed into functional units. This maximises the available parallelism in a given work load and aids the creation of data parallel streaming pipelines. They have a much more compact syntax than Java's original approach which used anonymous inner classes, motivating their use.

Java 8 also introduces the concept of a *stream*, which is a sequence of elements that can be operated on. Streams can be generated from collections and a pipeline of bulk data operations [7] can be performed on it. Streams can be sequential or parallel. Operations on parallel streams are automatically evaluated by the framework in parallel. Pipelines are also evaluated lazily; only enough elements are consumed as required by the terminal operation. Streams can be combined with common parallel programming operations such as `map` to provide a similar model to that of MapReduce, or chained together to provide a dataflow model reminiscent of Spark.

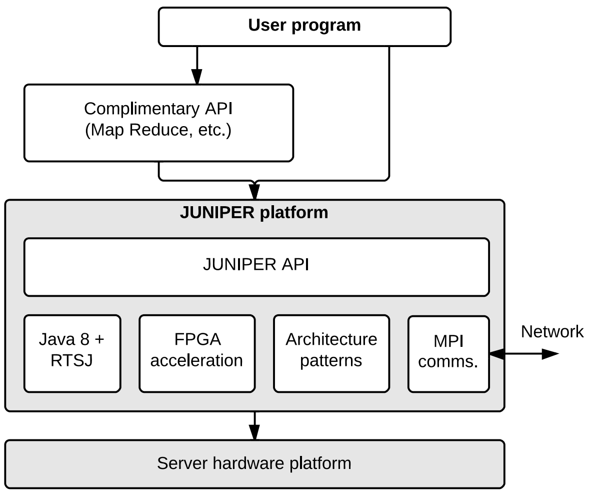

Figure 1: A single server in a JUNIPER cluster.

2. PROGRAMMING MODEL

A complete description of the programming model is outside of the scope of this paper, so this will instead provide an overview of the main features and concepts of the JUNPER programming framework. More details can be found in [5].

2.1 Scope

The JUNIPER programming framework aims to support the development of data-centric applications in a way that is general enough to cope with every need, but also allows common data processing patterns to be abstracted, modelled, and optimally deployed. It is not meant to fully replace existing parallel processing frameworks, such as MapReduce [6]. Instead, the layered design of the JUNIPER platform (see figure 1) allows frameworks to be built on top of the JUNIPER framework through standard Java approaches (e.g., as a Java library) and thus benefit from the hardware abstraction features for real-time and large-scale parallel data processing. This is similar to the approach taken in Apache Spark (section 1.1).

Accordingly, the programming model does not aim to provide any specific paradigm but to provide a new framework for building libraries to provide these paradigms in a portable yet architecture-aware and real-time way.

The model has two levels, *Application* and *Program*. The Application level considers the large-scale movement of data; i.e. How does data enter the application, how does it move from program to program, and where is it stored. It also describes the requirements placed on the application (i.e. response times or required throughput). At this level, communication is implemented using MPI [11]. The programs of the application use MPI for all coordination and data transfer.

At the Program level, a node of the cluster is programmed using a single Java program running inside a single JVM (al-

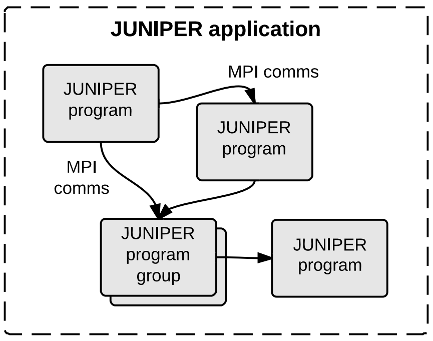

Figure 2: A JUNIPER application is composed of JUNIPER programs, which may be unique, or one of a group of identical program instances.

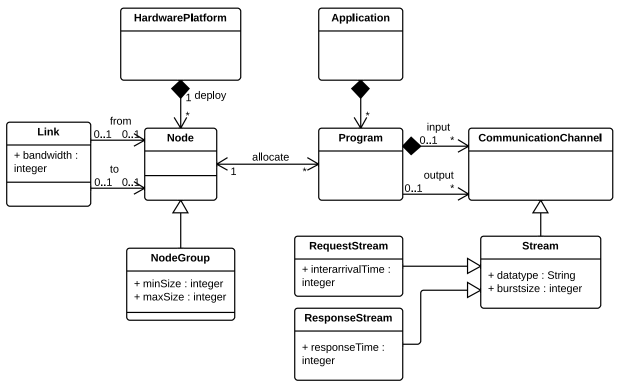

Figure 3: Simplified view of the modelling concepts of the JUNIPER modelling language.

