Scalable Computing:
Practice and Experience

Scientific International Journal
for Parallel and Distributed Computing

ISSN: 1895-1767

Universitatea de Vest din Timişoara

Volume 17(4) December 2016

SUBSCRIPTION INFORMATION: please visit http://www.scpe.org

Scalable Computing: Practice and Experience

Volume 17, Number 4, December 2016

TABLE OF CONTENTS

DOI 10.12694/scpe.v17i4.1199
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 4, pp. iii–iv. http://www.scpe.org

INTRODUCTION TO THE SPECIAL ISSUE ON NEW APPROACHES FOR INFRASTRUCTURE SERVICES

The special issue is dedicated to the current research and innovation challenges encountered at infrastructure -as-a-service level generated by the desire to improve the user experiences and the efficient use of the available resources. The current trends are including the integration of special devices from high performance computing ones to mobile devices, the design of decentralised service-oriented systems, the improvement of the virtualization technologies, the overcome of portability and interoperability issues, or the automation the organisation and management of the back-end resources. Cloud-based applications from the fields of Internet-of-Things and Big Data are expected to challenge the new services.

The first paper, entitled "SLA-based Secure Cloud Application Development" reports an implementation of the concept of security service level agreements (Security SLAs) and presents a framework that allows application developers to intervine in the secure provisioning of cloud resources and services.

The second paper, with the title "Impact of Single Parameter Changes on Ceph Cloud Storage Performance", shows how a change of a global parameter of Ceph distributed file system can effect the performance for a range of access patterns when tested with an OpenStack cloud system.

The third paper, entitled "Multi-objective middleware for distributed VMI repositories in federated Cloud environment", explains the design of easy-to-use interface capable of receiving unmodified and functionally complete virtual machine images from its users, as well as of a system that transparently distribute them to a specific Cloud infrastructure in a federation achieving an improved quality of service.

The fourth paper, with the title "Architecture of a Scalable Platform for Monitoring Multiple Big Data Frameworks" is dedicated to a new, distributed, scalable software platform able to collect, store, query and process monitoring data obtained from multiple Big Data frameworks.

The fifth paper, entitled "Exposing HPC services in the Cloud: the CloudLightning Approach", refers to a novel a self-organizing and self-managing cloud service delivery system with capabilities to deliver dynamic and tailored services offered by coalitions of heterogeneous cloud resources.

Prof. Dana Petcu, West University of Timisoara

DOI 10.12694/scpe.v17i4.1200
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 4, pp. 271–283. http://www.scpe.org

SLA-BASED SECURE CLOUD APPLICATION DEVELOPMENT

VALENTINA CASOLA*, ALESSANDRA DE BENEDICTIS†, MASSIMILIANO RAK‡, AND UMBERTO VILLANO§

Abstract. The perception of lack of control over resources deployed in the cloud may represent one of the critical factors for an organization to decide to cloudify or not its own services. The flat security features offered by commercial cloud providers to every customer, from simple practitioners to managers of huge amounts of sensitive data and services, is an additional problem. In recent years, the concept of Security Service Level Agreements (Security SLAs) is assuming a key role for the secure provisioning of cloud resources and services. This paper illustrates how to develop cloud applications that deliver services covered by Security SLAs by means of the services and tools provided by the SPECS framework, developed in the context of the SPECS (*Secure Provisioning of Cloud Services based on SLA Management*) European Project. The whole (SPECS) application's life cycle is dealt with, in order to give a comprehensive view of the different parties involved and of the processes needed to offer security guarantees on top of cloud services. The discussed development process is exemplified by means of a real-world case study consisting in a cloud application offering a secure web container service.

Key words: Secure cloud applications, Security Service Level Agreements, Automatic Enforcement of Security

AMS subject classifications. 68M14, 68Q85

1. Introduction. Nowadays, the adoption of the cloud computing paradigm is steadily spreading. The final step to convince the skeptics is the provision of solid security solutions for cloud applications and data. As a matter of fact, cloud resources are not permanently assigned to users and are not under the control of user software; they are just acquired *on-demand*. This is perceived as a security loss by some users, accustomed to have full control over all the resources involved in service delivery.

In the case of public clouds, the lack of full user control over resources is not the only security issue. Currently Cloud Service Providers (CSPs), who are the actual owners of the physical computing, storage and network resources hosted in their huge data centers, administer security according to common best-practice rules. Independently of the type of Cloud Service Customer (CSCs), they provide exactly the same security features. Most often, these features are simply *the best they can offer*. The very basic security guarantees offered are undoubtedly sufficient for a private computing practitioner, but surely not adequate for small enterprises or for publicly funded organizations managing, for example, healthcare and Personal Information (PI) data to be protected with specific security and privacy requirements.

The real problem is that security has a non-negligible cost, and so CSPs have no interest in offering such features to *every* CSC. To differentiate security features on a customer-by-customer basis is difficult, if not unfeasible. Currently there is actually a gap between CSCs, which look for "tailored" security features, possibly offered *on-demand* and *as-a-service*, exactly as other cloud resources, and CSPs, which offer security *as-a-whole*, integrated in the cloud services and transparently granted in the same way for all customers.

We deem that *Security Service Level Agreements* (SLAs) can play a key role for cloud security assessment, as they allow to declare clearly the security level granted by providers to customers, as well as the constraints posed to both parties (providers and customers). However, despite the strong interest recently shown in Security SLAs in the context of both academical research and industry and government-driven initiatives, their widespread adoption is not yet a reality. In 2011, ENISA published a report analyzing the use of security parameters in Cloud SLAs (mostly focused on the EC public sector) [1]. The report pointed out that, although security was considered by most respondents as a top concern, existing SLAs addressed only availability and other performance-related parameters, while security-related parameters were not taken into account. Since then the situation has not changed significantly, and Security SLAs are still far from being adopted by existing CSPs.

The framework developed in the context of the SPECS project [2] aims to promote the adoption of Security SLAs, by making it possible to develop applications offering cloud services controlled by such contracts. With

*DIETI, University of Naples Federico II, Napoli, Italy (casolav@unina.it)
†DIETI, University of Naples Federico II, Napoli, Italy (alessandra.debenedictis@unina.it)
‡DIII, Second University of Naples, Aversa, Italy (massiliano.rak@unina2.it)
§DING, University of Sannio, Benevento, Italy (villano@unisannio.it)

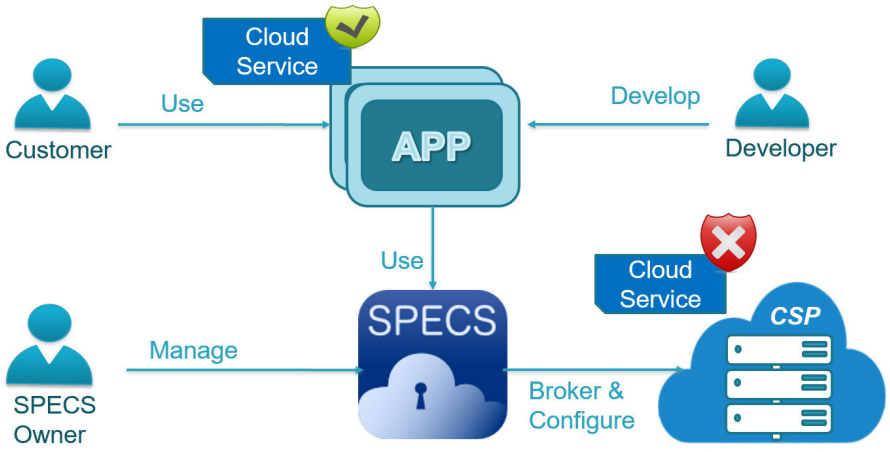


FIG. 2.1. *Overview of the SPECS solution*

SPECS, every cloud service is covered by a Security SLA that specifies the security grants offered, to be negotiated before cloud service delivery. Security features are automatically implemented by the SPECS framework according to the agreed SLA, and can be continuously monitored to verify that the SLA terms are actually respected.

The development of secure cloud applications by exploiting the SPECS framework was sketched in a previous paper [33]. In this paper, we provide a more comprehensive view of the SPECS applications' life cycle and discuss some of the tools that were developed to support it. Our exposition will go on as follows. In Sect. 2, we briefly introduce the SPECS framework and in Sect. 3 we describe the adopted Security SLA model. Sect. 4 illustrates the complete life cycle of a SPECS application, by discussing the methodology and tools adopted to enable the provisioning of secure cloud services based on SLAs. Sect. 5 discusses the introduced process with respect to a concrete example. Finally, Sect. 6 presents some related work and Sect. 7 reports our conclusions and plans for future work.

2. The SPECS framework. The SPECS project aims at designing and implementing a framework for the management of the whole Service Level Agreement life cycle, intended to build applications (SPECS applications) whose security features are stated in and granted by a Security SLA [3, 4].

The SPECS framework provides techniques and tools for: a) enabling user-centric negotiation of security parameters to be included in a Security SLA; b) enforcing an agreed Security SLA by automatically putting in place all security features needed to meet user requirements; c) monitoring in real-time the fulfillment of Security SLAs and notifying both users and CSPs of possible violations; d) reacting and adapting in real-time to fluctuations in the provided level of security (e.g., by applying proper countermeasures in case of an SLA violation).

As represented in Fig.2.1, the SPECS operation scenario involves four main parties:

- A **Customer** of the cloud services, offered by SPECS and covered by Security SLAs;
- The **SPECS Owner**, a provider of cloud services covered by Security SLAs;
- An **(External) CSP**, an independent (typically public) cloud service provider, which is unaware of the SLAs and offers just basic cloud resources and infrastructural services;
- A **Developer**, a cloud service partner that supports the SPECS Owner in the development and delivery of security-enhanced cloud services.

The Customer negotiates his/her security requirements with the SPECS Owner, who acts as a broker by acquiring resources from External CSPs and by reconfiguring/enriching them in order to fulfill the Customer's requests. This is accomplished by the activation and configuration of suitable software mechanisms and tools, provided in an *as-a-service* mode by SPECS. These mechanisms are automatically enforced on top of acquired resources, according to what has been agreed in a Security SLA. In the above process, the security-enhanced

services are delivered to end-users by a SPECS application, developed and deployed by exploiting the SPECS framework services, depicted in Fig. 2.2.

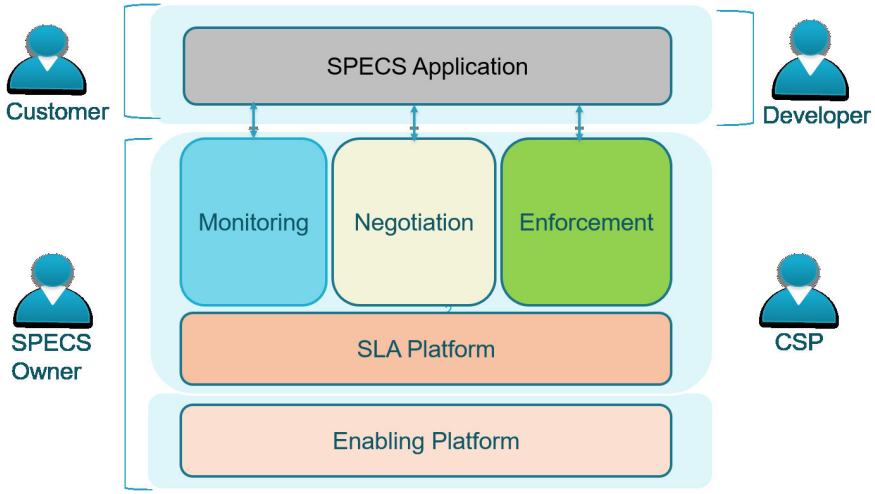


FIG. 2.2. *The SPECS framework*

A SPECS application orchestrates the SPECS *Core services* dedicated to SLA Negotiation, Enforcement and Monitoring, respectively, to provide the desired service (referred to as "Target Service") to the SPECS Customer (i.e., to the End-user). The Core services run on top of the *SPECS Platform*, which provides all the functionalities related to the management of Security SLA life cycle and needed to enable the communication among Core modules. In addition to this functionalities, referred to as "*SLA Platform* services", the SPECS Platform also provides support for developing, deploying, running and managing all SPECS services and related components [4]. These services are referred to as "*Enabling Platform* services".

3. The Security SLA model. As discussed in the previous section, the SPECS approach for *Security-as-a-Service* provisioning relies upon the idea that each cloud service is covered by a Security SLA, specifying related security-oriented terms and conditions, and that the cloud service delivery is controlled by the Security SLA life cycle. The Security SLA life cycle adopted in SPECS founds on and extends current standards on cloud SLAs (WS-Agreement [5], ISO19086 [6]) and consists of five phases: *Negotiation*, *Implementation*, *Monitoring*, *Remediation* and *Renegotiation*.

During the *Negotiation* phase, a cloud service customer and a cloud service provider carry out a (possibly) iterative process aimed at finding an agreement that defines their relationship as regards the delivery of a service. During the *Implementation*, the CSP provisions and operates the cloud service, but also sets up the processes needed for the management and monitoring of the cloud service, the report of possible failures and the claim of remedies. After the implementation of an SLA, the *Monitoring* phase takes place, where the service is continuously monitored to verify whether the SLA terms are respected. The Monitoring phase has also the responsibility of preventing, when possible, the violations, by rising alerts in presence of specific events. Alerts can be managed, during the *Remediation* phase, by reconfiguring the service while preserving the agreement, in order to avoid actual violations. If any SLA violation occurs, the cloud service customer may be entitled to a remedy (*Remediation* phase), which may take different forms, such as refunds on charges, free services or other forms of compensation. Finally, at any moment after implementation, either the cloud service customer or the cloud service provider may require a *Re-negotiation* of the SLA, aimed at changing any of its terms. The life cycle discussed above makes it possible to control cloud services according to SLA phases (and states). The interested readers are referred to [7] for a deeper analysis of the SLA life cycle and the description of a REST API for its management developed within the SPECS project, and to [8] for an illustration of some of the tools used in SPECS to monitor the SLA during the execution of a cloud service.

The negotiation, enforcement and monitoring of security-related terms are enabled by the adoption of a

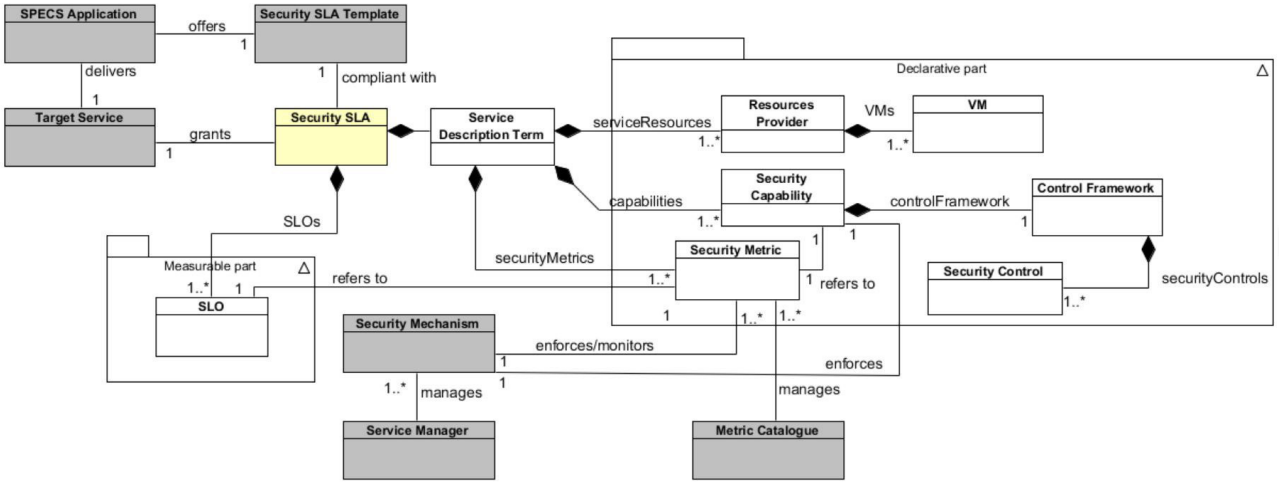


FIG. 3.1. *The SPECS Security SLA model*

novel Security SLA model, introduced in [32], which extends the WS-Agreement standard to include security concepts by taking into account both the End-user requirements and the technical offers from the providers' point of view.

The SPECS Security SLA model is depicted in Fig. 3.1, which shows more in general the *SPECS domain model*. In accordance with the WSAG standard, a Security SLA is compliant with a template (*Security SLA Template*). The template summarizes the available (and negotiable) offers and is used as a guideline during the negotiation of a *Target Service*. The whole process of the Target Service acquisition is managed by a *SPECS Application*, which is configured based on the template.

As depicted in Figure 3.1, a Security SLA (and the related template) consists of a *declarative* part and a *measurable* part. The former includes all the concepts that describe the service being delivered, both in functional and in non-functional terms. In particular, it reports the information regarding:

- the cloud resources used to build the *Target Service*. Note that, in SPECS, only Infrastructure-as-a-Service (IaaS) cloud resources are used (i.e., Virtual Machines, VMs), and the Target Service is built by properly deploying and configuring software components on the acquired VMs. For this reason, the SLA must contain the reference to the considered `Resource Provider(s)`, and the related offered `VMs`;
- the `Security Capabilities` [9] offered/required on top of the service covered by the agreement, each defined in terms of related security controls belonging to a `Security Control Framework` (NIST's Control Framework [9] and Cloud Security Alliance's Cloud Control Matrix [10] are currently supported);
- the `Security Metrics` that can be used to enforce (i.e., configure) and monitor different aspects of the declared security capabilities. Security metrics are specified in the *SPECS Security Metric Catalogue* and used to define security-related guarantees.

The measurable part of a Security SLA includes the specification of the guarantees expressed on the Target Service, represented by a set of Service Level Objectives (`SLOs`) built on top of the security metrics declared above. During negotiation, through the SPECS application, the End-user selects a subset of the available security capabilities, chooses the metrics of interest and defines SLOs on top of them.

The enforcement of security capabilities and the monitoring of related security metrics (as specified in the SLOs) is performed by software tools called *Security Mechanisms*: they are selected, deployed and configured during the *Implementation* phase. The *Service Manager* maintains all the information associated with available security mechanisms that are needed to automate their deployment and execution together with the Target Services.

4. Life cycle of a SPECS application. As discussed, a SPECS application enables an End-user to acquire an up-and-running secure cloud service after a negotiation process based on a pre-defined Security SLA template. The cloud service is delivered with specific security guarantees, which can be verified by the End-user

through monitoring functionalities, also made available by the SPECS platform.

In [33] we briefly illustrated the process of developing a SPECS application. In this paper, we aim at providing a more comprehensive view of the SPECS applications' life cycle that, at current state, is more mature and is supported by several tools. Like for any application, the SPECS application life cycle consists of three phases, i.e., (i) development, (ii) deployment, and (iii) execution. The actors involved in the first two phases are the SPECS Owner, who acquires the SPECS framework and uses it to offer secure services to his/her End-users, and the developer, at the service of the SPECS Owner, who is responsible for the implementation of the software tools needed to build secure services. The execution phase involves the interaction between the application and the End-users during the negotiation, enforcement and monitoring phases.

In the following, we discuss in detail the SPECS application's life cycle, with the aim of providing the reader with a deep understanding of the steps and tools needed to deliver secure services based on Security SLAs.

4.1. Development of a SPECS application. In order to support the development process, the SPECS framework provides a *default SPECS application* in the form of servlets for Apache Tomcat, which includes the basic functionalities to orchestrate the SPECS core services and enable the negotiation, enforcement and monitoring of an SLA, independently of the service to offer. To provide a specific Target Service, the developer must customize the default application by configuring a set of additional services that implement both the functionalities (e.g., a web container service, a database service) and the security features that the SPECS Owner is willing to offer. In order to automatize the deployment of such mechanisms, SPECS uses a cloud automation technology, represented by the *Chef* deployment solution [12], which automates the process of building, deploying, and managing software over ICT infrastructures. With Chef, it is possible to automate the deployment and configuration of a given software component on a resource such as a VM by specifying the operations to perform inside a *recipe*. Recipes are collected in *cookbooks* and stored in a Chef Server, which is responsible for launching their execution on specific nodes (hosting a Chef Client) in order to configure them. The SPECS Enforcement module includes a Chef Server, which is responsible for the set-up, at run time and based on an SLA, of all the software components needed to deliver a negotiated cloud service along with required security and monitoring mechanisms. Hence, customizing the SPECS default application implies supplying to the Enforcement module the needed cookbooks for each mechanism to support, and providing it with all the information needed to automatically configure the mechanism during the SLA implementation phase (i.e., the *mechanism descriptor*, described later).

