Real-Time Big Data: the JUNIPER Approach

N. C. Audsley, Y. Chan, I. Gray & A. J. Wellings

Real-Time Research Group, Department of Computer Science, University of York, UK

Abstract

Cloud computing offers the possibility for Cyber-Physical Systems (CPS) to offload computation and utilise large stored data sets in order to increase the overall system utility. However, for cloud platforms and applications to be effective for CPS, they need to exhibit real-time behaviour so that some level of performance can be guaranteed to the CPS. This paper considers the infrastructure developed by the EU JUNIPER project for enabling real-time big data systems to be built so that appropriate guarantees can be given to the CPS components. The technologies developed include a real-time Java programming approach, hardware acceleration to provide performance, and operating system resource management (time and disk) based upon resource reservation in order to enhance timeliness.

I. Introduction

A key challenge for *cyber-physical systems* (CPS) is the successful exploitation of the computing and storage facilities in the cloud, whilst still meeting resource constraints, such as eg. time, power, space or cost. Using the cloud, CPS can offload computation, store and query large quantities of data, and share resources and data with other related CPS. However, whilst CPS are generally designed and implemented to meet stringent timing and resource constraints, cloud based systems are in general built without such timing and resource constraints. Also, communication between CPS and the supporting cloud based applications may involve networks that are shared, and potentially public (ie. internet), making communication latencies difficult to bound.

Whilst cloud facilities can be exploited as currently implemented (with no meaningful guarantees regarding performance), far greater overall system utility can be achieved if the cloud system is able to provide levels of guaranteed performance to the CPS. This could enable the CPS to utilise the cloud as a fundamental part of the system, rather than merely an unreliable occasional additional service.

This paper discusses the key challenges posed by the integration of CPS and cloud systems, in the context of *Real-Time Big Data* (RTBD) systems [9]. In [11], [17] the the main technologies developed within the EU Framework 7 JUNIPER project [4] are discussed, showing how real-time big data systems can be built in a general purpose business computing context. In this paper, we examine how the JUNIPER approach can be utilised to enable real-time behaviour for CPS utilising cloud systems. We show how JUNIPER provides RTBD cloud infrastructure built from real-time technologies, using real-time principles, so that appropriate guarantees can be given to the services implemented on the cloud, and hence to CPS using the cloud services. The overall approach includes:

- a real-time scalable Java based platform (with supporting development methodology);
- acceleration of key components using FPGA hardware;
- support within commodity cloud OSs (ie. Linux) for real-time use of (parallel) mass-storage;
- cloud scheduling amenable to predictability.

The JUNIPER cloud infrastructure can be configured to build and support a range of high-performance cloud computing approaches and paradigms, from map-reduce to stream [13], [2], [6]. It enables real-time constraints to be met – noting that shared communications between cloud and the CPS, and the potentially shared nature of cloud platforms themselves, dictate soft real-time guarantees.

The remainder of this paper is arranged as follows. In section II further background on RTBD is given. Section III provides an overview of the JUNIPER project, with sections IV, V and VI giving details of the programming model, program acceleration and OS respectively. Conclusions are offered in section VII.

II. Big Data and Real-Time Big Data

Big Data is a widely used term to describe the rapid growth of the availability of data [18]. Essentially the growth of data can be thought of in three main ways [19]: volume, velocity and variety. However in many systems data is growing more rapidly than the ability of applications and systems to process, analyse and store that data [21]. This is more readily apparent when big data applications that process live streaming data are considered, where data is to be analysed sufficiently fast so that incoming data is not lost. Also techniques for analysing big data are not necessarily based upon traditional database techniques as these do not necessarily scale to the table sizes involved, or can necessarily cope with unstructured data. Alternatively data-analytic approaches such as map-reduce, which are amenable to massive parallelisation [6] are utilised.

A. Real-Time Big Data

Applications utilising “Big Data” resources increasingly include requirements for “real-time” behaviour – ie. *Real-Time Big Data* (RTBD) applications [9]. Such real-time requirements are, in general, business-critical – eg. querying past historical system data to aid current decision making.