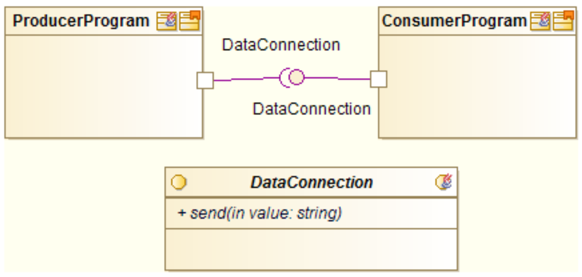

Figure 4: Fragment of the model used to generate the code in figure 5.

though the cloud infrastructure may map multiple programs to the same physical server). The program level focuses on efficient exploitation of the machine through the use of architecture patterns and locales (see section 3), and it makes use of reservations to ensure real-time behaviour (section 4).

2.2 Applications and Programs

The processing model specifies the design of a *JUNIPER application*. A JUNIPER application exists at the 'cluster' or 'cloud' level and is comprised of a set of Java *programs* that use the JUNIPER API (henceforth called *JUNIPER programs*) to communicate and coordinate to solve a problem. JUNIPER programs are mapped to the nodes (servers) of the target cluster, potentially multiple programs to a server.

The graph of communications in the model is fixed. Each program has a fixed set of input data flows and output data flows. These are modelled at the MDE level (see section 2.3). The only dynamism in the model is for situations where multiple identical instances of the same program are required (such as the mappers of a MapReduce application). A *Program Group* may be defined, which replicates a given program a runtime-variable number of times (subject to optional maximum and minimum bounds).

For example, consider a financial application in which incoming credit card transactions have to be scanned for suspicious activity before being approved or declined. The entire solution is a JUNIPER application, which consists of JUNIPER programs to read incoming requests, programs to perform database access, programs to correlate past behaviour etc. The relationship between applications and programs is shown in figure 2.

JUNIPER programs communicate using a communications API built on the industry-standard MPI [11]. Raw MPI is complex to use and must be updated as nodes are added to and removed from the system. This complexity is hidden through the use of MDE, as discussed in section 2.3.

2.3 Model-driven Engineering

As noted in section 1, existing programming languages are designed to specify an individual program, not a large distributed application. This can be seen in systems like Hadoop or Spark where deployment information is encoded outside of the language (in configuration files or launch configurations). In the JUNIPER system, the overall structure of the application and the deployment of that application onto hardware is described using MDE. The JUNIPER modelling language is based on a profile of SysML [14]. A simplified view of its modelling concepts is shown in figure 3.

MDE is used for four main reasons. First, application structure and deployment can be succinctly and rapidly described in a modelling language without interference from the node-level programming language. This description is language-independent. Second, MDE encourages portability and code reuse through the use of automatic code generation. The modelling tools automatically implement sections of the JUNIPER API that would otherwise be in a 'porting layer' for manual implementation, so that the node-level code does not need to worry about cluster-level details. This separation of concerns allows rapid redeployment of software over changing architectures, as exemplified in section 2.4. Third, MDE allows the designer to specify real-time constraints on aspects of the system, and so to analyse these through development. This is discussed in section 4. Finally, not discussed in this paper is that the system model provides a useful repository for online feedback and performance monitoring. The JUNIPER approach includes monitoring software to analyse how the system is performing. This information is passed back to the user via the system model.

2.4 Communications

An example of the code generation used in the JUNIPER approach is shown in figure 5. As can be seen in figure 4, the developer has modelled two communicating programs.

```
@objid("1342fb7a-1474-4e5c-8364-afb3dc330f78")
public class ConsumerProgram {
  @objid("4d8d9176-631b-484c-8582-fff35463d57a")
  public static final int RANK = 3;

  @objid ("4db8f2b4-6aed-4110-8e4a-18e678a69178")
  public static DataConnection dataConnectionImpl
    = new DataConnection() {};

  @objid("35dee53b-71b0-45c4-9e7f-6ffa3c4d4359")
  public static void initProvidedInterfaces() {
    Util.initProvidedInterface(
      ProducerProgram.class, dataConnectionImpl);
  }

  @objid("dfa4077e-5340-408c-b5e7-7bfe5f151371")
  public static void main(final String[] args) {
    MPI.Init(args);
    initProvidedInterfaces();
    while (true) {
      Thread.yield();
      Util.processReceivedMessages();
      if (execute()) break;
    }
    MPI.Finalize();
  }

  //...Further detail omitted
```

Figure 5: Fragment of code generated from the model in figure 4.