FIG. 4.1. *SPECS application development process*

The whole development process is depicted in Fig. 4.1. It consists in the following steps:

1. **Cloud Service Definition**: the developer identifies the functionalities that should be offered by the application (e.g., web containers, databases) and implements (or retrieves, if already available) the software mechanisms that provide them *as-a-service*. Moreover, for these mechanisms, the developer prepares related cookbooks.
2. **Security Mechanisms Definition**: the developer identifies the security capabilities that should be offered by the application over the cloud services defined at the previous step, and implements (or retrieves, if already available) the related security mechanisms. Afterward, the developer prepares the mechanisms' cookbooks. Moreover, for each mechanism, the developer has to prepare a *mechanism descriptor* that specifies:
   - the granted security capabilities (and related security controls);
   - the enforceable/monitorable security metrics (and related measurements, representing the actual parameters gathered to check the identified metrics);

- the monitoring events associated with reported measurements, used by the Enforcement module (Diagnosis component) to detect violations or alerts related to an SLA;
- the mechanism's metadata, which includes information on the software components implementing the mechanism and on respective deployment constraints (e.g., incompatibility or dependency of software components implementing the mechanism, used during the mechanism's deployment).

This information is used during the whole SLA life-cycle, since it enables to negotiate capabilities, select available metrics, configure and monitor related mechanisms and detect and manage related alerts or violations.

3. **Security SLA Template Preparation**: once all the mechanisms needed to build the target cloud service have been defined and set-up, the developer prepares an SLA template, compliant with the model discussed in Sect. 3, which summarizes all available features.

It is worth noticing that the SPECS application development mainly focuses on the development of ad-hoc Chef cookbooks for the security mechanisms to be offered. When cookbooks are already available (there are many archives of already-developed cookbooks), the only additional work consists in the preparation of the metadata and SLA templates used to automate the SLA implementation.

4.2. Deployment of a SPECS application. The deployment of a SPECS application is performed through the SPECS Platform Interface, publicly available at [34]. The functionalities available to the SPECS Owner by means of the Platform Interface are reported in the Use Case diagram of Fig. 4.2.

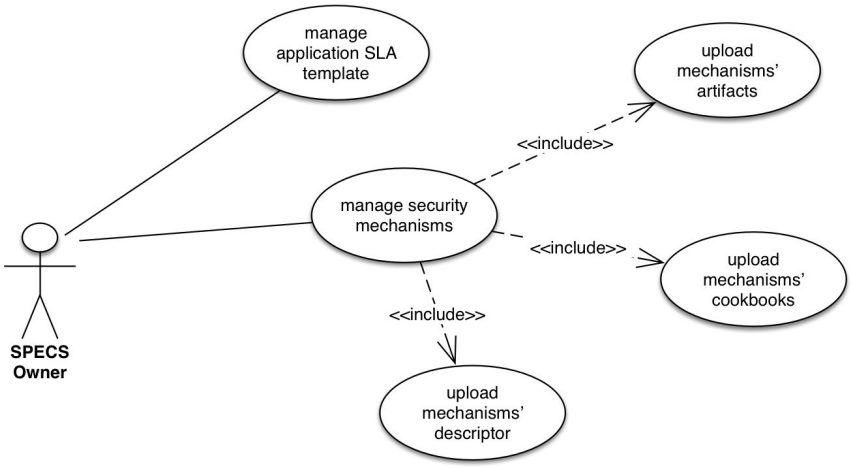


FIG. 4.2. *SPECS Owner use case diagram*

By selecting the "SPECS Services management" tab (see Fig. 4.3), the application allows the SPECS Owner to manage all the available (secure) services, along with related capabilities and security mechanisms. As discussed before, each Security Service is identified by an SLA template, prepared by the developer during the application development phase. At deployment time, the SPECS Owner must provide the template to the Negotiation module via the interface offered by the dashboard. It will be used during negotiation for the generation of the SLA Offers and during enforcement for the configuration of the Monitoring module based on included SLOs. Moreover, at deployment time, the SPECS Owner has to provide the Enforcement module with the cookbooks previously prepared for all supported mechanisms and with related artifacts. Finally, the SPECS Owner has to make available the mechanisms' descriptors to the SLA Platform, in order to enable their automatic deployment and configuration based on an SLA.

4.3. Execution of a SPECS application. A running SPECS application comes in form of a wizard that enables the End-user to negotiate, implement and monitor an SLA (cf. Fig. 4.4).

First, the negotiation wizard allows the End-user to select the security capabilities to activate. Related to these capabilities, the subsequent steps require the selection of the security metrics of interest and the definition

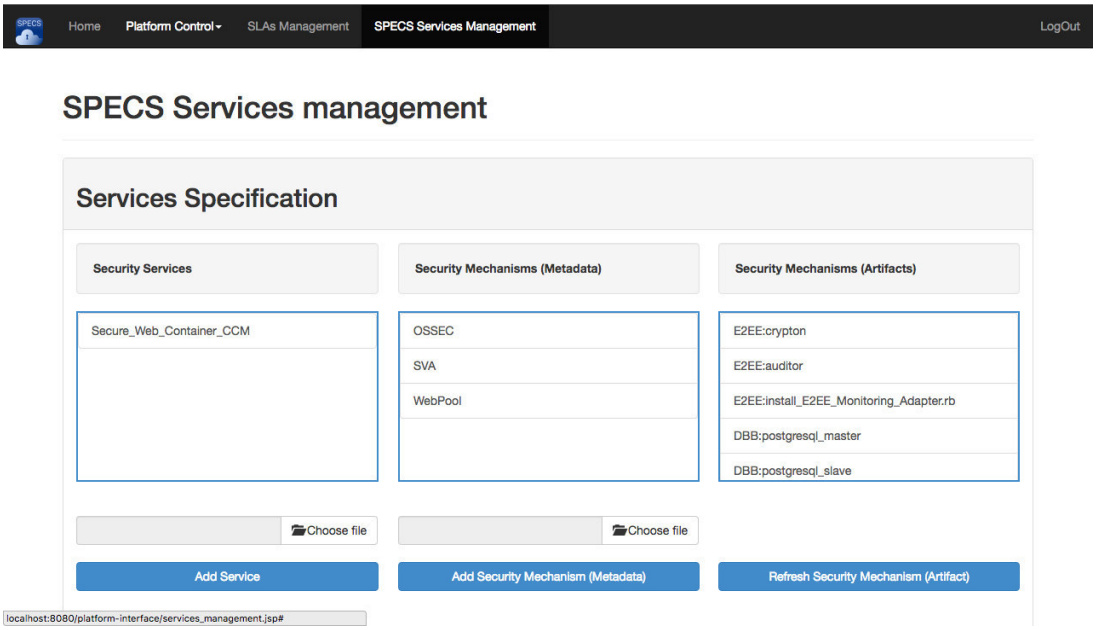


FIG. 4.3. *SPECS platform interface*

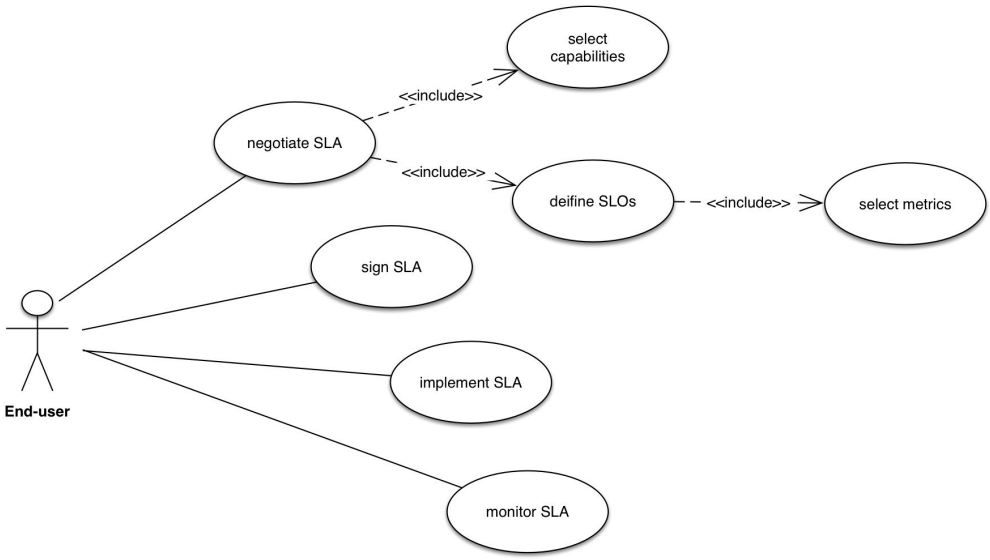


FIG. 4.4. *End-user use case diagram*

of the SLOs. Currently the negotiation focuses only on the SPECS-supported Security SLOs. However, it is possible to extend it to other non-functional SLOs. At the end of this process, the End-user can formally accept the SLA (i.e., sign it) and proceed with its implementation.

During implementation, the SPECS application orchestrates the Enforcement module's services to acquire the needed resources from external providers and to configure them with (i) the security mechanisms that implement the security capabilities included in the SLA and with (ii) the monitoring systems able to monitor the metrics reported there.

After the implementation, the SPECS application provides the End-user with a monitoring dashboard,

through which he can verify the values of the metrics and check the correct fulfillment of the SLA.

In the following section, we will illustrate the above discussed process with respect to a concrete application offering a secure web container.

5. A secure web container service. As an example of cloud service that may be enhanced through SPECS, let us consider a web container solution. An example of such a solution is Amazon AWS Elastic Beanstalk [35], which allows to quickly deploy and manage applications in the AWS cloud infrastructure. This solution, which is very complex indeed, provides support for different programming languages and web containers, and comes with dedicated security management tools developed by Amazon.

When considering smaller providers, it is reasonable to suppose that they would offer more simple platforms for web applications management, with very limited security features. A web developer targeting such providers but with specific security requirements should get on all the responsibility of managing them by developing and integrating ad-hoc software tools inside his/her applications.

It should be noticed that, at the state of the art, existing appliances offer predefined services (for example, a pre-configured web server), but checking and comparing the security features offered by different CSPs is not an easy task. The web developer has to (i) manually find the security features provided by each CSP, (ii) evaluate and compare existing offers, (iii) apply a suitable configuration, if not natively supported, and (iv) implement a monitoring solution to verify at runtime the respect of the security features.

The SPECS ecosystem provides a turnkey solution to the above issues, as it (i) offers a single interface to choose among multiple offerings on multiple providers, (ii) enables the web developer to specify explicitly the needed security capabilities on the target web container, (iii) automatically configures the VMs in order to enforce the security controls requested, (iv) offers a set of security metrics to monitor the respect of the security features requested, (v) enables continuous monitoring of the security metrics negotiated, and (vi) can automatically remediate to (some of the) alerts and violations that may occur to the SLA associated to the web container.

Below we will present the development of the secure web container as a SPECS application, following the steps dealt with in the previous section.

5.1. Cloud Service Definition. The main goal of this case study is to deliver web containers that an end-user can acquire by negotiating his/her desired security features. To this aim, we developed a mechanism named *WebContainerPool* (WEBPOOL) that not only provides the web container as a cloud service but also offers some basic security-hardening features on top of it. In particular, the mechanism allows to acquire a pool of virtual machines and to configure them with several replicas of the web container with different software solutions (e.g., Apache Tomcat, NGINX, Jetty), in order to ensure resiliency to failures through sw diversity and redundancy. Moreover, the mechanism enables to configure such replicas each with a different software solution, and to randomize the handling of incoming requests among the available replicas.

In practice, the *WebContainerPool* mechanism has been developed as a security mechanism, but it is mandatory for the set-up of the web container service delivered by the application. The information related to the mechanism has been included in the WEBPOOL mechanism descriptor, which specifies:

- the capability provided (*Web Resilience*),
- the metrics associated ((i) `LevelofRedundancy` and (ii) `LevelofDiversity`),
- the monitoring events that can be detected by the Monitoring module and handled by the Enforcement module for remediation activities (e.g., a web container replica is down),
- the actions to perform to prevent and manage violations (e.g., acquire and configure a new machine), and
- the mechanisms' metadata including the software components that implement it (i.e., Apache and Ngnix web containers and a load balancer based on HAProxy) and their deployment contraints (e.g., the load balancer must be hosted by a separate machine).

The Chef cookbook associated to the *WebContainerPool* mechanism can be used also independently of the SPECS framework and is available at [36]. It is worth pointing out that, if the aim is to apply the same process to a different cloud service (e.g., a Secure CMS), it is first necessary to develop a Cloud service cookbook dedicated to offer the CMS, and later on to select the security mechanisms that can be offered for it, possibly developing custom ones.

5.2. Security Mechanisms Definition. The proposed service (web container), as outlined above, relies on (a pool of) virtual machines, hosting synchronized web servers. The service offers some integrated security features (redundancy and diversity), but a lot of additional security capabilities can be provided. In SPECS, three main security mechanisms are already available to enhance the web container service:

- **TLS**: it is a preconfigured TLS server, configured according to security best practices.
- **SVA (Software Vulnerability Assessment)**: it regularly performs vulnerability assessment over the virtual machines, through software version checking and penetration tests.
- **DoSprotection**: it consists of a solution for denial of service attacks detection and mitigation based on the OSSEC tool.

The main role of the developer is to select the mechanisms to be provided together with the WEBPOOL mechanism from the catalogue of available security mechanisms (mantained by the SPECS Service Manager). If the developer is interested in offering additional security mechanisms, and/or in enforcing security metrics and capabilities not yet supported in SPECS, he has to implement the mechanism by releasing related artifacts, to prepare a mechanism's descriptor in the proper format and to develop a cookbook for it. Let us assume that the *SVA* and *DoSprotection* mechanisms are to be offered. The main information related to these mechanisms is provided in Table 5.1 and in Table 5.2, which respectively report the security controls and the metrics associated with them, these can be offered and guaranteed in the SLA. Moreover, Table 5.1 reports also the information that the WEBPOOL mechanism is required for the set-up of the service, while the others are optional. Table 5.2 reports the measurable information associated with the three mechanisms that will be offered in the SLA during the negotiation phase. For the sake of brevity, we do not report here all the information included in the mechanisms' descriptors. The interested reader is referred to the SPECS Bitbucket repository [15] for all the details.

TABLE 5.1
Definition of the capabilities to offer with the web container

ID	Name	Description	Req.	Controls
WEBPOOL	Web Resilience	Capability of surviving to security incidents involving a web server, by implementing proper strategies aimed at preserving business continuity, achieved through redundancy and diversity	yes	CONTINGENCY PLAN - CP-2
				HETEROGENEITY - SC-29
				DISTRIBUTED PROCESSING AND STORAGE - SC-36
				DENIAL OF SERVICE PROTECTION - SC-5
OSSEC	DOS Detection and Mitigation	Capability of detecting and reacting to security attacks aimed at disrupting a system's availability	no	DENIAL OF SERVICE PROTECTION - SC-5
				DENIAL OF SERVICE PROTECTION - SC-5(3)
				CONTINUOUS MONITORING - CA-7
				INFORMATION SYSTEM MONITORING - SI-4
SVA	Software Vulnerability Assessment	Capability of detecting and mitigating vulnerabilities	no	CONTINUOUS MONITORING - CA-7
				CONTINUOUS MONITORING — TREND ANALYSES - CA-7(3)
				PENETRATION TESTING - CA-8
				VULNERABILITY SCANNING - RA-5
				VULNERABILITY SCANNING — UPDATE BY FREQUENCY - RA-5(1)

5.3. Security SLA Template Preparation. This is the main developer task, as it summarizes all the possible offers to the End-user. Once the template is available, the SPECS application execution is fully automated. WS-Agreement templates are written according to the SLA model proposed in Section 3, following the WS-Agreement schema and the SPECS security extensions. The XML schema corresponding to our Security SLA model is available at [13], while the complete SLA template for the SPECS Web Container application is available at [37].

5.4. Application deployment. To complete the deployment of the case study application, the security mechanisms and the template have to be deployed.

TABLE 5.2
Definition of the security metrics associated with the mechanisms

Name	**Unit**	**Value Type**	**Metric Type**	**Description**
Level of Redundancy	n/a	integer	Quantitative/ Ratio	Number of replicas of a software component that are set-up and kept active during system operation
Level of Diversity	n/a	integer	Quantitative/ Ratio	Number of different versions of a software component that are set-up and kept active at the same time during system operation
Scanning Frequency - Basic scan	hours	integer	Quantitative/ Ratio	Frequency of the basic software vulnerability scanning.
Age of scanning report - Basic scan	hours	integer	Quantitative/ Ratio	Age of the scanning report (basic scan)
Vulnerability list availability	n/a	yes/no	Qualitative/ Nominal	Availability of the vulnerability list
Scanners availability	n/a	yes/no	Qualitative/ Nominal	Availability of the installed scanners
List Update Frequency	hours	integer	Quantitative/ Ratio	Frequency of updates of the list of known/disclosed vulnerabilities from OVAL/NVD databases
Age of vulnerability list	hours	integer	Quantitative/ Ratio	Age of the vulnerability list
Vulnerability repository availability	n/a	yes/no	Qualitative/ Nominal	Availability of the vulnerability repository
Scanning Frequency - Extended Scan	hours	integer	Quantitative/ Ratio	Frequency of the extended software vulnerability scanning. Scanning is performed with two scanners and both scanning reports are joint
Age of scanning report - Extended Scan	hours	integer	Quantitative/ Ratio	Age of the scanning report (extended scan)
Up Report Frequency	hours	integer	Quantitative/ Ratio	Frequency of checks for updates and upgrades of vulnerable installed libraries.
Age of the update/upgrade report	hours	integer	Quantitative/ Ratio	Age of the update/upgrade report
Availability of the scanning report	n/a	yes/no	Qualitative/ Nominal	Availability of the scanning report
Availability of the update/upgrade report	n/a	yes/no	Qualitative/ Nominal	Availability of the update/upgrade report
Penetration Testing Activated	n/a	yes/no	Qualitative/ Nominal	Activation of the penetration testing activity
DDoS Attack Detection Scan Frequency	hours	integer	Quantitative/ Ratio	Frequency of the dDoS attack report generation
Age of dDoS report	hours	integer	Quant/Ratio	Age of the dDoS attack scanning report

As said, this operation is accomplished by the SPECS Owner through the Platform Interface and entails that:

- All cookbooks of the chosen security mechanisms are added to the Chef repository associated to SPECS implementation component.
- All cookbook metadata are made available on the SLA Platform, which offers a simple REST API to upload such data and check them.
- The application template is uploaded to the Negotiation module.

The above example is available online as demonstrator application at [14].

6. Related Work. The large adoption of cloud computing solutions in a wide variety of domains opens several security issues: customers must often face the loss of control over their data and have to trust that their applications are securely executed on providers' resources. As pointed out in [16], many *ad hoc* security solutions have been proposed to cope with these issues, but they are often not portable and even not useful in different contexts.

Recent approaches are trying to address security problems from the start, i.e., at application design time, since it is hard (and ineffective sometimes) to add security features to an existing application *a posteriori*. Mohammadi *et al.* are working on the development of applications that can provide *trustworthiness* (the assurance that the system will perform as expected [17]) by design [18, 19]. The idea is to design the software in a way so that there will be mechanisms to ensure, evaluate and monitor trustworthiness, relying on reusable development process building blocks, consisting of method descriptions (guidelines, patterns and check-lists) ensuring that the right mechanisms are put in place to ensure trustworthiness.

However, as discussed previously in this paper, effective solutions exist that enable to profitably enhance the level of security of a cloud application by adopting a Security-as-a-service approach. In particular, the FP7-ICT Programme project SPECS addressed cloud security and proposed an open source development framework and a running platform dedicated to offer Security-as-a-Service using an SLA-based approach, by enabling negotiation, continuous monitoring and enforcement of security [3, 4].