Also increasing is the use of RTBD applications in conjunction with CPS, eg. automotive, where cars (ie. users) interact with remote cloud computing resources for data, and to optimise performance (eg. traffic data, routes), or to provide additional advisory services [3]. Some level of guaranteed performance is required from the RTBD application in the cloud for the CPS to be able to utilise the cloud effectively. Whilst hard (ie. safety-critical) guarantees are not appropriate for RTBD applications in the cloud, best-effort and soft real-time approaches are appropriate. These enable a response to be provided within a specific timeframe. Whilst the CPS may not receive optimal information, given the amount of data that may have to be searched, an appropriate (best-effort) response can be made.

There are three elements in a CPS utilising the cloud:

1) *CPS* – one or more CPS that can function autonomously, but can utilise RTBD applications in the cloud to provide value-added service;
2) *Cloud-based RTBD application* – which can accept streaming data inputs, process and store data, respond to CPS data queries;
3) *Communications between CPS and RTBD application* – essentially shared bandwidth, potentially public.

Note that whilst the CPS can function without the cloud, and meet required timing guarantees using its own resources, overall system utility can be increased by using the RTBD application in the cloud, where deadlines involved are essentially soft real-time. Private communications networks (or Service Level Agreements for network bandwidth) and private (unshared) cloud resources can significantly improve timeliness – whilst this is a crucial issue, it lies outside the scope of the paper.

B. Challenges

From a real-time CPS perspective, key challenges of many RTBD applications include:

- *Programming* – traditional approaches for programming large parallel or cloud platforms are based upon standard programming languages that are not amenable to real-time, and that abstract (or virtualise) the underlying platform. The former problem can be addressed by utilising real-time programming languages, eg. Real-time Specification for Java [16]. The latter problem is also important, as it prevents applications from exploiting features of the platform (eg. parallelism, accelerators) that could increase performance. Thus sufficient visibility of the platform is required by the programmer in order to exploit the platform without losing the power of abstraction.
- *Timing guarantees* – a number of issues arise in RTBD cloud applications that can compromise the ability to guarantee timing behaviour, including: cloud platforms offer a dynamic platform to an application, where the available number of CPUs and system performance available to the application can vary over time; storage can vary over time; processing and storing high bandwidth streaming I/O in the context of commodity operating systems (eg. Linux); accessing large file systems.
- *Scalability* – exploitation of the parallel cloud platform such that if more resources are available (ie. more CPUs) the RTDB application can scale to use these resources without compromising any timing guarantees given. This also has impact on the programming approach.

We note that current approaches to building RTBD applications do not adequately support real-time performance, or even provide any level of guaranteed performance. Often, real-time performance is sought by increasing the overall performance of the platform, hence increasing the raw speed of response to any request. However, such an approach is unlikely to be able to offer real-time guarantees regarding the speed of response, and is often expensive in terms of cost and power, as more and more resources are included in the platform in the hope of it being sufficiently fast to offer the illusion of real-time.

III. JUNIPER Overview

The JUNIPER project [4] constructs cloud infrastructure from real-time technologies, using real-time principles, so that appropriate guarantees can then be given regarding the performance of applications running in the cloud. In turn, this enables CPS using that cloud infrastructure to then deduce overall timing behaviour when using cloud services (noting communications issues outlined in section II).

The intuition behind the JUNIPER approach is the observation that traditional real-time systems approaches enable real-time guarantees to be given to applications executing upon a platform with limited resources – ie. the amount of resource (eg. processor, I/O) that can be allocated to an application processes is known *a priori*; offline analysis allows guaranteed levels of service to be established. JUNIPER applies real-time principles to RTBD systems, so that levels of performance can be guaranteed within a cloud context. We note that JUNIPER is not aiming to provide *hard* guarantees, concentrating upon *soft* or *best-effort* approaches instead – largely due to the nature of RTBD cloud platforms (as outlined in section II) where resources in the cloud are essentially shared with other dynamic workloads, the commodity nature of the platforms themselves, and the unbounded nature of the algorithms and applications run in the cloud on large data-sets. Clearly, if platforms are constrained (eg. closed RTBD systems), then tighter bounds on timing performance can be obtained.

When constructing an RTBD cloud system using real-time technologies developers can know that real-time constraints will be met prior to the system running. This is in contrast to conventional cloud system design, where the increase of performance by adding processing power is not fully understood until the system runs. When using real-time technologies, the scalability of a system is better managed, as the effect of adding resources (eg. processing power) can be understood before the system is changed.