This is reflected in the generated code; the developer does not have to program the communications layer manually.

3. EXPOSING ARCHITECTURAL DETAILS

A goal of the JUNIPER programming model is that it provides an infrastructure in which the programmer can manage the *locality* of the code and data of their system. Locality is a measure of proximity for code and data throughout the implementation architecture. The programmer can dynamically discover the host architecture of their software and respond accordingly by mapping the locality of their software to ensure that code is placed onto a CPU that is close to the memory in which its data is stored. The architecture is represented as composed of architectural patterns, as shown in figure 6.

The programmer calls the JUNIPER API to determine the layout of their host hardware, which may be highly dynamic in a cloud environment. They can then use *locales* (section 3) to tailor their application to the target by binding threads and data accordingly. A locale has a range of properties. They are the unit of allocation for mapping threads and objects to the CPUs and memories of the architecture. A locale is mapped to a subsection of the architecture (see 'architectural patterns' in the following section) and the contents of that locale will remain in that subset. A high-priority locale can request a guaranteed resource reservation (CPU time, bandwidth etc.) that is the result of a negotiation between the JVM and the host operating system (see section 4). The heap, immortal and backing store memory allocated to a locale are not allocated to any other locale. Finally, locales can be offloaded to attached FPGA accelerators (see section 3.4).

The approach taken is to provide factory methods to create threads (including real-time threads and asynchronous event handlers) and memory areas in the RTSJ. Creation of these objects outside of these factory methods have no locality defined and can be located at the JVM's discretion.

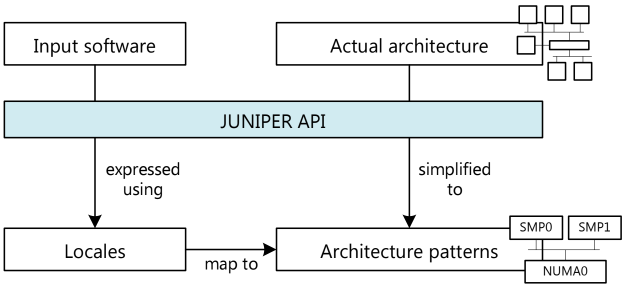

Figure 6: Architecture patterns used to assist software development

3.1 Architectural Discovery

Locales provide locality information; essentially that given software elements should be located together and with a given resource reservation. In the JUNIPER approach, locales may be mapped to a subset of the target hardware. The JUNIPER hardware model (section 2.3) is based on the idea of *architecture patterns*. Patterns are used because programmers of Big Data systems are more concerned about the *class* of architecture than its precise details. For example, whether or not it has coherent caches, or whether or not all memory is of equal speed. Architecture patterns capture this well.

Given the large platforms used for big data systems, the programmer does not require the low-level control afforded by techniques such as affinities, in which each thread is bound to a specific set of individual processors. This is onerous for large systems and lacks portability. Instead of having to individually map threads on a 100-core machine, the programmer wishes to be able to express that a given large group of threads should be located on a given large group of SMP-coupled processors, at which point the runtime and infrastructure can be trusted to schedule and place these appropriately.

Given the above points, the patterns exposed by the JUNIPER API are:

- NUMA: Provides few guarantees. It will contain a single address space, but caches may be incoherent and memory access times are unknown.
- ccNUMA: Constrains the NUMA architecture with the guarantee that caches will be kept coherent from the point of view of the Java programmer. Memory access speeds are still unknown and variable.
- SMP: Represents a tightly-coupled architecture in which access times to memory are uniform within a reasonable error bound. Variation is only due to bus contention or cache effects, not because memory is a greater 'distance' from the processors.

3.2 Stored Collections

One major drawback of the built-in Java collections is that all of their data must be in heap memory. The programmer must at least partially populate the collection before use, and this can be problematic in a Big Data system. Heap memory is small compared to disk space, so for Big Data computations there may not be enough heap memory to load the entire dataset from disk. The programmer must instead manage their use of memory carefully leading to unnecessary complication. Furthermore, Java 8's streams are lazily evaluated, but no advantage can be gained from this if the entire collection must be first loaded in memory.

To overcome these limitations, the JUNIPER API introduces the *Stored Collection* which reads its data lazily from a file on demand, thus eliminating the initial population step. To support different data formats, several types of Stored Collections are provided to support both Java's primitive types (`int`, `char` etc.) and serialized class instances.

Aside from avoiding collection population, the main advantage of Stored Collections is that they can internally make use of JUNIPER's architecture discovery to tune their data access to best utilize the storage medium. On a normal hard disk, the Stored Collection will serialise and coalesce data accesses because this is the fastest way to operate a hard disk. If the data is stored on a high performance parallel file system (such as Lustre [15]) then reads will use an appropriate level of parallelism to maximise throughput.