As said, SPECS strongly relies upon Security SLAs. The definition of Service Level Agreement is an active topic for standardization bodies, because they are at the interface between cloud user needs and the services and features that cloud service providers (CSPs) are able to offer. The European Commission has set up a dedicated Working Group (CSIG-SLA) to cope with the definition and usage of SLAs, whose first result was a guideline for standardization bodies; a more advanced state of SLA standardization is offered by ISO 19086 [6], which proposes a standard for SLAs in clouds. However, standards for the definition of the security terms in an SLA are still lacking, even if there is currently a lot of ongoing work by dedicated groups (as the CSIG itself and the CSCC SLA group [20]) and research projects (see CUMULUS [21], A4Cloud [22], and SPECS [3]) on the topic.

Despite the strong impact that the introduction of Security SLAs may have on providers' profit, the main commercial IaaS providers (Amazon, Rackspace, GoGRID, etc.) currently still do not offer negotiable Security SLAs (see [28] for a survey of the SLAs offered by commercial cloud providers).

Regarding the configuration of security requirements specified through SLA documents, a few proposal exist. Karjoth et al. [29] introduce the concept of Service-Oriented Assurance (SOAS). SOAS adds security providing assurances (an assurance is a statement about the properties of a component or service) as part of the SLA negotiation process. Smith et al. [30] present a WS-Agreement approach for a fine-grained security configuration mechanism to allow an optimization of application performance based on specific security requirements. Brandic et al. [31] present advanced QoS methods for meta-negotiations and SLA-mappings in Grid workflows.

7. Conclusions and Future Work. In this paper, we illustrated a solution to develop cloud applications offering *secure* services covered by Security SLAs. The proposed solution relies upon a framework of services and tools released in the context of the SPECS EU Project and founds on the adoption of a novel Security SLA model, based on WS-Agreement standard and compliant with current standards and guidelines provided by the NIST and by CSA.

The paper provided a detailed discussion of the methodology followed in SPECS to build secure cloud applications and of the tools introduced and leveraged to support their life cycle. The discussion was also supported by the application of the proposed methodology to a real-world case study, for which a prototype implementation is available.

Our plans for future research include the development of new security mechanisms to enhance existing cloud services and the support for a wider set of cloud service types (at current state, only storage and web container services are considered). Moreover, we plan to include more sophisticated functionalities in the SPECS default application, such as the reasoning on the security level associated with different SLA offers, in order to enable customers to make a selection among different possible offers.

The SPECS applications may be deployed and offered by Cloud Service Providers, in order to define and agree on SLAs with their customers, but even by third-party providers that can act as brokers of services to enrich security capabilities of larger providers. This last delivery model may open new business opportunities, especially in those contexts (i.e., public sectors) where security represents the key factor to decide to cloudify a service.

Acknowledgments. This work has been partially supported by the FP7-ICT-2013-10-610795 (SPECS).

REFERENCES

[1] M. Dekker and G. Hogben, *Survey and analysis of security parameters in cloud SLAs across the european public sector,* Technical report. European Network and Information Security Agency, 2011.

[2] SPECS Consortium, *SPECS Project Web Site.* [Online]. Available: http://www.specs-project.eu

[3] M. Rak, N. Suri, J. Luna, D. Petcu, V. Casola, and U. Villano, *Security as a service using an SLA-based approach via SPECS,* in Proc. of the 2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom 2013), 2013, pp. 1–6.

[4] V. Casola, A. De Benedictis, M. Rak, and U. Villano, *Preliminary design of a platform-as-a-service to provide security in cloud,* in Proc. of the 4th International Conference on Cloud Computing and Services Science (CLOSER 2014), 2014, pp. 752–757.

[5] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, *Web services agreement specification (WS-Agreement),* The Global Grid Forum (GGF), 2004.

[6] International Organization for Standardization, *ISO/IEC NP 19086-1. Information Technology – Cloud Computing – Service Level Agreement (SLA) Framework and Technology – Part 1: Overview and Concepts*, 2014.

[7] A. De Benedictis, M. Rak, M. Turtur, and U. Villano, *REST-based SLA Management for Cloud Applications,* in Proc. of the 2015 IEEE 24th International Conference on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE 2015), 2015, pp. 93–98.

[8] V. Casola, A. De Benedictis, and M. Rak, *Security monitoring in the cloud: an SLA-based approach,* in Proc. of the 2015 10th International Conference on Availability, Reliability and Security (ARES 2015), 2015, pp.749–755.

[9] NIST, *NIST Special Publication 800-53 Revision 4: Security and Privacy Controls for Federal Information Systems and Organizations*, 2013.

[10] Cloud Security Alliance, *Cloud Control Matrix v3.0*, https://cloudsecurityalliance.org/download/cloud-controls-matrix-v3/

[11] NIST, *NIST Special Publication 500-307 Draft: Cloud Computing Service Metrics Description*, 2015.

[12] Chef, *Chef Tool Web Site.* [Online]. Available: http://www.chef.io/chef/

[13] SPECS Consortium, *SPECS Security SLA Model.* [Online]. Available: http://www.specs-project.eu/schema

[14] SPECS Consortium, *SPECS Web Container Demo Application.* [Online]. Available: http://apps.specs-project.eu/specs-app-webcontainer-demo_CCM/

[15] SPECS Consortium, *SPECS Bitbucket Repository.* [Online]. Available: https://bitbucket.org/specs-team/

[16] C. A. Ardagna, R. Asal, E. Damiani, and Q. H. Vu, *From security to assurance in the cloud: a survey,* in ACM Computer Survey, vol. 48, no. 1, pp. 2:1–2:50, Jul. 2015.

[17] A. Aviienis, J.-C. Laprie, B. Randell, and C. Landwehr, *Basic concepts and taxonomy of dependable and secure computing,* in IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, pp. 11–33, 2004.

[18] F. Di Cerbo, P. Bisson, A. Hartman, S. Keller, P. Meland, M. Moffie, N. Mohammadi, S. Paulus, and S. Short, *Towards trustworthiness assurance in the cloud,* in Cyber Security and Privacy, ser. Communications in Computer and Information Science, M. Felici, Ed. Springer Berlin Heidelberg, 2013, vol. 182, pp. 3–15.

[19] N. Mohammadi, T. Bandyszak, S. Paulus, P. Meland, T. Weyer, and K. Pohl, *Extending software development methodologies to support trustworthiness-by-design,* in Proc. of the CAiSE 2015 Forum co-located with 27th International Conference on Advanced Information Systems Engineering (CAiSE 2015), 2015, pp. 213–220.

[20] CSCC, *The CSCC practical guide to cloud service level agreements,* Technical report*, 2012.* [Online]. Available: http://www.cloudstandardscustomercouncil.org/webSLA-download.htm

[21] A. Pannetrat, G. Hogben, S. Katopodis, G. Spanoudakis, and C. Cazorla, *D2.1: Security-aware SLA specification language and cloud security dependency model.* Technical report, Certification Infrastructure for Multi-Layer Cloud Services (CUMULUS), 2013.

[22] S. Pearson, *Toward accountability in the cloud,* in Internet Computing, IEEE, vol. 15, no. 4, pp. 64–69, July 2011.

[23] Contrail Consortium, *Contrail Project Web Site.* [Online]. Available: http://www.contrail-project.eu

[24] mOSAIC Consortium, *mOSAIC Project Web Site.* [Online]. Available: http://www.mosaic-cloud.eu

[25] Optimis Consortium, *Optimis Project Web Site.* [Online]. Available: http://www.optimis-project.eu

[26] PaaSage Consortium, *PaaSage Project Web Site.* [Online]. Available: http://www.paasage.eu

[27] R. Kübert, G. Katsaros, and T. Wang, *A RESTful implementation of the WS-Agreement specification,* in Proceedings of the Second International Workshop on RESTful Design (WS-REST '11) ACM, 2011, pp. 67–72.

[28] L. Wu and R. Buyya, *Service Level Agreement (SLA) in Utility Computing Systems,* in Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions, IGI Global, USA, 2011, pp. 1–25.

[29] G. Karjoth, B. Pfitzmann, M. Schunter, and M. Waidner, *Service-oriented assurance, comprehensive security by explicit assurances,* in Quality of Protection, ser. Advances in Information Security, D. Gollmann, F. Massacci, and A. Yautsiukhin, Eds., vol. 23. Springer US, 2006, pp. 13–24.

[30] M. Smith, M. Schmidt, N. Fallenbeck, C. Schridde, and B. Freisleben, *Optimising Security Configurations with Service Level Agreements,* in Proc. of the 7th International Conference on Optimization: Techniques and Applications (ICOTA 2007).IEEE Press, 2007, pp. 367–381.

[31] I. Brandic, D. Music, S. Dustdar, S. Venugopal, and R. Buyya, *Advanced QoS methods for Grid workflows based on meta-negotiations and SLA-mappings,* in Proc. of the 2008 Third Workshop on Workflows in Support of LargeScale Science, 2008, pp. 1–10.

[32] V. Casola, A. De Benedictis, M. Rak, J. Modic, and M. Erascu, *Automatically enforcing Security SLAs in the Cloud,* in IEEE Transactions on Services Computing, 2016, PrePrints: doi:10.1109/TSC.2016.2540630.

[33] V. Casola, A. De Benedictis, M. Rak and U. Villano, *SLA-Based Secure Cloud Application Development: The SPECS Framework,* in Proc. of the 2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2015, pp. 337–344.
[34] SPECS Consortium, *The SPECS Platform Interface.* [Online]. Available: http://apps.specs-project.eu:8080/platform-interface/.
[35] Amazon, *Amazon AWS Elastic Beanstalk Home Page.* [Online]. Available: https://aws.amazon.com/it/documentation/elastic-beanstalk/.
[36] SPECS Consortium, *The WEBPOOL mechanism cookbook.* [Online]. Available: https://bitbucket.org/specs-team/specs-mechanism-enforcement-webpool.
[37] SPECS Consortium, *The SPECS Secure Web Container Application Template.* [Online]. Available: https://bitbucket.org/specs-team/specs-utility-xml-sla-framework.

Edited by: Dana Petcu
Received: May 10, 2016
Accepted: July 17, 2016

DOI 10.12694/scpe.v17i4.1201
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 4, pp. 285–298. http://www.scpe.org

IMPACT OF SINGLE PARAMETER CHANGES ON CEPH CLOUD STORAGE PERFORMANCE

STEFAN MEYER AND JOHN P. MORRISON *

Abstract. In a general purpose cloud system efficiencies are yet to be had from supporting diverse applications and their requirements within a storage system used for a private cloud. Supporting such diverse requirements poses a significant challenge in a storage system that supports fine grained configuration on a variety of parameters.

This paper uses the Ceph distributed file system, and in particular its global parameters, to show how a single changed parameter can effect the performance for a range of access patterns when tested with an OpenStack cloud system.

Key words: Ceph, Cloud Storage, File systems.

AMS subject classifications. 68M14, 68P20

1. Introduction. Cloud systems are becoming more and more complex and can be adopted to suit various types of use cases and users make use of cloud resources in very distinct ways. Their requirements change depending on the workload, which include services such as web-, database-, email- or directory-servers, but they can also be used for other purposes, such as virtual desktops, rendering or genome sequencing. The requirements of these services vary not only in the component of the system that is used the most, such as the CPU, memory system or networking, but also the characteristics of it.

When looking at storage systems the requirements of services vary greatly. Access patterns consist of a mixture of reads and/or writes, in sequential and/or random fashion, with varying block sizes. Depending on the application the metric with the highest importance might be the throughput, the average or maximum latency or a mixture of both. The prime motivation for the technical solution here is an attempt to answer the question of how a cloud operator can change the storage configuration to better support these specific access patterns.

The most popular open source cloud stack is the very actively developed OpenStack. In the development of the Kilo release there were 148 contributing organizations plus many independent developers [11] making over 19.000 commits [7]. It is highly customizable and supports many implementations as a back-end for the different storage services. Due to the popularity and interest in the system many companies offer solutions or interfaces that hooks into their own systems, which can be seen in the supported storage or networking back-ends. The storage back-end that will be looked at in this paper for improving performance under specific loads is the open source distributed file system Ceph[1], which was acquired by RedHat in 2014.

The options this combination offers will be subject of this paper. In Section 2 the general idea of offering distinct storage solutions will be explained. In Section 3 the OpenStack storage components will be introduced, followed by an introduction to Ceph in Section 4 and the integration of both and the offered options for performance optimizations. Section 5 is showing the results when the cluster is using different configurations. The conclusion of the experiments is presented in Section 6.

2. Motivation. OpenStack offers the option to set quotas for the I/O system of the VMs. Limits can be set for the amount of read/write/total bytes per second (B/s) or the same categories for the operations per second (IOPS). This allows fine grained limits that can limit the throughput effectively for small file (via IOPS) and large file (via B/s) access patterns at the same time.

With these quotas it is not possible to create flavours that fit a specific workload, as some important characteristics, such as latency, cannot be captured by those limiting parameters. It is therefore necessary to be able to offer storage options that can be more targeted to the use case. This ranges from using different technologies, such as conventional slow and fast spinning hard drives to solid state drives. These drives in return can be used in many different configurations that influence the respective performance. The configuration options include, among others, the storage I/O scheduler and the file system. It becomes even more challenging

*University College Cork, Ireland ({s.meyer, j.morrison}@cs.ucc.ie).
[1]http://ceph.com

when used in combination with a distributed file system, which seems to be the most common storage back-end used for OpenStack according to [10].

In our previous paper [14] we looked at the option of offering distinct storage pools using different file systems. In this paper we extend this work and look at the impact of changing a single parameter in the Ceph configuration on the performance when tested through an OpenStack cloud deployment using block storage devices. The OpenStack storage components will be described in Section 3 followed by an overview of the distributed file system Ceph in 4.

3. OpenStack Storage Components. OpenStack has three different storage services: Glance, Cinder and Swift. They cover three very different areas. The OpenStack image service Glance is an essential component in OpenStack as it serves and manages the virtual machine images that are central to Infrastructure-as-a-Service (IaaS). It offers an RESTful API that can be used by end users and OpenStack internal components to request virtual machine images or metadata associated with them, such as the image owner, creation date, public visibility or image tags.

Using the tags of the images should allow an automatic selection of the best storage back-end for an individual VM. When an image is tagged with an database tag the storage scheduler should be able to automatically select the appropriate pool to host the VM, like it does for other components, like the amount of CPU core or the memory capacity. In case the tag is absent the image will be hosted on the fall-back back-end, which uses a standard or non-targeted configuration that is used for general purpose scenarios. When users use the correct tag they will benefit from it and the operator will potentially be able to increase the number of users that he can host without risking overall storage performance degradation and different billing rates, as they are optimal solutions for specific workloads (value added features).

The OpenStack object storage service Swift offers access to binary objects through an RESTful API. It is therefore very similar to the Amazon S3 object storage. Swift is a scalable storage system that has been in use in production for many years at Rackspace[2] and has become a part of OpenStack. It is highly scalable and capable to manage petabytes of storage. Swift comes with its own replication and distribution scheme and does not rely on special RAID controllers to achieve fault tolerance and resilient storage. It can also be used to host the cloud images for the image service Glance.

The third storage system of OpenStack is the block storage service Cinder. Cinder is used to either create volumes that are attached to virtual machines for extra capacity that show up as a separate drive within the VM, but it can also be used directly as the boot device. In that case the image from Glance will be converted/copied into a Cinder volume. After it has been flagged as bootable it can be used as the root disk of the VM. The back-ends for Cinder cover a great variety of systems [4], including proprietary storage systems such as Dell EqualLogic or EMC VNX Direct, distributed file systems, such as Ceph or GlusterFS, and network shares using Server Message Block (SMB) or NFS.

Normally it would be necessary to have at least two dedicated storage systems available when all three storage services are desired. This does not allow the flexibility to deal with extra capacity demands on individual services as the hardware has to be partitioned when the system is rolled out. Currently there are two storage back-ends available that can be used for all three services within one system with the support for file-level storage, which is required for supporting live-migration between compute hosts. These are Ceph (see Section 4) and GlusterFS [3]. By using one of these storage back-ends it is possible to consolidate the three services on a single storage cluster and keeping them separated through logical pools. According to the OpenStack user survey in 2014 [10] Ceph has currently the highest popularity as a storage back-end, which will be looked at in more detail in the following section.

4. Introduction to Ceph. Ceph [16] has been designed under the assumption that a large peta-scale storage system is an incremental growing system, where the failure of components are not the exception but rather the norm, and where workloads are constantly changing. At the same time the storage system has to be able to handle thousands of user requests and deliver high throughput [17]. The system replaces the traditional interface to disks or RAIDs with object storage devices that integrate intelligence to handle specific operations

[2] www.rackspace.com
[3] www.gluster.org

themselves. Clients interact with the metadata server to perform operations, such as open and rename, while communicating directly with the OSDs for I/O operations. The algorithm that is used to spread the data over the available OSDs is called CRUSH [18]. From a high level, Ceph clients and metadata servers view the object storage cluster, that consists of possibly tens or hundreds of thousands of OSDs, as a single logical object store and namespace. Ceph's Reliable Autonomic Distributed Object Store (RADOS) [19] achieves linear scaling in both capacity and aggregate performance by delegating management of object replication, cluster expansion, failure detection and recovery to OSDs in a distributed fashion. The following list shows again the individual components and their role and function:

OSDs: A Ceph OSD Daemon (OSD) stores data, handles data replication, recovery, backfilling, rebalancing, and provides some monitoring information to Ceph Monitors by checking other Ceph OSD Daemons for a heartbeat. A Ceph Storage Cluster requires at least two Ceph OSD Daemons to achieve an `active + clean` state when the cluster makes two copies of your data (Ceph makes 2 copies by default, but it is adjustable).

Monitors: A Ceph Monitor maintains maps of the cluster state, including the monitor map, the OSD map, the Placement Group (PG) map, and the CRUSH map. Ceph maintains a history (called an "epoch") of each state change in the Ceph Monitors, Ceph OSD Daemons, and PGs.

MDSs: A Ceph Metadata Server (MDS) stores metadata on behalf of the Ceph Filesystem (i.e., Ceph Block Devices and Ceph Object Storage do not use MDS). Ceph Metadata Servers make it feasible for POSIX file system users to execute basic commands like `ls`, `find`, etc. without placing an enormous burden on the Ceph Storage Cluster.

Ceph is also highly customizable and offers many ways to change the configuration of the cluster. This is not just limited to the number of placement groups per pool (pgp), but spans more than 600 parameters. The value of these parameters will result in many cases from the use case, such as the number of replicas. Furthermore the storage system relies on components that are not part of Ceph directly, like the Kernel I/O scheduler and the corresponding queue length, that have an impact on the performance. How these can be used to support discrete storage pools will be discussed in Section 4.2 to 4.4.

4.1. OpenStack and Ceph. Creating pools within the cluster is necessary to allow access to the storage. These pools can in return be used for the OpenStack services Glance and Cinder. Running Swift with Ceph is also possible through the Rados gateway, which is part of Ceph, that offers a RESTful API to the objects in the cluster. As Glance is only hosting the images for the VMs it does not have any real performance requirements, especially when Cinder is used for actually running the VMs by choosing the `copy to volume` option when creating VMs. Furthermore, even though it is possible to just use Glance without Cinder and use it for booting the images, it does not offer the option to use multiple back-ends at the same time. It is a requested feature[4], but currently it is not available.

Cinder supports multiple back-ends that are attached at the same time. This offers the possibility to create different Cinder Tiers that are connected to different back-end systems with varying capabilities or features, such as having one proprietary storage system and a network share set up as the back-ends or to use different pools from Ceph or completely different Ceph clusters. In the first case the Ceph configuration file is identical and only the Ceph pool are different. In the second case it is necessary to provide multiple Ceph configurations that point to the different clusters, as the Ceph monitors will be running on separate hosts. This might be used to offer low specification versions that offer no resilience to failures for low cost solutions or high cost solid state drive solutions.

4.2. Optimization on a Cluster Level. Tuning the distributed file system on a cluster level gives access to whole range of parameters [3] of Ceph and the underlying components, such as the caching size, recovery settings, journal settings or logging. This allows for perfect adoption to a specific workload, but in contrast does not allow for differentiation and therefore multiple workloads in one cluster. When a decision is made it affects the overall system and can only be slightly tweaked by the parameters that are accessible through the pools (see 4.3).