The JUNIPER conceptual tool-flow is illustrated in Figure 1. Cloud applications are modelled in UML, using the Marte subset to represent real-time and platform architecture aspects, allowing real- time constraints to be modelled. The target language is Java, adopting the Real-time Specification for Java restrictions [16] (see section IV). To increase performance, we ensure that the parallel nature of the hardware platform is suitably abstracted to the application, and we accelerate

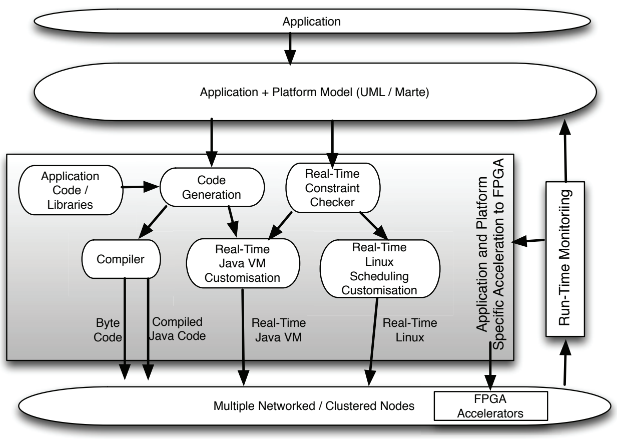
Application
Application + Platform Model (UML / Marte)
Application Code / Libraries
Code Generation
Real-Time Constraint Checker
Compiler
Real-Time Java VM Customisation
Real-Time Linux Scheduling Customisation
Application and Platform Specific Acceleration to FPGA
Run-Time Monitoriing
Byte Code
Compiled Java Code
Real-Time Java VM
Real-Time Linux
Multiple Networked / Clustered Nodes
FPGA Accelerators

Fig. 1. JUNIPER Conceptual Tool-Flow

key parts of the Java application, run-time and libraries (see section V).

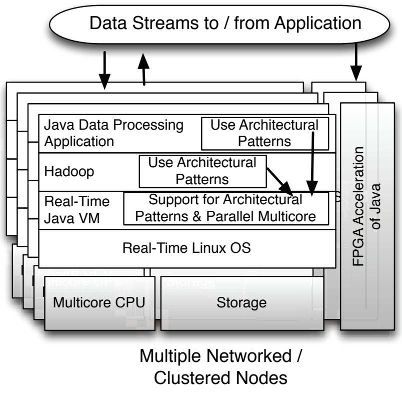
Data Streams to / from Application
Java Data Processing Application
Use Architectural Patterns
Hadoop
Use Architectural Patterns
Real-Time Java VM
Support for Architectural Patterns & Parallel Multicore
Real-Time Linux OS
Multicore CPU
Storage
FPGA Acceleration of Java
Multiple Networked / Clustered Nodes

Fig. 2. JUNIPER Software Architecture

The software architecture resulting from the tool-flow is illustrated in Figure 2. The Real-Time Java application is executed on a Real-time Java Virtual Machine [16] allowing for both interpreted and compiled Java (the latter for performance purposes). Appropriate support is provided for building a wide range of distributed big data applications (see section IV) – hence JUNIPER is not limited to a particular form of RTBD application such as Hadoop, Spark or Storm [6], [2], [8], [7].

At run-time, monitoring and profiling of the developed application and system infrastructure occurs. This permits further modification of run-time scheduling parameters (i.e. time and resource bandwidths allocated to individual applications and processes), and of the accelerated components (i.e. changing the components accelerated in the FPGA).

IV. JUNIPER Programming Model

JUNIPER believes that the programming challenges for RTBD applications are not well-addressed. Existing programming models are based on standard desktop programming languages and abstract hardware details (of both the target node and the inter-node communications) in a way that makes it difficult for the programmer to exploit the full power of the underlying platform. JUNIPER defines a new programming model based on Java 8 [22] [1] and the Real-Time Specification for Java [16] to enable the development of systems that can provide timing and resource usage guarantees. The programming model has the following core principles:

- It is not possible to express an entire RTBD applications at the source-code level, so elements of model-driven engineering (MDE) are employed to ease development, portability, and deployment.
- RTBD application developers need 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.
- System 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 within the software.

A complete description of the programming model is outside of the scope of this paper, so the remainder of this section will instead provide an overview of the main features and concepts of the JUNPER programming framework. More details can be found in [11], [17].