Stored Collections have been shown [5] to increase execution times of disk-bound programs by 44% and reduce heap usage by very large amounts (up to 84% when compared with in-memory collection programs). MapReduce workloads are shown to be at least 24% faster than equivalent Hadoop programs when only looking at the performance of individual nodes, and in some pathological cases over 8 times faster.

3.3 Locality in Java Streams

The JUNIPER project has also used the concept of Stored Collections to provide Java streams with locality. In standard Java 8 programs, a pipeline of stream operations is executed without regard to the physical layout of the target system. In a large SMP system, different cores can have very different access times to different memory banks [10]. Optimal performance is obtained by splitting data throughout the memory of the system and then scheduling threads that need that memory onto the nearest processing cores. Java's stream pipelines provide enough semantic information to be able to do this, but existing implementations do not implement it.

In JUNIPER, the implementation of Stored Collections attempts to maximise locality by leveraging its lazy evaluation. Rather than rely on a single large disk buffer (as in a naive implementation) smaller buffers are allocated on-demand by the threads that are executing the stream pipeline. In large architectures combined with a NUMA-aware allocator, this maximises the chance that data can be allocated near to where it is required. Preliminary results have shown success with this approach, as shown in figure 7. The figure shows the execution times of 200 executions of a simple benchmarking program that uses Stored Collections, running on a 16-core (4×4) AMD Opteron NUMA machine, expressed as a cumulative frequency graph. The affinity-aware implementation of Stored Collections can be disabled, bound to individual cores, or to individual NUMA nodes. As can be seen, affinity-awareness decreases the execution time over the naive implementation due to greater use of caching and

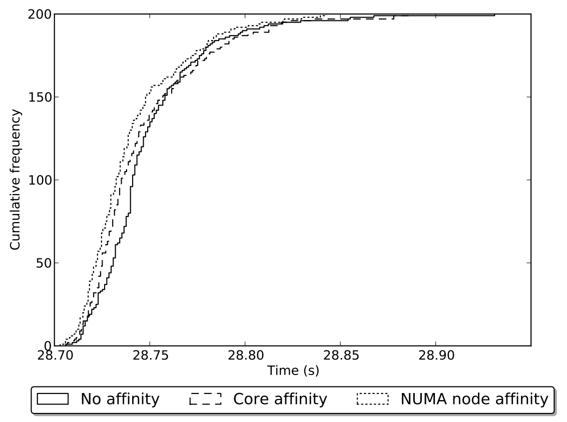

Figure 7: Stored Collections with, and without, locality-aware allocation of buffers

lower memory contention. However, it must also be noted that in other situations and for other benchmarks, affinity was not observed to significantly increase performance, and so work continues to characterise this effect.

3.4 FPGA acceleration

The final area in which the JUNIPER project is encouraging architecture-awareness is in the use of FPGA acceleration. The JUNIPER framework has support for FPGA boards to be connected to the nodes in the target cluster. The framework attempts to make use of these FPGAs easy, without knowledge of FPGA design.

To do this, the project leverages the fact that Locales already provide the programmer with a way of expressing their software in terms of sections of relatively self-contained, tightly-coupled, functionality. Locales are therefore ideal as the unit of offload to the FPGA. The JUNIPER project uses a Java to C compiler, and then a C to hardware description language tool to automatically create FPGA components that implement parts of the input software. This flow is shown in figure 8.

It is not possible to translate general Java to hardware efficiently. However, JUNIPER already includes the use of the RTSJ which removes garbage collection with scoped memory management, and JUNIPER implements its own communications based on MPI. Therefore the JUNIPER API defines an `AcceleratableLocale` class, which is a restricted form of Locale which can be supported by the FPGA design tools. A full description of this process is outside of the scope of this paper, but the main restriction is that `AcceleratableLocale` includes an abstract method called `initialise` which creates all of the threads that will be allocated inside that locale. The `initialise` method is analysed and used to begin constructing the hardware component.

The project defines support for both static and dynamic acceleration. In the static scheme, the specific subset of Acceleratable Locales which will be offloaded is defined ahead of time. In the dynamic scheme, the JUNIPER framework uses online monitoring and feedback to swap Locales in and out of the FPGA at system run time.