[4] https://blueprints.launchpad.net/glance/+spec/multi-store

To offer distinct storage variants, the whole storage cluster has to be partitioned and multiple clusters have to be created. It is possible to run multiple clusters on individual hosts, but it has to be guaranteed that the services have unique IP and port combinations. Copying data from one cluster to the other cannot be achieved very easily on the Ceph level, but within OpenStack it is supported with specific limitations. It is not possible to migrate volumes that have snapshots.

4.3. Optimization using Pools and Tiering. When optimizing on a Ceph pool level without changing the Crushmap, the parameters that can be changed is limited in comparison to the options that are available when optimizing on a cluster level. In total there are 19 parameters (full list available at [13]) that change the characteristics of the pool.

Some of these parameters have a direct effect on the performance and behaviour of a storage pool, as they directly influence the pool characteristics. Parameters such as `size`, `min_size`, `pgp_num`, `crush_ruleset`, `hashpspool` and `crash_replay_interval` have a significant impact on how the pool distributes the data across the OSDs and how many copies are stored. They also directly influence reliability, stability and recovery.

Ceph offers the option to add a cache Tier to a pool. Adding fast expensive drives for highly frequented objects with slower cheaper disks that are acting as cold or slower storage. Such tiered systems are very common in enterprise storage systems, *e.g.* Dell Compellent [6], and is now available in software solutions, such as Ceph, as well. Changing the pool parameters effects the movement of the objects between the cache and normal Tier and the used caching algorithm.

The using a cache Tier it is necessary to select one of two operation modes. When using the writeback mode the client will write the data directly to the fast cache tier and will receive an acknowledgement when the request has been finished. Over time the data will be send to the storage tier and potentially flushed out of the cache tier. When a client requests data that does not reside within the cache, the data is transferred first to the cache tier and then served to the client. This mode is best used for data that is changeable, such as video, photo and transactional data.

When using read-only mode the cache will only be used for read access, where it will copy the data from the storage tier to server the read requests. Write access will be sent directly to the storage tier. This operation mode is best used for immutable, such as images and videos for webservices, DNA sequences or radiology images, as reading data from a cache pool that might contain out-of-date data provides weak consistency. For that reason it should not be used for mutable data.

Using pools for differentiation still relies on the underlying configurations, such as disk scheduler or OSD file system settings. All pools will share the same general settings with specific configuration changes for replication count or distribution enhancements. Therefore a pure pool based approach does not offer the best way to make significantly different storage configurations accessible. On the other hand, pools are a good way to partition the storage and to expose it to different users/services through the access control mechanisms of Ceph.

4.4. Heterogeneous Pools. The term heterogeneous pools in this case is used for pools that have different underlying components, such as the I/O scheduler or the file system. The I/O scheduler is set in the operating system for a specific block device, such as `/dev/sdb` and cannot be set for a specific partition. The scheduler comes along with it own settings, such as the scheduler type and the queue depth. Changing these can have a substantial influence on the performance of the file system under specific workloads [15].

The other component that has an impact on performance is the file system. Besides the different functionalities that they offer, they handle some patterns much better than others due to their internal design. Both of these aspect of a heterogeneous cluster configuration have been shown in our previous paper [14].

5. Experimental results. The experiments are focused on changing the whole storage cluster configuration. The storage pools used by OpenStack were deployed on the same hardware and software configuration. By changing the storage cluster configuration for each individual benchmark run shows the impact of each one of those individual changes in comparison to the default configuration.

5.1. Testbed. The configuration used for the practical evaluation consists of three Dell R610 servers connected to a directly attached storage expansion tray through an LSI SAS3444E SAS controller using SAS multi-lane cables, each. Each server is equipped with an Intel Xeon E5603 quad core processor with 1.6 GHz clock speed, 16 GB DDR3 memory and 8 Gigabit Ethernet ports. Each storage tray consists of twelve 1 TB

harddrives (Ultrastar A7K1000 [8], Barracuda ES.2 [2], Western Digital RE4 WD1003FBYX [12]), of which only four of each tray will be used for the experiment, but all disks are part of the cluster. The Ceph cluster network uses NIC bonding (IEEE 802.3ad [9]) with four network links for increased bandwidth. The Ceph public network uses three links in the same way. The operating system used for the testbed is Ubuntu 14.04 LTS with the 14.04.2 hardware enablement stack (Ubuntu Utopic) which uses Linux kernel version 3.16, which includes many patches for BTRFS and XFS. The mounting options used for both file systems were the default settings of Ceph. The mounting settings can have an influence on the performance of a file system, but they are not in the focus for this paper.

To run the benchmarks against the Ceph cluster an OpenStack cloud (version Kilo) is used. The OpenStack cloud consists of a dedicated cloud controller and of three Dell R710 servers with two Intel Xeon E5645 hexa cores with 2.4 GHz clock speed and Intel Hyper Threading enabled acting as compute nodes. Each server is connected to the Ceph public network with three bonded Gigabit Ethernet links. The used switch is a Dell PowerConnect 6248 [5]. Jumbo frames were enabled on the switch and all Ethernet links.

The network bandwidth between the nodes is measured using `iperf`. The results are shown in Table 5.1 with the Storage network representing the Ceph cluster network and the Management network being the Ceph public network. The 2 bonded port configuration is used on the controller to access the Ceph cluster for handling the Cinder and Glance volumes and images whereas the 3 bonded ports are used on the compute nodes to run the virtual machines directly off the Cinder volumes.

5.2. Cluster configuration. The Ceph cluster is configured to host multiple pools, pinned to different drives. The pool used for the benchmarks is isolated on the 12 Western Digital RE4 drives. To reduce the impact of the cluster network bandwidth limitation of about 3 Gb/s shown in Table 5.1 the replication count is set to 2. This ensures that each block is transferred only once through the cluster network for replication. With a replication count of three the file would be written to two other hosts which would double the network traffic and therefore reduce the write performance. In a bigger cluster the load from replication is spread and therefore the network bandwidth dependency will be less crucial overall, but in a small cluster it is a limiting factor.

The Ceph configuration for these experiments is left to the default settings and the parameters that are set can be seen in the following configuration snippet. It has the debugging and reporting function on the OSD and Monitors disabled and uses CephX for authentication.

```
[global]
osd_pool_default_pgp_num = 1024
osd_pool_default_pg_num = 1024
osd_pool_default_size = 2
osd_pool_default_min_size = 2

[client]
rbd_cache = false

[osd]
debug ms = 0
debug osd = 0
debug filestore = 0
debug journal = 0
debug monc = 0

[mon]
debug ms = 0
debug mon = 0
debug paxos = 0
debug auth = 0
```

The specific cluster wide parameters tested differ from the default configurations in exactly one parameter. This allows to detect parameters being harmful or beneficial for specific workloads. The tested parameters are listed in Table 5.2.

The selected parameters are all for settings related to the OSDs and the filestore. Using Ceph in combination

TABLE 5.1
Measured (`iperf`) bandwidth on the different networks.

Network	**Storage**	**Management**	
Bonded Ports	4	2	3
Bandwidth	3.08 Gb/s	1.96 Gb/s	2.50 Gb/s

TABLE 5.2
Tested parameter values and their default configuration.

Parameter	**Default**	**Tested**
osd_op_threads	2	1 (B), 4 (C), 8 (D)
osd_disk_threads	1	2 (E), 4 (F), 8 (G)
filestore_op_threads	2	1 (H), 4 (I), 8 (J)
filestore_wbthrottle_xfs_bytes_ start_flusher	41943040	4194304 (K), 419430400 (L)
filestore_wbthrottle_xfs_ios_start_ flusher	500	5000 (M), 50 (N)
filestore_wbthrottle_xfs_inodes_ start_flusher	500	5000 (O), 50 (P)
filestore_queue_max_bytes	104857600	1048576000 (Q), 10485760 (R)
filestore_queue_committing_max_ bytes	104857600	1048576000 (S), 10485760 (T)
objecter_inflight_op_bytes	104857600	1048576000 (U), 10485760 (V)
objecter_inflight_ops	1024	8192 (W), 128 (X)

with OpenStack Cinder and Glance does not require using components such as the RADOS Gateway, which would be necessary when using OpenStack Swift, or the metadata server (MDS).

The `osd_op_threads` parameter sets number of threads to service Ceph OSD Daemon operations. When set to 0 it will disable multi-threading. Increasing the number may increase the request processing rate. Ceph uses a 30 seconds timeout for the requests. So it depends on the used hardware if altering the parameter has a positive or a negative effect.

`osd_disk_threads` sets the number of disk threads, which are used to perform background disk intensive OSD operations. These include scrubbing, which is analogous to `fsck` on the object storage layer, and snapshot handling. This can effect the performance when the scrubbing process happens while there is concurrent data access. Otherwise only one operation can be worked on at one time.

`filestore_op_threads` sets the number of file system operation threads that execute in parallel.

`filestore_wbthrottle_xfs_bytes_start_flusher`, `filestore_wbthrottle_xfs_ios_start_flusher` and `filestore_wbthrottle_xfs_inodes_start_flusher` configure the filestore flusher that forces data from large writes to be written out before the sync in order to (hopefully) reduce the cost of the eventual sync. Previously, the filestore had a problem when handling large numbers of small ios. Ceph throttles dirty data implicitely via the journal, but a large number of inodes can be dirtied without filling the journal resulting in a very long sync time when the sync finally does happen. The flusher was not an adequate solution to this problem since it forced writeback of small writes too eagerly killing performance.

`filestore_queue_max_bytes` and `filestore_queue_committing_max_bytes` set the size of the filestore queue and the amount of data that can be commited in one operation.

`objecter_inflight_op_bytes` and `objecter_inflight_ops` modify the Ceph objecter, which handles where to place the objects within the cluster.

5.3. Performance evaluation. To evaluate the impact of the different configurations the Ceph cluster is exercised using the OpenStack cloud system described in Section 5.1. As the benchmark the tool `fio` [1] is used. The benchmark settings are set to 300 seconds runtime and a 10GB testsize. The IO engine is set to sync, which uses fseek to position the I/O location. Access is set to direct and buffering is disabled. For each run there is a start and a ramp delay of 15 seconds. Random and sequential access patterns are tested for both reads and writes. Block sizes to test are 4k, 128k, 1MB and 32MB. A total of 9 runs for each benchmark configuration is executed to achieve a representative average over multiple runs.

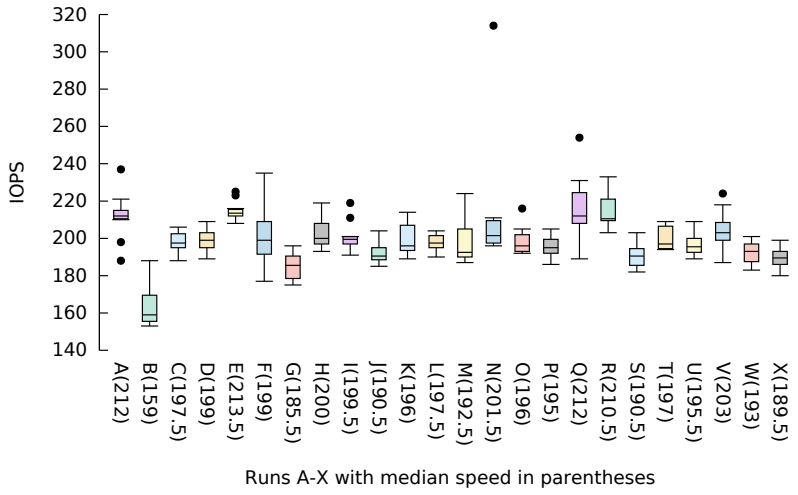


FIG. 5.1. *FIO random read 4k.*

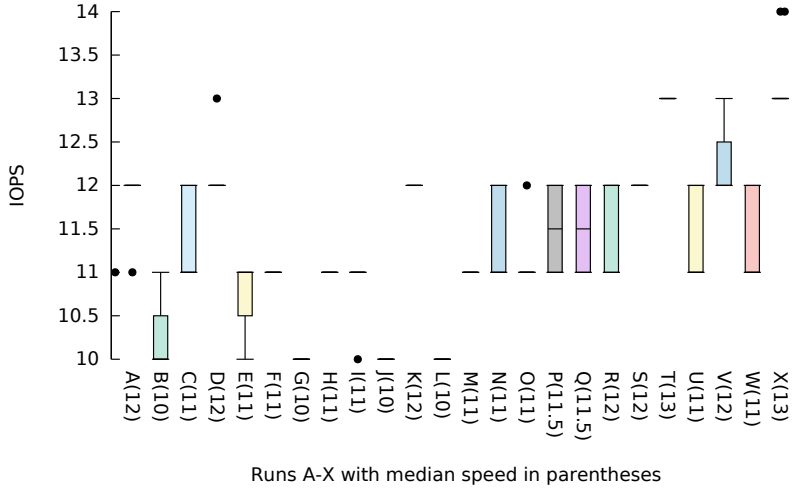


FIG. 5.2. *FIO random write 4k.*

A total of 12 virtual machines, equally distributed across the three compute hosts, is used to stress the system. Each VM is set to use 4 cores and 4GB of memory. The virtual disk is set to use a 100GB Cinder volume. Rados Block Device (RBD) caching is disabled on the Ceph storage nodes and on the compute hosts in the QEMU/KVM hypervisor settings. The diagrams show the mean value across all 12 VMs and their 9 runs.

5.3.1. 4k. The smallest file block size tested is 4k. The performance of mechanical hard drives suffers quite a lot under such loads, whereas SSDs perform much better well under such loads. The unit used to in these graph is IOPS (**I**nput/**O**utput Operations **P**er **S**econd).

Figure 5.1 and 5.2 show the performance under random access workloads for reading and writing. Under random read workloads the most disruptive configuration is with `osd_op_threads=1` (B) with 159 IOPS. In comparison to the other configurations that achieve between 189 (X) and 213.5 (E) IOPS this configuration performs 15% lower than the second lowest and 25% less than the default configuration. In comparison to the default configuration the performance can only be matched, but not be surpassed. For random writes the performance is more even with a difference of 3 IOPS between the best and the worst performing configuration.

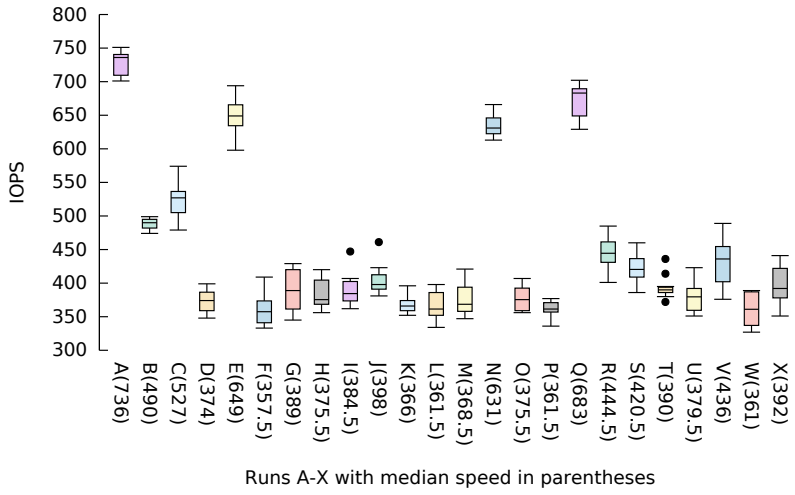


FIG. 5.3. *FIO sequential read 4k.*

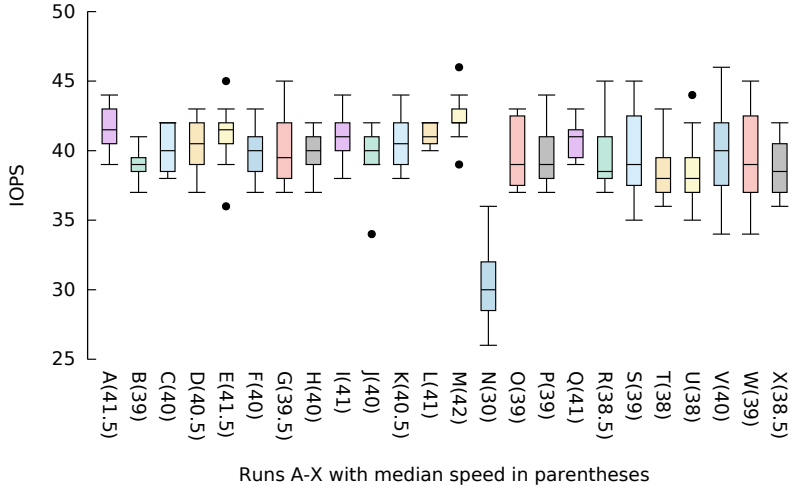


FIG. 5.4. *FIO sequential write 4k.*

Due to the low number of IOPS there can't be any real conclusions about the impact of the parameters.

When the storage system is tested against 4k sequential read access (see Figure 5.3), the difference between the lowest performing configuration (`osd_disk_thread=4` (F)) and the highest (default configuration (A)) is over 105% or 378 IOPS. Well performing configurations, but worse than the default, are `filestore_queue_max_bytes` =1048576000 (Q), `osd_disk_threads=2` (E) and `filestore_wbthrottle_xfs_ios_start_flusher=50` (N). For writing (see Figure 5.4) the results are very even again, except for configuration N, that scores well for read accesses. When this configuration is used the performance drops by 25% in comparison to the others. This means when writing small blocks the small flusher threshold is harmful, whereas it is beneficial when used for reading.

5.3.2. 128k. When using 128k block sizes for random read accesses (see Figures 5.5) all but two configuration (B and N) achieve gains over the default configuration. A maximum gain of 8% is observed for configuration K. When writing random blocks with the same block size the difference is much more significant with K being 14% faster than the default. The performance difference between `filestore_wbthrottle_xfs_bytes_start_`

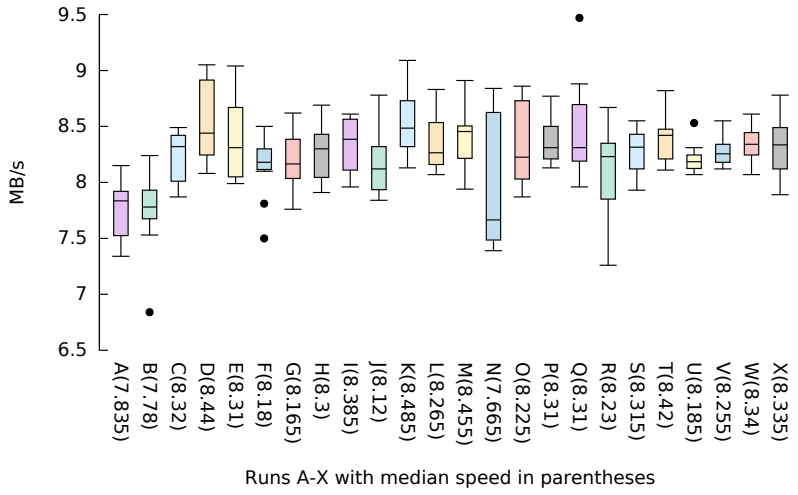


FIG. 5.5. *FIO random read 128k.*

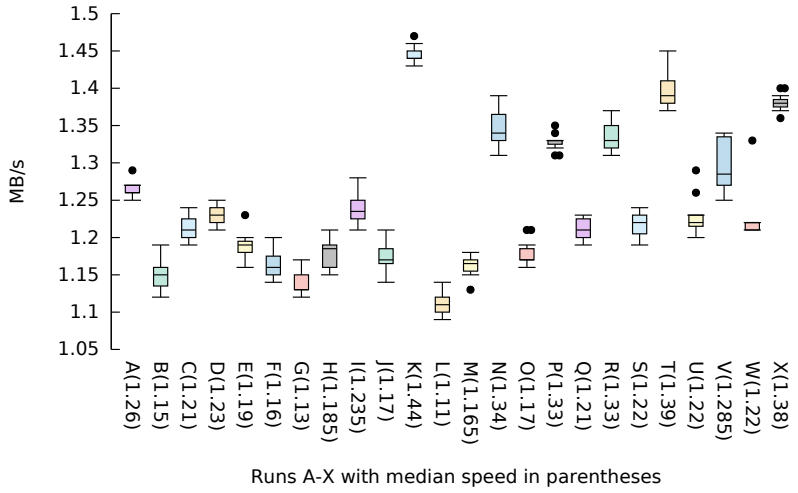


FIG. 5.6. *FIO random write 128k.*

`flusher` being 4194304 (K) and 419430400 (L) is almost 30%. The interesting part is that it is the same parameter with different values that changes the behaviour. A similar behaviour can be observed for the pairs M to X where the smaller value performs better than the larger. The default configurations performance for these runs is always in between the lower and higher configurations.