A. Overview

The model has two levels:

1) *Application*: this considers the large-scale movement of data; i.e. how data enters the application, how data moves from program to program, and where data is 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 [5]. The programs of the application use MPI for all coordination and data transfer.
2) *Program*: a node of the cluster is programmed using a single Java program running inside a single JVM (although 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 IV-B), and it makes use of reservations to ensure real-time behaviour (section VI).

The concepts of Application and Program are illustrated further in Figure 3.

The programming 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)

[1]Java 8 is the latest release of the Java language [22] and adds features to aid big data programming via streaming. Lambda expressions are included, being a way to express functional programming concepts in Java [23]. This aids in minimising data dependencies and maximising parallelism.

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 architecture of a single node is illustrated in Figure 4.

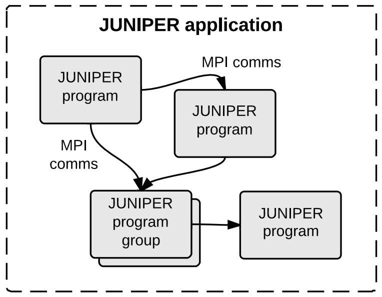

Fig. 3. A JUNIPER application is composed of JUNIPER programs, which may be unique, or one of a group of identical program instances.

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. 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 number of times (subject to optional maximum and minimum bounds) – the precise number of times being determined at run-time (eg. subject to resource availability).

B. Exposing the Architecture

Key to the JUNIPER programming model is to control the *locality* of computation and data, ie. control its proximity. This has important benefits for real-time behaviour. Within the programming model the programmer can dynamically discover the host architecture and map code to a node that is close to its data – noting that the host hardware architecture can be dynamic in a cloud environment.

Within the model, a *Locale* is the unit of allocation for mapping Java threads and objects to the CPUs and memories of the nodes. A locale is mapped to a subset of nodes within the architecture, and will remain within that subset. 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 JVMs discretion. A locale has the following properties:

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

In the JUNIPER approach, the hardware architecture is encapsulated as an architecture pattern. Patterns are used because programmers of RTBD systems are more concerned about the class of architecture than its precise details, eg. whether or not coherent caches, or whether or not all memory is of equal speed. Given the large platforms used for RTBD systems, the programmer does not require the low-level control afforded by techniques such as affinities [10], in which each thread is bound to a specific set of individual CPUs. This is onerous for large systems and lacks portability. Instead of individually mapping threads to a cloud platform, 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 run- time and infrastructure can be trusted to schedule and place the threads 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

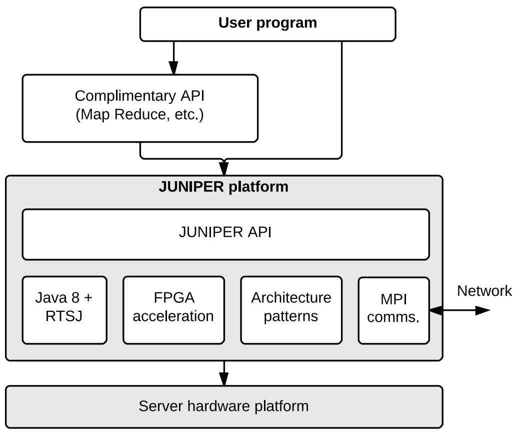

Fig. 4. A single server in a JUNIPER cluster.

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 at a greater distance from the processors.

V. Acceleration

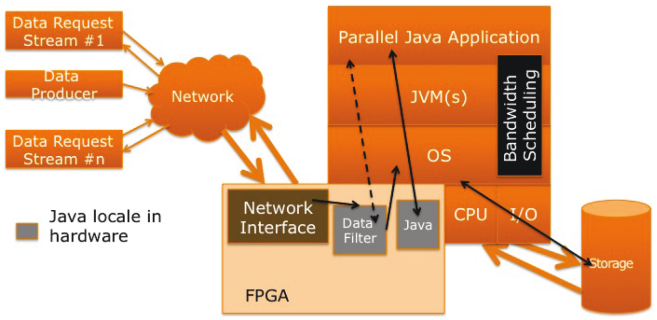

Fig. 5. Acceleration of the Programming Model using Java Components Synthesised to FPGA

JUNIPER incorporates the ability for Java application and OS components to be accelerated in hardware. An FPGA [12], [1] is provided as a part of physical architecture to give a platform for hardware acceleration – the FPGA has known physical characteristics (ie. size, memory size etc.). We note that this approach requires the provision of FPGAs within the cloud platform [2]. The JUNIPER approach makes use of these FPGAs easily, without knowledge of FPGA design.