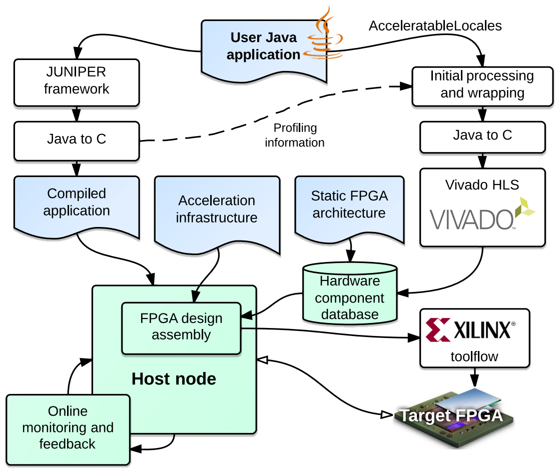

Figure 8: The FPGA acceleration tool flow

4. REAL-TIME CONSIDERATIONS

A full discussion of timing analysis in JUNIPER is outside of the scope of this paper. This section provides a brief overview.

In most Big Data contexts, the term 'real-time' means that any required processing occurs as data arrives, rather than as a batch job at a later time. In JUNIPER, 'real-time' also means that parts of the application may be given true real-time constraints and supporting scheduling analysis will be used to determine that these timings constraints are met. Clearly, due to the uncertainty in internet-based communications the analysis performed is not hard real-time (in which no deadline may be ever missed) but rather that a suitably high quality-of-service (QoS) can be maintained by the system under expected conditions.

The system model (section 2.3) is used to provide timing requirements. Requirements are placed on the communication channels of the system. If program A uses the JUNIPER API to send data to program B, then there exists a communication channel $A \rightarrow B$. Channels are marked as periodic or sporadic and therefore have either a period or inter-arrival time. Also they are typed with a Java type and describe the size and burst size of the data. This information is then used to derive timing requirements for individual nodes in the system, which can then be analysed separately.

The system designer uses locality (section 3) to reduce pessimism in the analysis of their code. Further, Locales can be given reservations to guarantee that the real-time part of the system receives the resources it requires to attain a suitable QoS. The JUNIPER infrastructure (including schedulers and OS support) is then responsible for managing system resources (processor time, communications and disk bandwidth) to ensure that the reservations are met.

5. CONCLUSIONS

This paper has discussed how the programming model developed in the JUNIPER project is targeting future, cloud-based, distributed architectures. The extreme demands imposed on systems due to Big Data mean that efficiency and real-time considerations are paramount. Developers are required to optimise their software for the architectures of their target systems. However, due to the uncertainty introduced by virtualisation and cloud-computing middleware layers, existing techniques for optimising against architectural details are inappropriate.

The JUNIPER approach uses model-driven engineering to introduce a higher-level view of the system. This allows the introduction of real-time constraints, and the use of automatic code generation for rapid deployment. At the node level, an API is introduced that provides a range of abstractions that aid the developer to write code which can react appropriately to the underlying architecture. Java 8 is used, and extended with a new set of Collections to better fulfil the requirements of Big Data. The approach is already showing some promising results.

6. REFERENCES

[1] Apache Hadoop Website. http://hadoop.apache.org/, 2011.
[2] The JUNIPER project. http://www.juniper-project.org, July 2014.
[3] Apache Software Foundation. Apache Spark – Lightning-Fast Cluster Computing. http://spark.apache.org/.
[4] Apache Software Foundation. Apache Storm – Distributed and fault-tolerant realtime computation. http://storm.incubator.apache.org/.
[5] Y. Chan, I. Gray, A. Wellings, and N. Audsley. Exploiting multicore architectures in big data applications: The JUNIPER approach. In *Proceedings of MULTIPROG 2014 : Programmability Issues for Heterogeneous Multicores*, 2014.
[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun.*, 51,1:107–113, 2008.
[7] M. Duigou. JEP 107: Bulk Data Operations for Collections. http://openjdk.java.net/jeps/107, Sepetmber 2011.
[8] J. Gosling and G. Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.
[9] A. Jacobs. The pathologies of big data. *Communications of the ACM*, 52(8):36–44, 2009.
[10] S. Kim. *Node-oriented dynamic memory management for real-time systems on ccNUMA architecture systems*. PhD thesis, Department of Computer Science, University of York, UK, 2014.
[11] Message Passing Interface Forum. MPI: A message-passing interface standard, version 2.2. Specification, September 2009.
[12] Oracle Corporation. JDK 8 Schedule and status. http://openjdk.java.net/projects/jdk8/, September 2013.
[13] Oracle Corporation. Project Lambda. http://openjdk.java.net/projects/lambda/, December 2013.
[14] T. Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
[15] Xyratex Ltd. The Lustre file system. http://www.lustre.org, December 2013.