For sequential read access (Figure 5.7) the performance is very even with a difference of 9% between configuration O (lowest) and Q (highest). The default configuration is surpassed by only four configurations. For sequential write access (Figure 5.8) configurations N and K are the most disruptive, with K performing 16.5% lower than the default configuration. Gains are not observed under this workload.

5.3.3. 1MB. For random 1MB block access (Figure 5.9) `disk_threads=4` (F), `filestore_queue_max_bytes` equals 10485760 (R) and 1048576000 (Q) perform better than the rest. Configuration F is able to improve performance by 10% over the default. The most disruptive configuration is `filestore_op_threads=8` (J) that reduces performance by 13%. For random write access (Figure 5.10) configuration `filestore_wbthrottle_xfs_bytes_start_flusher=4194304` (K) improves the performance by 44.5% over the default configuration. Only configuration Q shows reduced performance. All other configurations improve performance.

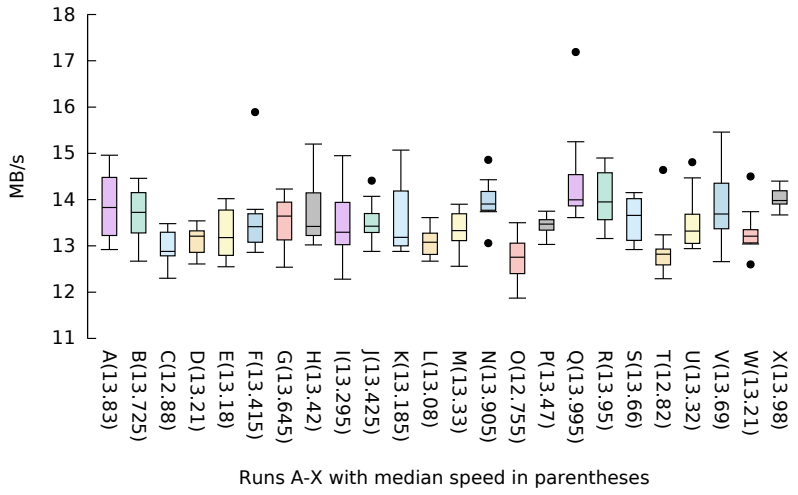


FIG. 5.7. *FIO sequential read 128k.*

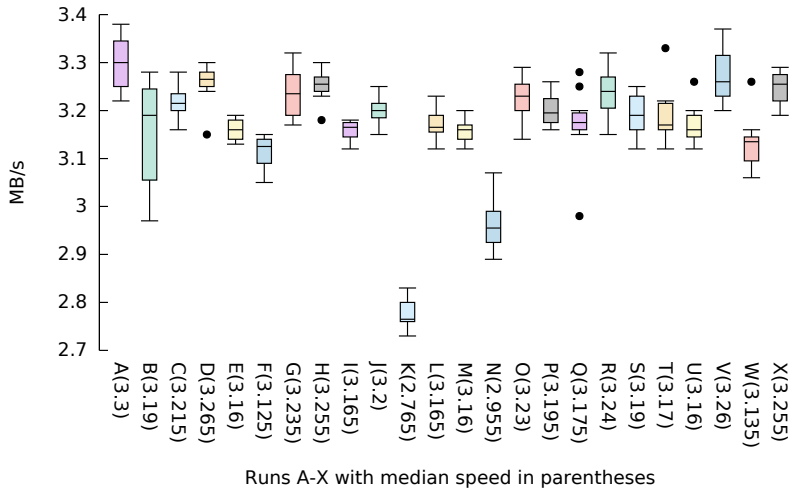


FIG. 5.8. *FIO sequential write 128k.*

Under sequential 1MB read workloads (Figure 5.11) there is no configuration that improves performance. The maximum regression observed is 8% (W). For writing (Figure 5.12) configuration K show the highest gains of 28%.

5.3.4. 32MB. In random 32MB read and write access loads, see Figures 5.13 and 5.14, configuration `filestore_queue_max_bytes=1048576000` (Q) are able to improve performance over the default configuration. For random writes the same parameter with a hundred times smaller queue size (R) is able to surpass it, but not for random reads. Configuration `filestore_wbthrottle_xfs_inodes_start_flusher=5000` (O) is the most disruptive for random reads, reducing performance by more than 2 MB/s in comparison to the default configuration. For random writes multiple configurations (E, H, O) have a strong negative impact. In comparison to the default configuration changing the parameters for 32MB random access workloads is more harmful than useful.

For sequential 32MB read access (see Figure 5.15) all configurations that deviate from the default reduce the performance by up to 14% (C) or 2.2 MB/s. Configurations B and Q reduce performance the least. With

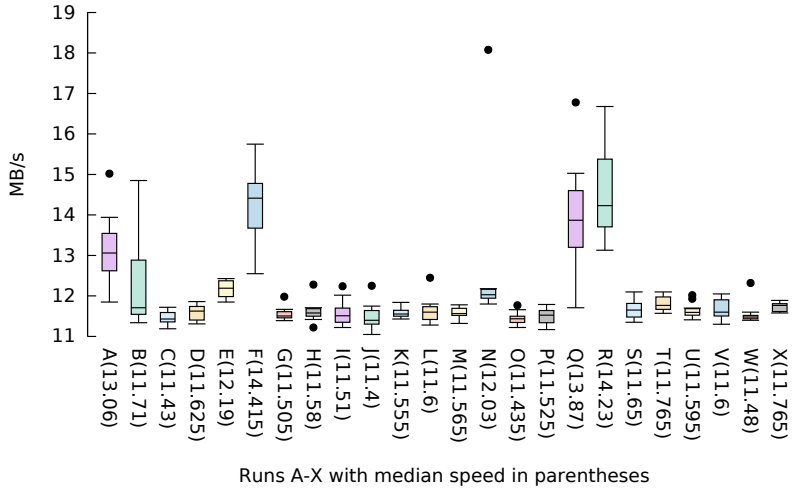


FIG. 5.9. *FIO random read 1MB.*

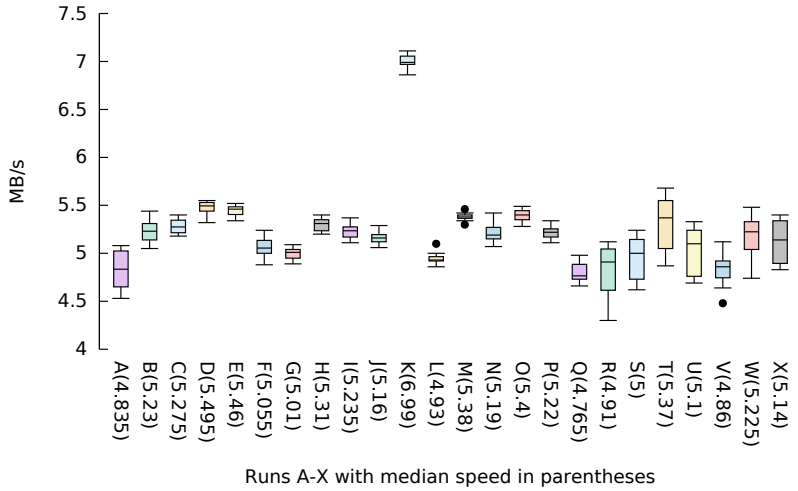


FIG. 5.10. *FIO random write 1MB.*

sequential writes (see 5.16) the results look a bit different. Configuration R can improve performance by 6%, whereas the other configurations reduce performance by up to 14.5% (L).

6. Conclusion. The experiments show that changing a single parameter of a Ceph cluster configuration can have a significant impact on performance when used as a storage back-end for virtual machines. Under specific workloads an improvement of up to 44.5% was observed. In other access patterns the improvement is not that significant, but still an improvement over the default configuration.

Configurations that showed performance degradations of up to 50% under certain workloads have also been observed. The severity of losses in comparison to the default configuration are heavily depending on the workload.

Overall the performance of the default configuration is well balanced. The other configurations were not always able to achieve better performing. Only with some access patterns the majority of the tested configurations was able to improve performance. This shows that with the used hardware configuration the default settings are working well, but leave room for improvement and are not the best possible.

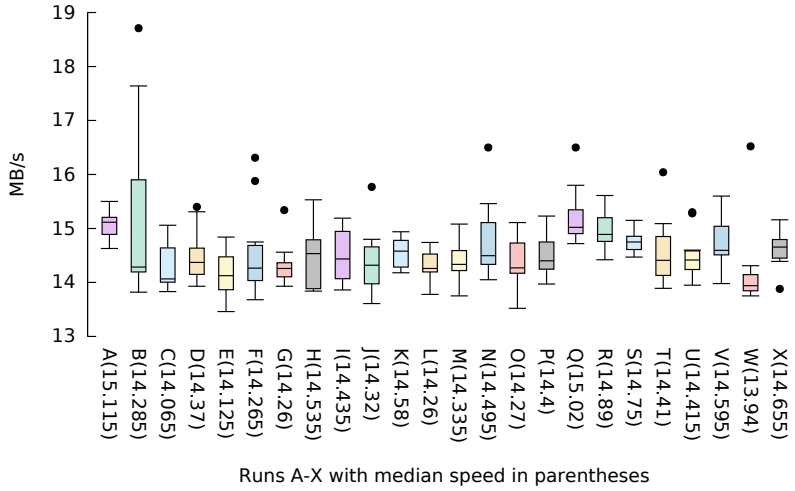


FIG. 5.11. *FIO sequential read 1MB.*

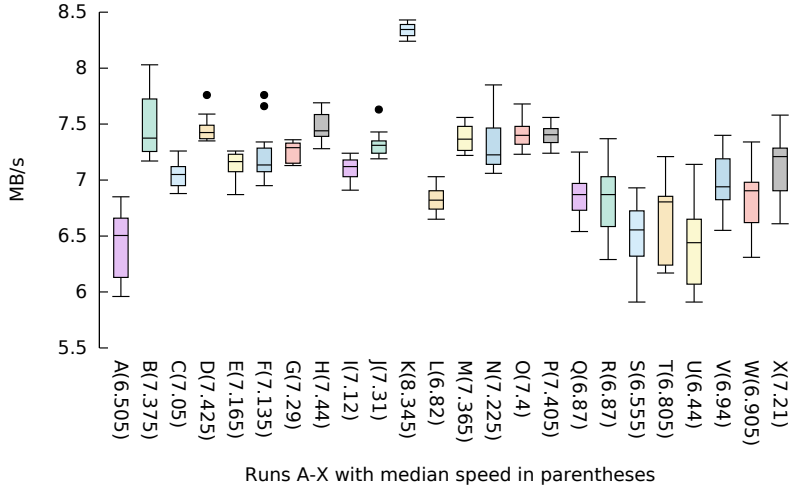


FIG. 5.12. *FIO sequential write 1MB.*

7. Acknowledgement. The research of the first author is supported by the enterprise partnership grant EPSPG/2012/480 from IRCSET and Intel Ireland Ltd.

The research of the second author is supported by the CloudLightning project, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 643946.

REFERENCES

[1] *axboe/fio GitHub*, `https://github.com/axboe/fio` (accessed 2015-12-14).
[2] *Barracuda ES.2 serial ATA*, `http://www.seagate.com/staticfiles/support/disc/manuals/NL35%20Series%20&%20BC%20ES%20Series/Barracuda%20ES.2%20Series/100468393f.pdf`(accessed 2015-02-11).
[3] *ceph/config_opts.h*, `https://github.com/ceph/ceph` (accessed 2015-01-09).
[4] *CinderSupportMatrix OpenStack*, `https://wiki.openstack.org/wiki/CinderSupportMatrix` (accessed 2014-02-12).
[5] *Dell PowerConnect 6200 series switches*, `http://www.dell.com/downloads/emea/products/pwcn/PowerConnect_6200_spec_sheet_new.pdf` (accessed 2015-03-18).
[6] *Dell storage SC series*, `http://www.dell.com/us/business/p/dell-compellent` (accessed 2015-05-25).

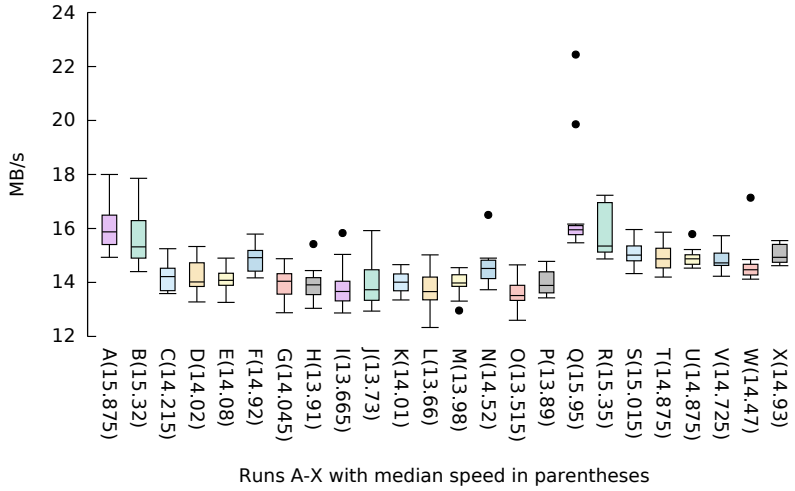


FIG. 5.13. *FIO random read 32MB.*

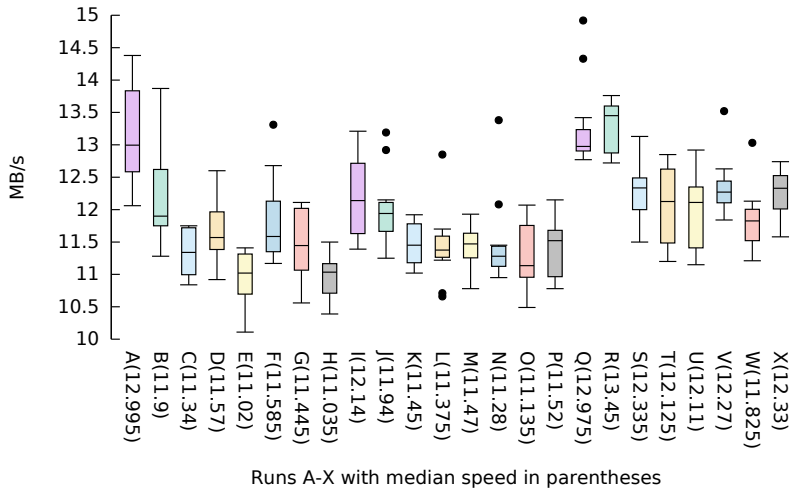


FIG. 5.14. *FIO random write 32MB.*

[7] *A guide to the OpenStack kilo release*, https://developer.ibm.com/opentech/2015/04/30/a-guide-to-the-openstack-kilo-release/ (accessed 2015-07-01).

[8] *HitachiUltrastar a7k1000*, http://www.hgst.com/tech/techlib.nsf/techdocs/DF2EF568E18716F5862572C20067A757/\$file/Ultrastar_A7K1000_final_DS.pdf (accessed 2015-02-11).

[9] *IEEE standard for local and metropolitan area networks–link aggregation*, pp. 1–163, http://dx.doi.org/10.1109/IEEESTD.2008.4668665 doi:10.1109/IEEESTD.2008.4668665.

[10] *OpenStack user survey insights: November 2014*, http://superuser.openstack.org/articles/openstack-user-survey-insights-november-2014 (accessed 2015-03-03).

[11] *Stackalytics | OpenStack community contribution in kilo release*, http://stackalytics.com/?release=kilo (accessed 2015-07-01).

[12] *WD RE4 series disti spec sheet - 2879-701338.pdf*, http://www.wdc.com/wdproducts/library/SpecSheet/ENG/2879-701338.pdf (accessed 2015-12-17).

[13] I. INKTANK STORAGE AND CONTRIBUTORS, *Pools ceph documentation*, http://docs.ceph.com/docs/master/rados/operations/pools/ (accessed 2015-07-07).

[14] S. MEYER AND J. P. MORRISON, *Supporting heterogeneous pools in a single ceph storage cluster*, in 2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 352–359, http://dx.doi.org/10.1109/SYNASC.2015.61 doi:10.1109/SYNASC.2015.61.

[15] S. PRATT AND D. A. HEGER, *Workload dependent performance evaluation of the linux 2.6 i/o schedulers*, in 2004 Linux

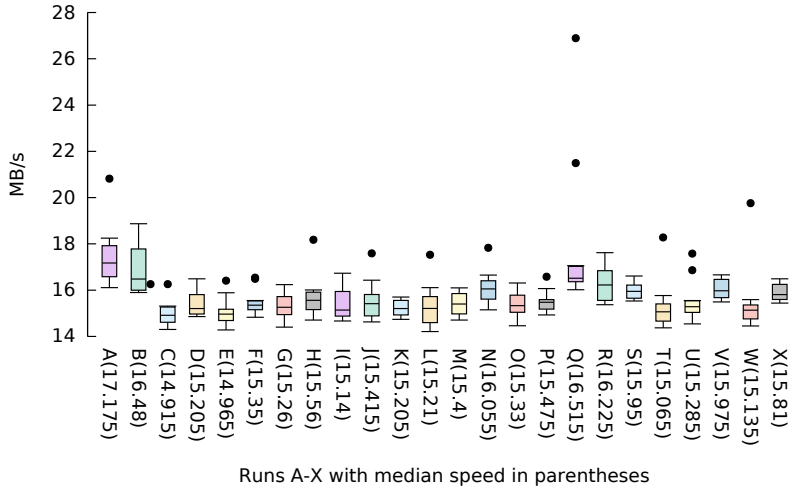


FIG. 5.15. *FIO sequential read 32MB.*

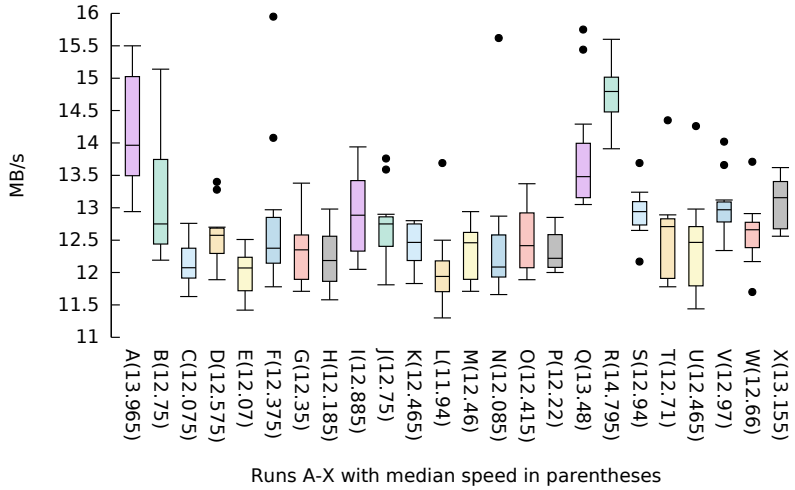


FIG. 5.16. *FIO sequential write 32MB.*

Symposium, http://landley.net/kdocs/mirror/ols2004v2.pdf#page=139 (accessed 2015-06-30).

[16] S. A. WEIL, *Ceph: Reliable, scalable, and high-performance distributed storage*, 2007, http://pdf-release.net/external/4047524/pdf-release-dot-net-weil-thesis.pdf (accessed 2014-03-26).

[17] S. A. WEIL, S. A. BRANDT, E. L. MILLER, D. D. LONG, AND C. MALTZAHN, *Ceph: A scalable, high-performance distributed file system*, in Proceedings of the 7th symposium on Operating systems design and implementation, USENIX Association, 2006, pp. 307–320, http://dl.acm.org/citation.cfm?id=1298485 (accessed 2014-03-26).

[18] S. A. WEIL, S. A. BRANDT, E. L. MILLER, AND C. MALTZAHN, *CRUSH: controlled, scalable, decentralized placement of replicated data*, in Proceedings of the 2006 ACM/IEEE conference on Supercomputing, ACM, 2006, p. 122, http://dl.acm.org/citation.cfm?id=1188582 (accessed 2014-03-26).