Within the JUNIPER programming model, a parallel Java application is accelerated by placing statically identified and selected locales within the FPGA (locales placed on the FPGA remain notionally under the control of the JVM on the host CPU). This is illustrated in Figure 5, where we note:

- *Data Filter* – part of the Java application will be responsible for processing incoming data (for storage on disk), this is placed on the FPGA (alongside the network input) to allow fast processing;
- *Java Application Locales* – locales within the application (statically) selected for acceleration on the FPGA. Communication of structured data between locales on the CPU and those on the FPGA is achieved via the JUNIPER API.

This approach requires integration of the FPGA with the OS to enable efficient access to/from the FPGA for loading Java components to the FPGA; also for monitoring etc – this is considered in section VI.

The developer is required to identify the locales within the application that are amenable to static acceleration on the FPGA. Such locales are termed *Acceleratable Locale*. JUNIPER contains a subclass of the Locale class called `AcceleratableLocale` which includes an abstract method `initialise` which creates all of the threads and data that will be allocated inside that locale[3]. When an `AcceleratableLocale` is created it is assigned to a location, ie. a physical CPU or FPGA coprocessor. This is analysed ahead of time during compilation so that the code from the locale can be compiled for the FPGA.

Application locales executing as software on the FPGA require communications to hardware locales on the FPGA. This requires the ability to pass and share structured data between software and hardware. This is achieved via the JUNIPER API, which then passes data through to the FPGA via the underlying OS (see section VI).

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. JUNIPER therefore includes support for dynamic acceleration to allows the system to discover at run-time 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. We note that this makes use of FPGA partial dynamic reconfiguration techniques [27].

VI. Operating System Support

The OS support within Linux developed for JUNIPER is based upon provision of *resource reservations*. Resource reservations [24] were proved as effective techniques to achieve the goals of temporal isolation and real-time execution in open systems. Essentially, reservations allow a fraction of the bandwidth of a resource access to be reserved for a given process. JUNIPER utilises the resource reservation framework of Lipari *et al* [20] for CPUs, with further work extending the framework for access to shared resources such as disks and storage, using the M-BWI approach [15] and the Budget Fair Queueing (BFQ) disk scheduler [26], [25]. Importantly, these extensions work in conjunction with the standard `RT_PREEMPT` patch to Linux which adds preemption points to the kernel, by replacing most kernel spin-locks with mutexes that support Priority Inheritance and by moving interrupts and software interrupts to kernel threads. The `RT_PREEMPT` patch enables the Linux kernel to be more deterministic.

A. Integration of FPGAs within OS

Integration of FPGAs within the programming model (see section V) implemented upon a Linux platform requires support within Linux. JUNIPER extends Linux with additional kernel modules providing:

- support for multiple locale components to be accelerated on an FPGA simultaneously;
- communications support between Java software locales (within a user-space process) and locales on an FPGA;

[2]The JUNIPER cloud platform at York incorporates FPGAs within some of the nodes, being commodity FPGAs hosted on PCIe cards within a PC based cloud. In general, FPGAs are being incorporated increasingly into cloud platforms [14], particularly for streaming data handling [1].

[3]Normal locales can allocate threads and data freely, subject to the normal RTSJ restrictions

- communications across PCIe between kernel and FPGA board (physically located on the PCIe bus);
- a separate physical memory space for locales on an FPGA, and a means for transferring data between CPU and FPGA memory spaces;
- facility to input incoming network traffic direct to the FPGA to be processed directly by the application data filter on the FPGA before subsequently being passed to the OS for storage within the filesystem.

B. Scheduling

Linux scheduling is effectively extended by the support for resource reservations stated above. However, the scope of this is limited to a single node. Within a cloud platform, there lie higher level scheduling and allocation concerns, that is which nodes within the cloud should be allocated to an application, and how much parallelism should be employed by the application in order to utilise the available resources.

Typical approaches available from the cloud computing communities involve profiling of running applications to determine how effectively they utilise resources. This allows cloud schedulers to determine the best number of CPUs and resources to allocate to an application – hence the number of CPUs available to an application may dynamically change. However, this only effects the degree of parallelism the application can exploit. Within JUNIPER a high-level real-time scheduling advisor is being developed based on statistical analysis of worst case execution time and the analysis of data dependency on RTBD application workflows. We note that traditional real-time systems are not usually able to take advantage of run-time profiling in a live system as they tend not to have the spare CPUs to dedicate to gathering and processing profiling data.