[19] S. A. WEIL, A. W. LEUNG, S. A. BRANDT, AND C. MALTZAHN, *Rados: a scalable, reliable storage service for petabyte-scale storage clusters*, in Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07, ACM, 2007, pp. 35–44, http://dl.acm.org/citation.cfm?id=1374606 (accessed 2014-03-26).

Edited by: Dana Petcu
Received: May 11, 2016
Accepted: July 17, 2016

DOI 10.12694/scpe.v17i4.1202
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 4, pp. 299–312. http://www.scpe.org

MULTI-OBJECTIVE MIDDLEWARE FOR DISTRIBUTED VMI REPOSITORIES IN FEDERATED CLOUD ENVIRONMENT

DRAGI KIMOVSKI [*], NISHANT SAURABH *, VLADO STANKOVSKI [†] AND RADU PRODAN *

Abstract. Virtualization represents an essential technology in Cloud computing, which allows virtual machines (VM) to be executed within their own environment on top of physical hardware. The modern methods for software delivery are utilizing the concept of Vitual Machine as a efficient tool for software packaging. Typically, VMs are created using specific templates that are stored in proprietary repositories, thus leading to provider lock-in and reduced portability in the cases of simultaneous usage of multiple federated Clouds. Unfortunately, the current state-of-the-art does not provide any efficient means for streamlined management of VM images across multiple repositories, especially within federated Cloud environments. In this paper we present a novel multi-objective middleware for distributed VMI repositories in federated Cloud environment. The middleware has been designed to provide easy to use interface capable of receiving unmodified and functionally complete VM images from its users, and transparently distribute them to a specific Cloud infrastructure in a federation with respect to their size, configuration, and geographical distribution, such that they are loaded, delivered, and executed faster and with improved QoS compared to their current behaviour.

Key words: Distributed Repositories, Cloud Federation, Multi-Objective optimization

AMS subject classifications. 68M14, 90C26

1. Introduction. Virtualization represents an essential technology in Cloud computing, which allows virtual machines (VM) to be executed within their own environment on top of physical hardware [14]. The modern methods for software delivery are utilizing the concept of Vitual Machine as a efficient tool for software packaging. The exploiting of this concept enables efficient scaling of applications by providing elastic on-demand provisioning of VMs in response to their variable load, thus increasing the utilization efficiency at a lower financial cost [4]. Typically, VMs are created using specific templates that are stored in proprietary repositories, thus leading to provider lock-in and reduced portability in the cases of simultaneous usage of multiple Clouds [8, 15, 10].

Unfortunately, the current state-of-the-art does not provide any efficient means for streamlined management of VM images (VMI) across multiple repositories within federated Cloud environments [cite cloud federation]. In such environments, the VMIs are currently stored by Cloud providers in proprietary centralised repositories without considering application characteristics and their runtime requirements, causing high deployment and instantiation overheads. Moreover, users are expected to manually manage the VMI storage, which is tedious, error-prone and time-consuming process, especially if working with multiple Cloud providers. Furthermore, each Cloud provider utilizes different type of storage technique and implements different interfaces for accessing it, thus reducing the usability even further. In this paper we present a novel Multi-objective middleware for management of VMIs in federated Cloud repositories, which has been developed as an essential part of the H2020 ENTICE project [1]. The middleware has been designed to provide easy to use interface capable of receiving unmodified and functionally complete VM images from its users, and transparently distribute them to a specific Cloud infrastructure in a federation with respect to their size, configuration, and geographical distribution, such that they are loaded, delivered, and executed faster and with improved QoS compared to their current behaviour. The proposed middleware has been focused towards overcoming the barriers that prevent the wide usage of distributed VMI repositories in federated Cloud environment and it aims to: (i) automatically distribute VM images based on multi-objective optimisation to meet application runtime requirements; and (ii) provide interface which enables users to manage their VM images in federated Cloud infrastructures without provider lock-in.

The paper is structured as follows. In Section 2 we present the basic concepts of the ENTICE environment. In Section 3 we introduce the ENTICE functional and non-functional requirements and how those affect the middleware functionality. Furthermore, a detailed overview of the middleware and all provided services is

[*]University of Innsbruck, dragi@dps.uibk.ac.at
[†]University of Ljubljana,
[1]http://www.entice-project.eu/

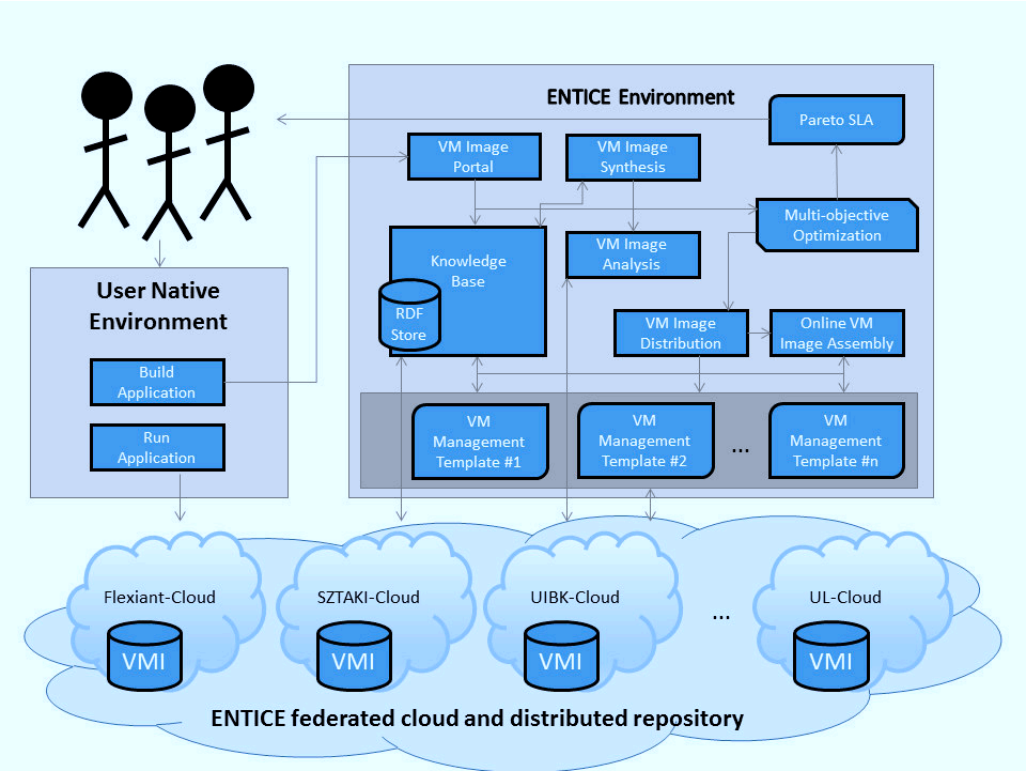


FIG. 2.1. *Top level view of the ENTICE environment*

presented in Section 4. Then we proceed by presenting the developed multi-objective framework for VMI redistribution. Further to this, we present details of the implementation of the framework and it's integration within the middleware. Section 7 summarizes the main results, their impact and future work.

2. ENTICE environment for efficient and transparent virtual machine operations in distributed Cloud repositories. The ENTICE project aims on development of an dynamic environment capable of receiving unmodified and functionally complete VM images from users, and transparently tailor and optimise them for specific Cloud infrastructures with respect to their size, configuration, and geographical distribution, such that they are loaded, delivered, and executed faster and with improved QoS compared to their current behaviour. ENTICE gradually stores information about the VMI and fragments in a knowledge base that is used for interoperability, integration, reasoning and optimisation purposes.

VM images management is supported by ENTICE at an abstract level, independent of the middleware technology supported by the underlying Cloud computing infrastructure. To further shield the users from the complexity of underlying Cloud technologies and simplify the development and the execution of complex use cases, ENTICE focuses on providing the flexibility for tailoring the VM images to specific Cloud infrastructures.

The ENTICE repository includes, among other features, techniques to optimise the size of VM images while maintaining their functionality, automatically share images (or parts of the images) among repositories (even in multiple administrative domains or cloud infrastructures), and optimally deploy them in response to application and data centre requirements.

The ENTICE environment, depicted in Figure 2.1, aims to prove a universal backbone and middleware for IaaS VM management operations, which accommodate the needs for different use cases with dynamic resource and other QoS requirements. The ENTICE technology is completely decoupled from the applications and their specific runtime environments, but continuously supports them through optimised VM image creation, assembly, migration and storage.

Within the ENTICE environment, the Multi-objective middleware for distributed VMI repositories has

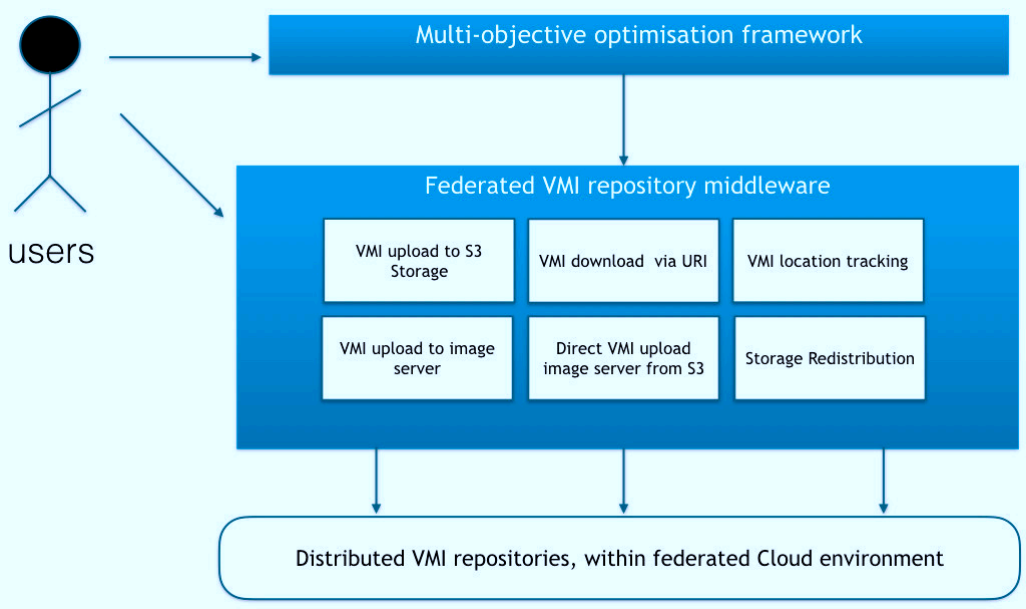


FIG. 2.2. *Top level view of the Multi-objective middleware for distributed VMI repositories*

been layered in multiple modules. To be more concrete, the middleware is composed of VM image distribution module, online VM image assembly module and multi-objective optimization framework. All components interact by using service based API interfaces, thus allowing independent integration of the middleware in various distributied VMI repository architectures. The top level view of the middleware, as an independed fully integrated module, is presented in Figure 2.2.

3. Functional requirements. As discussed in Section 1, in order to overcome the barriers that limit the possibilities for Cloud federation, we present a specific use-case scenario, which is highly relevant for proper definition of the middleware interface that ENTICE environment has to offer. Furthermore, based on these requirements, we list down the imminent functionalities specific to relevant components of the ENTICE environment.

The use-case of ENTICE environment at the outset, typically initiates with the Application Provider uploading VM Image encompassed with appropriate application functionalities. Henceforth, an easy interface to the user which integrates to the funtionalities including image optimization, storage and the knowledge oriented image portal with enhanced reasoning based module for easy access of services (example: upload, download, update, optimize etc.) and VM image management is required. In general, user specific images are large sized comprising of redundant functionalities incurring high storage cost resulting to image distribution and instantiation overheads. Henceforth, the splitting of an image into multiple fragments serving common functionality and storing them only once is necessitated to reduce the redundancy. Furthermore, the repository optimization with regards to the storage location of images evaluating performance metrics of each attached repository medium is imminent for faster VMI distribution across multiple clouds. Based on these factors, we identify the generalized requirements for the ENTICE environment. Hence, we divide the ENTICE design requirements in the following broad categories: (i) Application data-related requirements, (ii) Security-related requirements, (iii) VMI and fragment management-related, (iv) Information and metadata management-related requirements, and (v) QoS metrics-related.

The Application-data related requirements encompass the delivery of various content alongside the VM image, for the efficient execution of the corresponding application. To this extent, the ENTICE environment provides a functional descriptions tool that contains references to the additional VM image content and evaluate its presence in terms of files, folders, sizes, distribution and etc.

Furthermore, the ENTICE environment needs to address various security measures, to prevent unauthorized

access of VMIs. For this purpose, the repository must support content delivery through encrypted channels. Moreover, the analysis methods for VMIs of the ENTICE environment should not compromise the privacy and security of the Cloud application owners and should not expose any user data, which should be securely stored in the knowledge base.

One of the requirements that ENTICE repository holds is to provide fast VM image delivery, hence identify the VMIs and fragment management-related requirements. The management module requires efficient integration of specific modules for VM image and meta-data delivery, with the a Multi-objective optimization framework for online and offline distribution of the data. In addition, the ENTICE environment needs emphasis to support the delivery of data in the form of files or folders containing user specifications, thus requiring relevant informations and metadata management. In such cases, ENTICE must provide appropriate storage and data delivery mechanism together with the VMIs. Further, the data needs to be indexed, so that the appropriate distribution techniques allow efficient delivery to relevant geographical locations. Hence, it is required to miantain index hashing of VMIs and correspondin fragments. The semantic model represented in the Knowledge Base requires to store information about the VMIs and their functionality, the geographic location, the URI, and other details for the search facility.

Lastly, monitoring of QoS metrics is necessary to be able to observe the time needed to move VMIs and/or fragments among repository locations and to deploy a VMI needed by a particular Cloud application. A further requirement is to monitor the infrastructure on which the ENTICE repository is deployed to ensure that the scaling and speed of operation is adequate.

Hence, based on the discussed requirements, we list down the important use-case functionalities for the Multi-objective VMI repository as a federation middleware:

- *Upload of unoptimised images to S3 storage*
- *Upload of optimised images to S3 storage*
- *VM Image download via URI*
- *VM Image location tracking*
- *Upload of VM Image directly to Image Server of Cloud provider for deployment*
- *Upload of VM Image to the image server of specific Cloud provider from S3*
- *Storage redistribution of VM Images stored within S3 based VMI repository*

Furthermore, for the optimization framework we have identified and implemented the following use-case functions:

- *Applying Multi-objective Optimization for distribution of the VM Images and fragments across distributed storage sites*
- *Optimized extraction of fragments, based on functionality, for assembly of the VM Images during the deployment stage*
- *Movement of VM image from one storage location to another*
- *Location Tracking of each VM Image to enhance storage and distribution*
- *VMI conversion from one provider specific template to another to achieve interoperability over multiple cloud*

4. VMI repository as a federation middleware. The *ENTICE* VMI storage repository evolves from the typical storage systems[13] providing general storage services, and instead focuses to act as a middleware corresponding to specific Virtual Machine operations regading the storage, deployment and interoperability of VMI over multiple clouds. These middleware services enhances flexibility of user with respect to maintaining VMI accomplished with user-specific funtionalities and applications and hence surpasses the manual error-prone VM oeprations performed by the users in a federated cloud model. The *ENTICE* federation middleware is attached to storage systems of varying cloud providers(currently supports S3 object storage) accompanied with the Multi-objective optimization service, providing enhanced availability and hence optimizing the VMI distribution time with enhanced deployment.

In general, individual cloud providers constitute of centralized image repositories with services specific to their environment. *ENTICE* recognizes the limitations including the interoperability of VMI and and hence intializes a middleware API with a number of services corresponding to storage of VMI across federated multiple clouds[11] with enhanced Quality of service[2] with regard to VM provisioning and deployment. This enables

the user to have flexible usage of Cloud Infrastructure as a Service as a global paradigm[9]. Henceforth, in this section, we identify the following middleware services provided by *ENTICE* satisfying the use case functionality defined in the previous section and explain the requirements and service it holds.

4.1. Upload of Un-optimized VMI to S3 Storage. This functionality services the storage of unoptimized images to *ENTICE* S3 repository. The un-optimized images basically refer to the initial version of user-specific images without any optimizations performed with respect to size and application functionality it provides. This image version is often termed as master copy consisting of all the user built applications. The *ENTICE* middleware API intiates the storage of such images onto the corresponding S3 repository with return values namely, VMI name, identifier, S3 storage repository location and Unique resource identifier(URI) of the VMI. The return values are transmitted onto the *ENTICE* knowledge base for future reference by the user.

4.2. Upload of Optimized VMI to S3 Storage. The stored master copy of VMI onto S3 often consists of unused libraries and functionalities. Henceforth, any optimization performed over the VMI by the user by pruning the images, reduces the size of image allowing enhanced provisioning and faster deployment. The upload request of such optimized image is serviced by uploading the latest optimized version along side the master copy of the image. However, the master copy of the image is not deprecated from S3 repository with a view to safeguard the requirement of the user to restore the initial version consisting of complete user specific applications at any point of time. The API return field values namely, name of VMI master copy, name of opimized VMI, storage location, optimized image version and its corresponding URI. These field values allows to maintain the versioning tree for image and its optimizations.

4.3. Upload of VMI to Image Store directly. In general, cloud providers have centralized image store which allows the upload and registry of images suitable to the corresponding cloud infrastructure. The image store is basically attached to varying storage sites differing from one cloud provider to another. For example, Amazon enables the bridging of the S3 storage with the image store, allowing upload of registry of VMI to S3 following a set of rules separating from other content related files and data. The *ENTICE* middleware provides the flexibility to the user to upload their image directly to the image store of their favaorable cloud provider region. As discussed earlier, the upload functionality to S3 is used in the case where decision improbability surfaces within the user about the cloud provider and the region, where VMI has to be deployed. In this functionality, user can choose the image store of the specific cloud provider and corresponding region. The return values of this service provide field values with respect to VMI name, VMI unique identifer, size of the image onto the image store, cloud provider identifier and the corresponding zone identifier.

4.4. Upload of VMI to image store from S3 storage. The *ENTICE* API providing the service enables the transfer of un-optimized or optimized images from S3 repository to the image store of the chosen cloud provider, where it can be registered as an instance and hence deployed. As dicussed earlier, S3 storage is used as a back-up to enhance availability of images specifically in the case of indecision related to the cloud provider and region for the deployment of virtual machine(VM). Once the user is decided over the region in which VM has to be deployed, the API service distributes the image to the corresponding destined image store. Henceforth, the field values namely, VMI name, VMI unique identifer, size of the image onto the image store, cloud provider identifier and the corresponding zone identifier are returned and stored in the knowledge base of future reference.

4.5. VMI Location Tracking. The API service tracks the location of stored and instantiated VMI for every user to return the filed values namely, cloud provider identifier and zones where a specific user usually instantiates or store the image. These image specific details allow the Multi-objective optimization module as discussed in next section, to analyse the popularity of cloud provider and zones for varrying users.

4.6. Redistribution of VMI over storage sites. This service corresponds to the Multi-objective optimization service and is performed internally without the interference of the user. The decision making module provides the input parameters for redistribution of the stored images onto the various repository sites corresponding to the user related Quality of service metrics. This services is likely to enhance the distribution of VMI onto the cloud provider, further improving the deployment time and elatic auto-scaling of VM.

4.7. Download via URI. As defined earlier, each stored VMI onto S3 repository returns a URI allowing users to download images from the *ENTICE* environment to their private virtualized cloud infrastructure. Hence, in such case the image is stored at a repository site closer to user location enabling faster download.

4.8. VMI Interoperability. The varying Cloud providers allow specific image formats to be instantiated as VM over their corresponding infrastruture. For example, Amazon require specific Amazon Machine Images(AMI). Hence, this hinders the federated Infrastruture as a Service model[11] leading to vendor lock-in[12] and promotes the manual conversion of images to suit the respected cloud provider to make the process more error-prone. This unique funtionality of *ENTICE* enables interoperability of any stored images onto S3 over multiple clouds by converting to the image format as per the suitability of the user defined cloud infrastructure irrespective of the user provided image format.

5. Multi-objective Optimization Framework for VM image re-distribution. In this section a detailed description of the multi-objective optimization framework for VMI distribution in Federated Cloud repositories will be presented. The optimization framework is capable of directly interacting with the middleware module and has been applied on two distinctive application levels: (*i*) initial VMI distribution, (*ii*) offline VM image redistribution and (*iii*) online VM image redistribution.

5.1. Framework description. The framework is encompassed around unified multi-objective optimization module, which can be utilized for multiple different optimization purposes. Internally, the optimization module is branched in two distinctive sub-modules. Each of the sub-modules has been tailored specifically for a given task. The Initial Distribution sub-module covers the multi-criteria evaluation of the possible repository sites where the VMIs or associated data sets can be initially stored. Afterwards, the Offline VMI Redistribution sub-module encapsulates the optimization of the VM images distribution within the federated repository sites. By taking into account the VMIs usage patterns, the algorithm is capable of providing multiple trade-off solutions, where each solution represents a possible mapping between the stored images and available repository sites. The Online VM image Redistribution sub-module aims on dynamical redistribution of particular VM images/fragments during particular application execution. This sub-module is only applied for users specific VM images and in accordance with the technicalities of the application that is being executed. The framework is dependent on the repositorys usage patterns to properly optimize the distribution of the VM images. To this aim a specific module is required to store information on the previous transfers within the federation and to provide the collected data in a proper format. The module has been realized as an ontology-based knowledge base [1]. The framework has been designed to acquire input data from the knowledge base, and also to return the output results there. Moreover, a specific monitoring agent is required for proper documentation of the data transfers. The monitoring tool itself can be realized in multiple different manners, and it is dependent on the specifics of the Cloud infrastructure.

Furthermore, the framework provides a service based API, through which the Decision Maker (DM) can access the list of optimal Pareto solutions in a guided manner, thus reducing the complexity of the VMI storage management process.The high level structure of the optimization framework is presented on Figure 5.1.

5.2. Initial VM image upload. It is of paramount importance to properly store new VMIs and related data sets in federated Cloud repositories. In this section we introduce concepts from the field of Multiple-criteria decision making, to assist image providers and users to efficiently store new VMIs in accordance with their needs and repository characteristics [3]. The described module, provides a tool which mitigates the process of initial VMI upload, when the available storage sites possibilities are so large that can overwhelm the user during the decision process.

The problem of initial VM image upload consist of a finite number of combinatorial alternatives, which are explicitly known in the beginning of the solving process. In this case, each alternative solution represents one storage site in the federated repository, where the image or data-sets can be stored. Every solution is evaluated on the basis of two conflicting objectives. For the specific problem, the following objectives have been defined:

$$f(P) = B_r \tag{5.1}$$

$$f(C) = C_{st} + C_{tr} \tag{5.2}$$

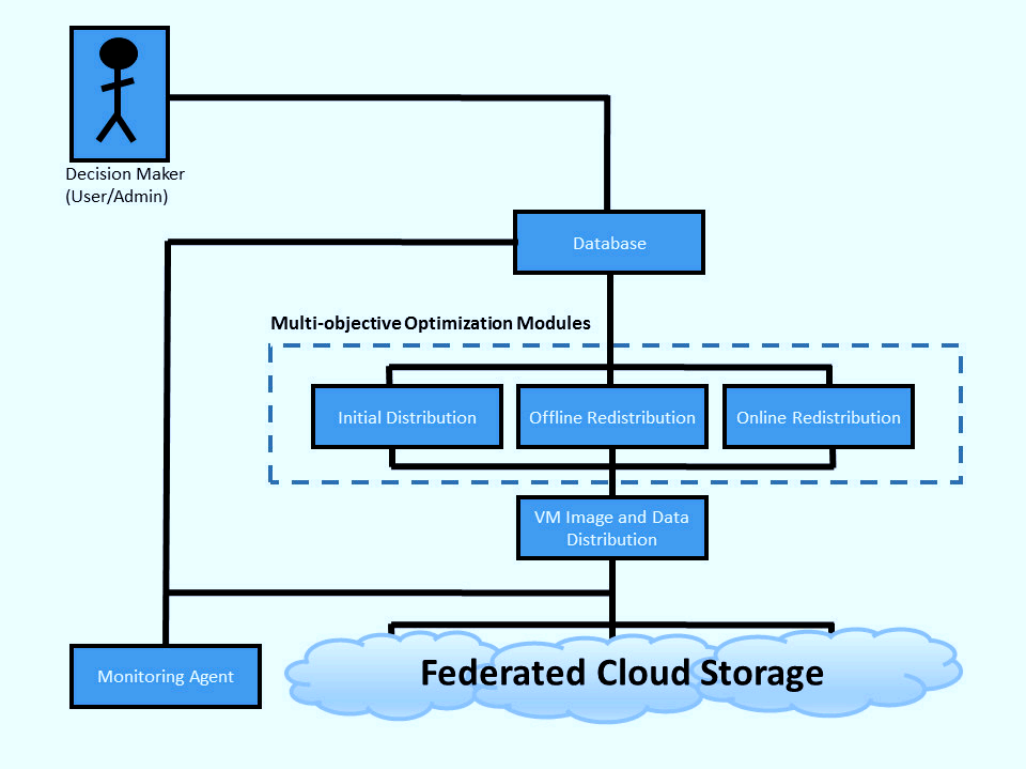


FIG. 5.1. *Top level view of the Multi-objective Optimization Framework for VM Image distribution*

where B_r represents the maximal theoretical performance of the interconnections of the repository, while C_{st} is the cost for storing data on the given repository and C_{tr} is the cost for transfer. Based on the given objectives, all possible storage sites in the repository, are then evaluated. It is important to be noted, that the evaluation is performed only on the feasible solutions, i.e only on the list of available repository sites. This means that prior to evaluation, all constraints for storing the VMI are taken into account. Afterwards, by introducing the concept of domination all evaluated solutions are sorted. The solutions which are non-dominated by any other solution are presented to the user in the form of Pareto front. In a sense, those solutions represent multiple optimal storage sites for storing a single VM image within the federated repository. Next, the user, as a decision maker, can choose where to initially store its own images.

It also worth mentioning, that due to the static nature, this type of evaluation should only be performed when new storage sites have been added or removed from the federated repository. Afterwards, if there are no changes in the structure of the federated repository, the evaluation data can be used for selecting the appropriate storage site for every VM image that might be uploaded in future.

5.3. Offline VM image redistribution. Unlike the initial image upload, the problem of offline VMI redistribution consist of a finite, but very large, number of combinatorial alternatives, which are not known in the beginning of the solving process. The optimization process is conducted by utilizing two conflicting objectives: cost for storing and transferring of the data, which we simply call Cost objective and Performance objective. This process is performed by analyzing the repositories usage patterns, and results in optimized distribution of the VMIs and the associated data-sets across the federated environment. In what follows the exact sequence of steps of the offline VMI redistribution sub-module is presented.

5.3.1. Objective functions modeling. The cost model is described around the notion of the financial expenses which are needed to store a unit of data in a given repository site C_{st} and the economical burden for transferring the data from the initial to the optimal site C_{trnew}. The exact values of the financial expenses for

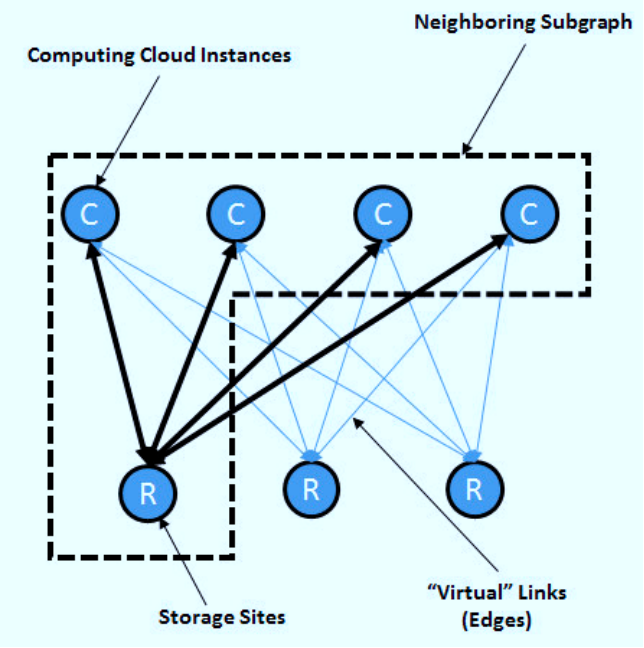


FIG. 5.2. *An example of a neighbouring sub-graph in a structure with 3 repository sites and 4 different cloud providers*

data storage and transfers should be provisioned by all Cloud providers within the federation. For each VM image the cost objective can be calculated by using the formula below:

$$f(C) = C_{st} + C_{trnew} \tag{5.3}$$

The performance model includes much more complex reasoning behind it. It is based on the VM image usage patterns and it requires proper monitoring tool for efficient execution. The raw theoretical throughput of the interconnecting structure within a Cloud federation does not properly describe the factual communication performance, as it is difficult to predict the actual route the packets may take to reach the destination and the load on the intermediate communication channels. Opportunely, it is possible to leverage the data from the frameworks monitoring module to perform a coarse but sufficient estimation on the actual throughput between any pair of end points in the federation. In this way, if there is a sufficient information on the previous transfers among the repository sites and the Cloud computing instances, a direct virtual links between the above mentioned entities can be abstracted over the physical network and their bandwidth can be estimated.

Furthermore, it is possible to model an undirected weighted graph, where the vertices correspond to either a repository site or a computational Cloud instance and the edges of the graph are represented by the virtual links. The weighted graph actually enclosed a union of multiple neighboring subgraphs, where each storage site vertex, as direct neighbor, is linked to all known computational cloud vertices. The weights of the edges in the graph are determined by leveraging the estimated average bandwidth B_{rc_i} on the corresponding virtual links. The weights are calculated dynamically, based the VMI distribution that is being considered. To properly model the weight of the edges, we introduce weight function, which considers the total number of downloads of the VMI to all neighbours G_{tv} and the number of downloads to particular Cloud neighbor G_i. The ratio of those two values is then multiplied with the estimated bandwidth of the particular virtual link to provide the final value of the edges weight. The structure of the neighbouring sub-graph has been represented on Figure 5.2.

Subsequently, for modeling of the performance objective, the sum of the weights of the edges in the neighbouring subgraph is exploited, thus the performance can be described as:

$$f(P) = \sum_{i=1}^{n} B_{rc_i}(\frac{G_i}{G_{tv}}) \tag{5.4}$$

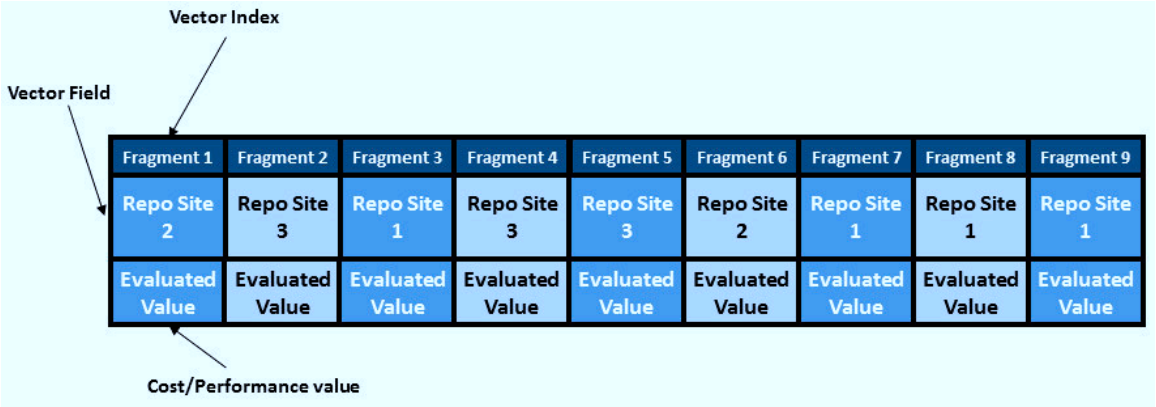


FIG. 5.3. *An example individual represented as a solution vector*

5.3.2. Search Algorithm and Decision Making. The core of the offline VMI redistribution sub-module is constructed over the NSGA-II multi-objective optimization algorithm [5]. As with any population based genetic heuristic the basic entity is the individual. Within the given problem description the individual has been represented as vector with a size equal to the number of stored VMIs. The value kept in every element of the vector corresponds to a single storage repository where a particular VMI can be stored. For accomplishing the above statement, within the proposed framework, each VMI is assigned with a unique ID value, which correspond to the index of the vector element. Respectively, all storage sites in the federation are also assigned with unique IDs that are parallel to the appropriate values saved in the vector elements. In such way, each individual corresponds to a solution vector that represents unique global mapping of all VMIs to storage sites in the federated repository.

Afterwards, multiple solutions vectors are created and then randomly populated with values in the range from one to the number of available storage sites, thus creating the initial population. Every single individual represents one possible distribution solution that has to be evaluated. Then, the evaluation of each individual is performed by reading the values stored in the vector fields. Based on those values, starting from every element in the vector, a neighboring subgraph is constructed and the appropriate objective functions are applied. Those values are then grouped together and the median value is selected as the overall fitness of the given individual. An example of a single individual that correspond to a solution vector for mapping 9 VMIs to 3 storage repository sites in a given federation is presented on Figure 5.3. When all individuals in the initial population have been successfully evaluated, the proper mutation and crossover operators are applied to create the children population. Then, the parents and children populations are grouped together and sorted according to dominance. Afterwards, only the best solution of the newly formed group are selected for the next iteration. This process is then repeated for a predefined number of iterations. The solutions which have been acquired after the last iteration are sorted based on the dominance. The non-dominated solutions are then presented to the administrative entity of the federation, which acts as a DM, and should select the most appropriate solution based on the pre-defined decision making policy.

Decision making on the alternatives discovered by the optimization algorithm requires an explicit model of the decision maker preferences. For the case of offline VMI redistribution the DM model will depend on the implementation of the federated infrastructure. As the offline image redistribution envelops federation wide distribution of the VMIs we envision that the DM will be an administration entity, which will implement the federation storage policy based on the decision making model. Based on the decision done by the DM entity the storage redistribution sub-module would be capable to transparently moving all affected VMIs in the more optimal position.

5.4. Online VMI redistribution. One very important aspect that should be considered in federated cloud environment and repositories is the optimization of specific users VM images and corresponding data sets while correlated applications are being executed. Even though the offline VM image redistribution should place

the VM images in the optimal storage site, there might be cases where the optimization is required only locally, for some particular images or data sets. For example, if a user continuously deploys particular VM image within a short period of time, the position where that image is stored can be additionally optimized based on the newly available data. Consequently, the image can temporarily be transferred to the more optimal solution for the given scenario. The same principle can be applied to the associated datasets, which can be redistributed closer to the physical machines where the VM images are deployed.

By using the same methods implemented in the offline VM image redistribution, the online VM image provisioning can be managed. As both processes are analogous, the only difference comes from the scope and the time interval in which the optimization is performed. With the online VM image redistribution, the optimization is only executed by users request, and only on its own images. When the user ask for optimization of the VM images storage position while deployment, the algorithm is initiated with a limited scope. The input data of the optimization module is only narrowed to the user images and the optimization only takes into account the users usage patterns in a previously set time interval. In this way it becomes possible to further optimize the position of VM images in the cases when they are frequently deployed in a short intervals of time. In a sense it can be commenced, that the offline VM image storage optimization is coarse grained and infrequently applied on all VM images and datasets stored in the federated repository. Contrary, the online VM image optimization is fine grained, and it involves only a users specific VM images that are frequently deployed in a given time interval.

From an algorithmic point of view, the cost and performance objectives are modeled in the same way. The only difference are the input parameters, which in the case of online VM image optimization only involves few VM images, and the output solution vector is also limited only to the images that need to be transferred to more optimal place. As this process only involves fraction of the images stored in the federated repository it can be performed more frequently with reduced computational penalty.

6. Evaluation. In this section, the efficiency of the Multi-objective middleware has been experimentally evaluated based on a synthetic set of benchmark data. As our research deals with the implementation of a combinatorial multi-objective problem in federated Cloud environment, we present an experimental results that demonstrate the ability of our approach to provide an adequate VMI distribution across federated repositories.

With respect to the different application levels of the middleware and the multi-objective optimization framework overall, distinctive set of experiments were conducted. The initial VMI upload module has been evaluated on the basis of the degree of scalability, while the behaviour of the redistribution module has been examined from multiple aspects, such as accuracy, scalability and computational performance.

The simulation experiments on the middleware and Multi-objective optimization framework were conducted on a standalone Linux based machine with an Intel core i7-3770K processor (4 cores and 2 threads per core) working at 3.5 GHz with a 16 GB of RAM. Furthermore, the functionality and behavior of the middleware ware tested in a distributed VMI repository, composed of three storage sites located in Austria, Slovenia and Hungary. In order to guarantee efficient integration and platform independency, all software modules were developed in Java. For the purpose of multi-objective optimization we have leveraged the jMetal optimization library [6].

To begin with, the scalability and computational performance of the initial VMI upload module have been evaluated by varying the number of repository sites in the federation from 10 up to 10000 sites. Figure 6.1 shows the correlation between the average execution time and the number of storage sites in the federation. It is evident that the module can be lightly scaled up to large sizes. For relatively small federations the module can be invoked at each VMI upload, as it requires only few milliseconds to be executed.

On the other hand, the VMI redistribution module encloses diverse operations that can affect its behavior to a various degree. Due to the nature of the algorithm it is not adequate to evaluate it's computational performance based on the number of repositories in the federation. Increasing the number of storage sites, influences on the number of possibilities where to store a single VMI image, which translates into reduced quality of the proposed solutions, but relatively constant execution time. For example, on Figure 6.2 a scenario in which the vector size (number of fragments) and number of evaluations have been kept constant, while the number of available repositories has been increased from 10(blue) to 100 (red), is presented. The Pareto fronts from both executions have been plotted together to show the difference in quality of the final solutions. The

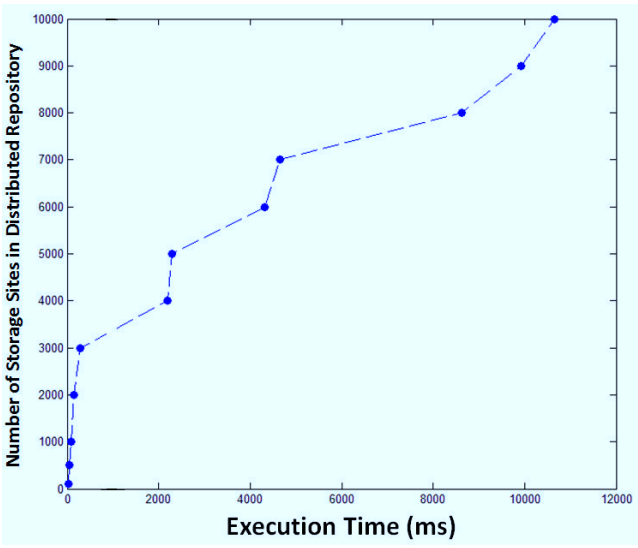


FIG. 6.1. *Correlation between the execution time and the number of storage sites in the case of initial distribution*

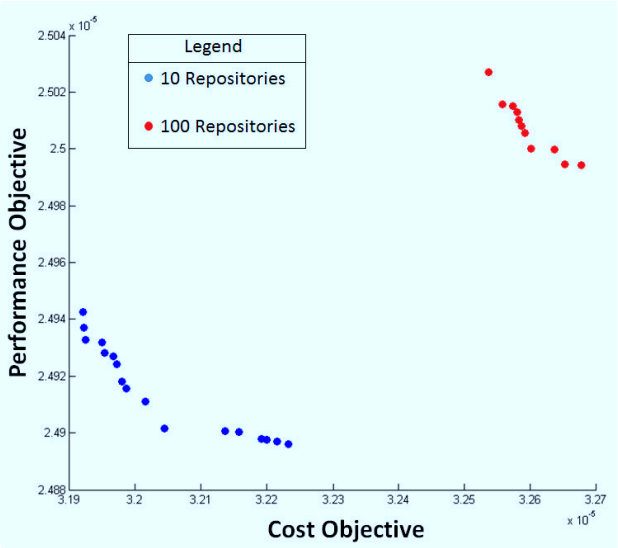


FIG. 6.2. *Comparison of two Pareto fronts during redistribution with varying storage sites*

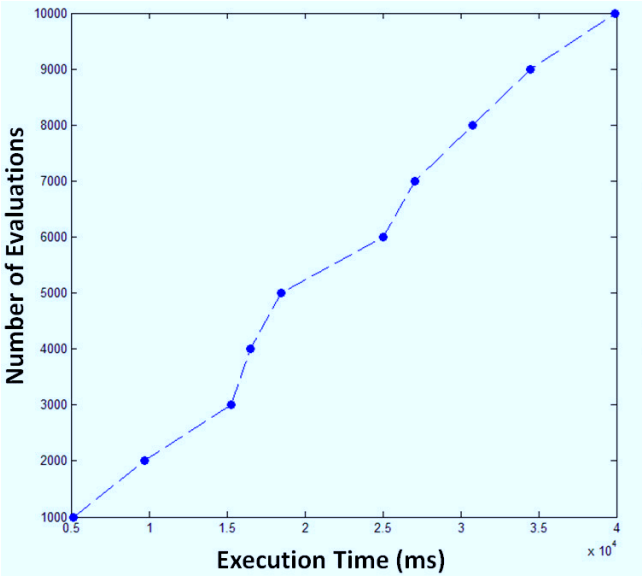


FIG. 6.3. *Influence of the number of evaluations over the execution time during offline redistribution*

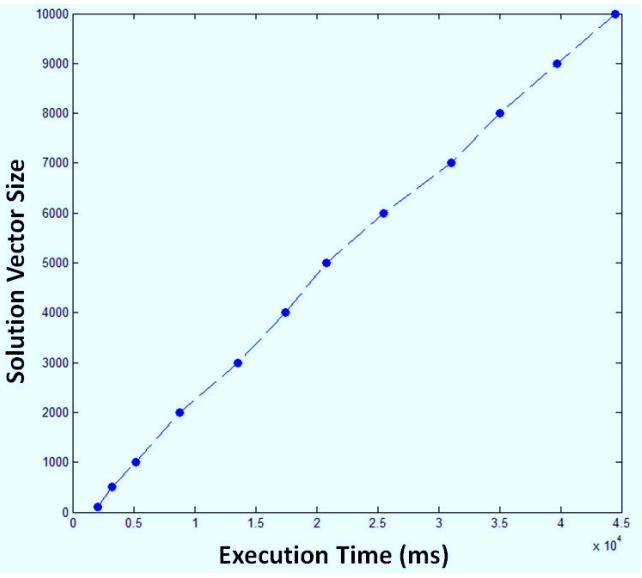


FIG. 6.4. *Influence of the solution vector size over the execution time during offline redistribution*

experimental scenario clearly shows that if we increase the number of storage sites, while maintaining constant number of evaluations, the quality of the solutions will decrease.