VII. Conclusions

This paper has outlined the JUNIPER approach for building real-time big data applications for deployment in the cloud, to enable real-time guarantees to be given to CPS using the real-time big data applications. The approach addresses the following key challenges: programming, timing guarantees and scalability. The programming model ensures that sufficient detail of the architecture is available to enable developers to optimise their application; timing guarantees are enabled by using resource reservation within the OS for both time and disk access; performance is enhanced via hardware acceleration; scalability is supported by both the programming model and OS scheduling.

Whilst the paper shows some progress towards effective real-time big data applications, the challenge remains to further enhance the timeliness of such cloud applications. The current proposed approach concentrates upon soft best-effort approaches – the challenge remains to move these guarantees towards more traditional real-time guarantees (eg. hard, or weakly-hard).

Acknowledgments

This work is partially funded by the European Union's Seventh Framework Programme under grant agreement FP7-ICT-611731

References

[1] Xilinx Programmable Logic, howpublished = http://www.xilinx.com, month = September, year = 2014.
[2] Apache Hadoop Website. http://hadoop.apache.org/, 2011.
[3] Big Data Insight Group. http://www.thebigdatainsightgroup.com, September 2014.
[4] The JUNIPER Project. http://www.juniper-project.org, July 2014.
[5] The Portable Hardware Locality project (hwloc). http://www.open-mpi.org/projects/hwloc/, January 2014.
[6] V. Agneeswaran. *Big Data Analytics Beyond Hadoop: Real-Time Applications with Storm, Spark, and More Hadoop Alternatives*. Pearson, 2014.
[7] Apache Software Foundation. Apache Spark – Lightning-Fast Cluster Computing. http://spark.apache.org/.
[8] Apache Software Foundation. Apache Storm – Distributed and Fault-Tolerant Real-time Computation. http://storm.incubator.apache.org/.
[9] M. Barlow. *Real-Time Big Data Analytics: Emerging Architecture*. O'Reilly Media, Inc, 2013.
[10] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180–186. IEEE, 2010.
[11] 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.
[12] P. P. Chu. *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*.
[13] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun.*, 51,1:107–113, 2008.
[14] A. Dellson, G. Sandberg, and S. Möhl. Turning FPGAs into Supercomputers. In *Proceedings of Cray User Group (CUG) Conference*, 2006.
[15] D. Faggioli, G. Lipari, and T. Cucinotta. Analysis and Implementation of the Multiprocessor Bandwidth Inheritance Protocol. *Real-Time Systems*, 48(6):789–825, 2012.
[16] J. Gosling and G. Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.
[17] I. Gray, N. Audsley, Y. Chan, and A. Wellings. Architecture-Awareness for Real-Time Big Data Systems. In *Proceedings of EUROMPI 2014*, 2014.
[18] A. Jacobs. The Pathologies of Big Data. *Communications of the ACM*, 52(8):36–44, 2009.
[19] D. Laney. 3D Data Management: Controlling Data Volume, Velocity, and Variety. Technical report, META Group, February 2001.
[20] G. Lipari and E. Bini. A Framework for Hierarchical Scheduling on Multiprocessors: From Application Requirements to Run-Time Allocation. In *RTSS*, pages 249–258, 2010.
[21] V. Mayer-Schonberger and K. Cukier. *Big Data: A Revolution That Will Transform How We Live, Work and Think*. John Murray, 2013.
[22] Oracle Corporation. JDK 8 Schedule and Status. http://openjdk.java.net/projects/jdk8/, September 2013.
[23] Oracle Corporation. Project Lambda. http://openjdk.java.net/projects/lambda/, December 2013.
[24] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems. In *Proc. Conf. on Multimedia Computing and Networking*, January 1998.
[25] P. Valente and M. Andeolini. Improving Application Responsiveness with the BFQ I/O Scheduler. In *Proceedings 5th Annual International Systems and Storage Conference (SYSTOR'12)*, 2012.
[26] P. Valente and F. Checconi. High Throughput Disk Scheduling with Fair Bandwidth Distribution. *IEEE Trans. Computers*, 59(9):1172–1186, 2010.
[27] Xilinx. Partial Reconfiguration User Guide UG702. Technical report, July 2012.