Furthermore, on Figure 6.3 and Figure 6.4, respectively, the influence that the number of evaluations and the size of the solution vector have on the computational performance is presented. In both cases, the number of associated cloud computing instances and storage sites were maintained constant; only the corresponding parameters were increased gradually. The presented results support the assumption of satisfactory scalability, both in a sense of increased number of stored VMIs and number of iterations needed to provide mapping solutions with good quality.

Moreover, on Figure 6.5, the influence of the number of known computational Cloud instances over the

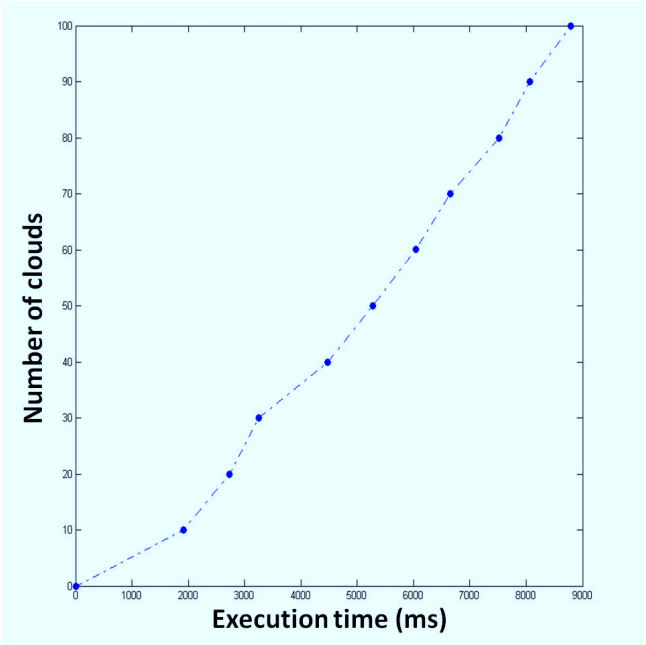


FIG. 6.5. *Influence of the number of known computational Cloud instances over the computational performance of the optimization framework during offline redistribution*

computational performance of the optimization framework is presented. During execution, the number of known computational Clouds was linearly increased, thus inducing higher execution times. The evaluation results are clearly supporting the assumption that the proposed framework scales-up very well with the increment of the problem complexity.

Lastly, Figures 6.6 and 6.7 provide a comparison of the quality values for the conflicting objectives considered in this work. The values for the cost objective have been calculated based on the publicly provided price list for storing data in the Cloud by Amazon. The performance objective has been modelled based on the reported communication performance measures for 10Gbit and 1Gbit Ethernet [7]. For readability reasons, the bandwidth values, were converted to delivery time needed for 1Mbit of data to be transferred from the source to the destination.

With respect to the parameters of the evolutionary algorithms, we have used a population of 1000 individuals, that iterates from 1 to 100 generations across populations. Every single individual(solution vector) is comprised of 1000 chromosomes, thus inducing mapping solutions for 1000 VMIs. Taking into account the results obtained in preliminary experiments, we have used simulated single point crossover with a crossover probability of 0.9, a mutation probability equal to 1/n (n is the number of decision variables). The results indicate very high efficiency of the redistribution module, as it can provide better quality mapping solutions, especially in regards with the performance objective. It can be concluded that in the cases where public Cloud providers, such as Amazon, are being used, it is possible to decrease the delivery time by 143%, while maintaining lower price for storing the VM images.

7. Conclusion. In this paper a novel Multi-objective middleware for management of VMIs across distributed repositories in federated Cloud environment has been proposed. The research work has resulted in development of a optimization framework that exploits multiple different factors, such as communication performance requirements, VMI use patterns, and structure of images, in order to optimize the distribution and placement of VMI across distributed repositories and to significantly lower their provisioning time for complex resource requests and for executing the user applications. Furthermore, streamlined interface that implements a wide array of behavioural functions beyond the typical storage has been provided. The middleware interface offers complex functionalists, such as:

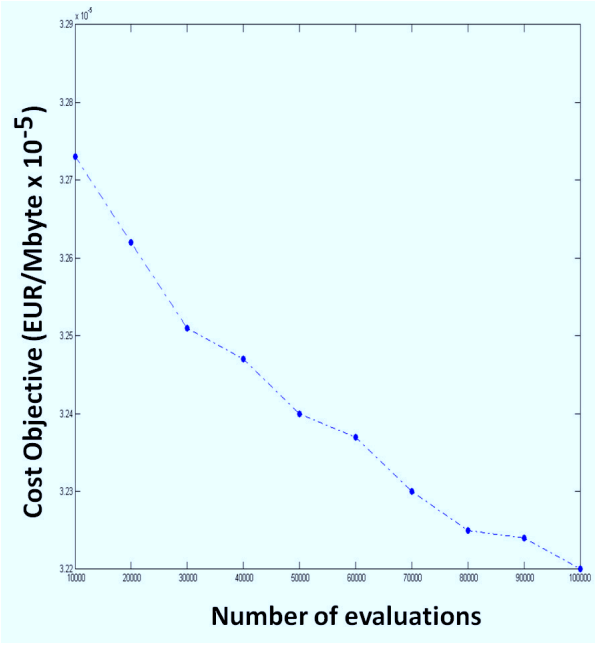


FIG. 6.6. *Average quality values of the Cost objective in comparison with the total number of evaluations*

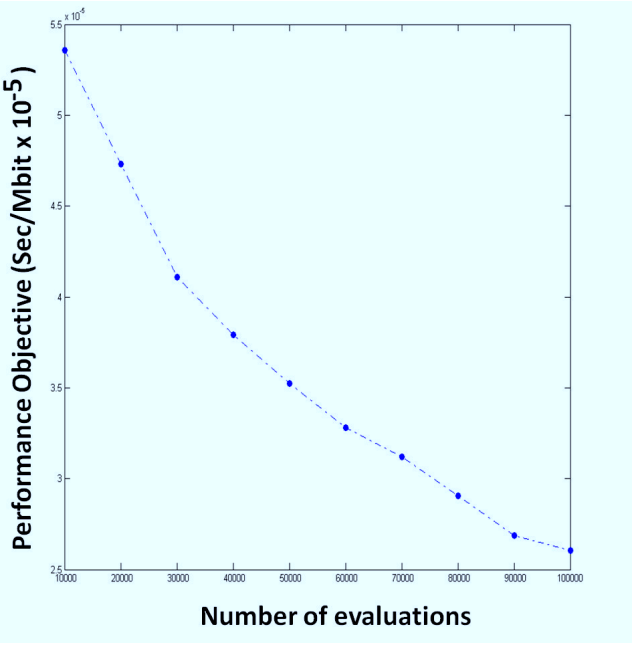


FIG. 6.7. *Average quality values of the Performance objective in comparison with the total number of evaluations*

- *Upload of unoptimised images to S3 storage*
- *Upload of optimised images to S3 storage*
- *VM Image download via URI*
- *VM Image location tracking*
- *Upload of VM Image directly to Image Server of Cloud provider for deployment*
- *Upload of VM Image to the image server of specific Cloud provider from S3*
- *Storage redistribution of VM Images stored within S3 based VMI repository*

The optimization framework, and the middleware overall, have been evaluated based on synthetic simulation benchmark. As our research deals with the implementation of a combinatorial multi-objective problem, where the main incentive is to find the proper mapping of VMIs across storage sites, we present an experimental results that demonstrate the ability of our approach to provide an adequate VMI distribution across federated repositories.

This research area still poses multitude new challenges that urge additional efforts for the accomplishment of the ultimate goal, effortless management of VM images and associated meta-data in federated Cloud environments. In near future, we plan to introduce more complex reasoning and decision making mechanisms, to further improve the multi-objective optimization framework with support for automated creation of VM images in multiple different formats by utilizing virtual machine management templates, and to provide more efficient user interface.

Acknowledgments. This work is being accomplished as a part of project *ENTICE: "dEcentralised repositories for traNsparent and efficienT vIrtual maChine opErations"*, funded by the European Unions Horizon 2020 research and innovation programme under grant agreement No 644179.

REFERENCES

[1] S. ABBURU. *A Survey on Ontology Reasoners and Comparison.* International Journal of Computer Applications (0975 8887), Volume 57 No.17, November 2012.

[2] A. K. Bardsiri and S. M. Hashemi. *Qos metrics for cloud computing services evaluation.* International Journal of Intelligent Systems and Applications (IJISA), 2014.
[3] J. Branke et al., eds. *Multiobjective optimization: interactive and evolutionary approaches.* Vol. 5252. Springer, 2008.
[4] P. C. Brebner, *"Is your cloud elastic enough?: Performance modelling the elasticity of infrastructure as a service (iaas) cloud applications".* In Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE), 2012, ACM, 263266.
[5] K. Deb, A. Pratap, S. Agarwal and T. A. M. T. Meyarivan. *A fast and elitist multiobjective genetic algorithm: NSGA-II.* IEEE Transactions on Evolutionary Computation 6(2), 2002, 182-197.
[6] J.J. Durillo and A. J. Nebro. *jMetal: A Java framework for multi-objective optimization.* Advances in Engineering Software 42(10), 2011, 760-771.
[7] W. C. Feng, P. Balaji, C. Baron, L.N. Bhuyan, and D.K. Panda. *Performance characterization of a 10-Gigabit Ethernet TOE.* In 13th Symposium on High Performance Interconnects, IEEE, 2005, 58-63.
[8] I. Goiri, J. Guitart and J. Torres. *Characterizing cloud federation for enhancing providers' profit.* In 2010 IEEE 3rd International Conference on Cloud Computing (CLOUD), 2010, 123-130.
[9] A. Iosup, R. Prodan and D. Epema. *IaaS Cloud Benchmarking: Approaches, Challenges, and Experience.* In Proc. of 5th Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS), 2012.
[10] K. Kaya, Kamer. *Heuristics for scheduling file-sharing tasks on heterogeneous systems with distributed repositories.* Journal of Parallel and Distributed Computing 67 (3), 2007, 271-285.
[11] G. Kecskemeti, A. Kertesz and Z. Nemeth. *Developing Interoperable and Federated Cloud Architecture.* IGI Global, 2016, 1-398.
[12] J. Opara-Martins, R. Sahandi and F. Tian. *Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective.* J. Cloud Comput. 5 (1), 2016, Article 54.
[13] M. Placek and R. Buyya, *A Taxonomy of Distributed Storage Systems*, www.cloudbus.org/reports/DistributedStorage Taxonomy.pdf.
[14] K. Razavi and T. Kielmann, *"Scalable virtual machine deployment using vm image caches".* In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2013, ACM, 65:165:12.
[15] D. Villegas, N. Bobroff, I. Rodero, J. Delgado, Y. Liu, A. Devarakonda and M. Parashar. *Cloud federation in a layered service model.* Journal of Computer and System Sciences, 78(5), 2012, 1330-1344.

Edited by: Dana Petcu
Received: June 27, 2016
Accepted: August 16, 2016

DOI 10.12694/scpe.v17i4.1203
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 4, pp. 313–321. http://www.scpe.org

ARCHITECTURE OF A SCALABLE PLATFORM FOR MONITORING MULTIPLE BIG DATA FRAMEWORKS

GABRIEL IUHASZ*, DANIEL POP†, AND IOAN DRĂGAN‡

Abstract. Latest advances in information technology and the widespread growth in different areas are producing large amounts of data. Consequently, in the past decade a large number of distributed platforms for storing and processing large datasets have been proposed. Whether in development or in production, monitoring applications running on these platforms is not an easy task and dedicated tools and platforms were proposed for this scope. In this paper we present a distributed, scalable, highly available platform able to collect, store, query and process monitoring data obtained from multiple Big Data frameworks. We present its architecture and initial results obtained.

Key words: Big Data, monitoring, scalability

AMS subject classifications. 68M14, 62-07

1. Introduction. Big Data technologies have become a more present topic in both academia and industrial worlds. These technologies enable businesses to extract valuable insight from their available data, hence a large number of SMEs are showing increasing interest in using these types of technologies. Distributed frameworks for processing large amounts of data, such as Apache Hadoop [1], Spark [2] [18], or Storm [3] gained in popularity and applications developed on top of them are widely accepted [15]. However, developing software that meets the high-quality standards expected for business-critical Cloud applications remains a challenge for SMEs. In this case model-driven development (MDD) paradigm and popular standards such as UML, MARTE, TOSCA hold strong promises to tackle this challenge [4].

During the development of Big Data applications it is important to monitor the performance of each version of the application, so that software architects and developers can track how their application evolves over time. It is also useful in determining the main factors that impact the quality of the different application versions [3]. Throughout the development stage, running applications tend to be more verbose in terms of logged information so that developers can get information they need, hence data-intensive applications produce large amounts of monitoring data, which need to be collected, pre-processed, stored and made available for high-level queries and visualization.

This paper introduces, for the first time, a scalable, highly available and easy deployable platform for monitoring multiple Big Data frameworks. It currently integrates resource-level metrics, such as CPU, memory, disk or network, together with framework level metrics collected from Apache HDFS, YARN, Spark and Storm. The platform is easily extensible to other Big Data frameworks, or NoSQL / relational database systems.

The paper is structured as follows: next section introduces the design drivers for our platform, presents the overall architecture of the distributed monitoring platform and details its services. Section 3 presents initial validation results obtained against Apache Hadoop, Storm and Cloudera's Oryx frameworks. Section 4 contrasts our approach to similar tools and frameworks. Finally, we conclude with a discussion of ongoing and future work in Section 5.

2. Platform Architecture. This section presents the architecture of the DICE Monitoring platform (DMon). The platform draws its name from the EC-funded DICE project [4][4] The monitoring platforms provides both historical and near real-time performance data for other tools of the DICE ecosystem.

*West University of Timisoara and Institute e-Austria Timisoara Romania, (iuhasz.gabriel@e-uvt.ro).
†West University of Timisoara and Institute e-Austria Timisoara Romania, (daniel.pop@e-uvt.ro).
‡"Victor Babes" University of Medicine and Pharmacy and Institute e-Austria Timisoara Romania, (idragan@ieat.ro).
[1] http://hadoop.apache.org/
[2] http://spark.apache.org/
[3] http://storm.apache.org/
[4] Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements (DICE) aims at providing a toolchain that makes the task of developing Big Data applications less daunting. It provides a set of artifacts (tools, models, methodology) that support software engineers in the process of designing and tuning data intensive applications.

In a nutshell, DMon is designed as a web service that enables the deployment and management of several subcomponents. Each of the subcomponents are responsible for enabling the monitoring of Big Data applications and frameworks. In contrast to other monitoring solutions [1, 2], DMon aims at providing the user with as much data as possible about the current status for the Big Data subcomponents. By doing so a wide range of new technical challenges arise. Due to the fact that DMon is serving near real-time fine grained metrics it must exhibit high availability and easy scalability.

DMon provides a distributed, high availability monitoring service. It is tailor made for Big Data technologies and is easily extendible to collect metrics from a wide range of platforms. Big data service integration/monitoring into monitoring platforms is still an open issue. The ingestion of large amounts of data in a timely manner is also an open question. During production only warning or error level logs and metrics are important. However, during development this level of detail is not sufficient. Consumption and most importantly presentation in a useful manner of collected metrics in near real time represents one of the key problems addressed by our approach. This is the main rationale behind the new distributed monitoring solution for Big Data technologies presented here.

Traditionally, web services have been build using a monolithic architecture where almost all components of a system run in a single process, usually a Java Virtual Machine (JVM). In case of this architecture type there are major advantages when it comes to deployment and networking. On the other hand scaling of such a system is a highly non-trivial task which requires running several instances of the service behind a load-balancer instance.

Although the monolithic approach seems the way to go when deploying monitoring platforms, there are some severe limitations to it. Changes to one component can have an unforeseen impact on seemingly unrelated area of the application. Thus, adding new features to the platform can be potentially really expensive both in terms of time and resources. Secondly, individual components cannot be deployed independently. That is, partial functionality cannot be achieved in those systems. Using such solutions it is likely to partially loose reusability of the entire platform. This issue can be addressed at design time by creating reusable components. In practice we observe that it is not always the case that reusability is a major concern for developers, but rather the focus is on readability of code resulting in some of the cases in side effects like loss of performance.

Taking into consideration both the pluses and the limitations exposed by a monolithic design for a monitoring platform, we decided to use another fundamentally different approach for DMon. That is, we are deploying a widely used approach in the Internet companies [6], the so called **microservice architecture** [13]. By using this architecture we replace the monolithic service with a distributed system of lightweight services. Each of the services are by design independent and narrowly focused. This approach allows us to deploy, upgrade and scale individual services rather than entire monolithic components. Because of the loosely coupled manner of microservices, code reusability is much easier and changes made to individual services do not necessarily require changes in other ones. In the microservice architecture, integration and communication between services should be done either by using HTTP (REST APIs) or using RPC requests. The use of microservice architecture allows as to group related behaviours into separate services, thus enabling us to easily modify the overall system without the hustle for editing of multiple services.

DMon uses REST APIs for communication between different services with request payload encoded as JSON messages. This makes the creation of synchronous or asynchronous messages much easier.

Figure 2.1 shows the overall architecture of the DMon platform. The context in which DMon is developed is relying on the so called lambda architecture [11] . As defined the lambda architecture consists of three layers: *speed*, *batch* and *serving layer*. In order to follow this guidelines DMon is using Elasticsearch as serving layer responsible for loading batch views of the collected monitoring data and enabling other tools/layers to do random reads on it. The speed layer will be used to look at recent data and represent it in a query function. In the case of anomaly detection, it will require the use of unsupervised learning techniques or pre-trained models from the batch layer. The batch layer needs to compute arbitrary functions on large sections of the dataset stored in Elasticsearch. That is, running long-running jobs to train predictive models that than can be instantiated on the speed layer. All trained models will then be stored inside the serving layer and accessed via DMon queries.

Core components of the platform are Elasticsearch, for storing and indexing of collected data, and Logstash

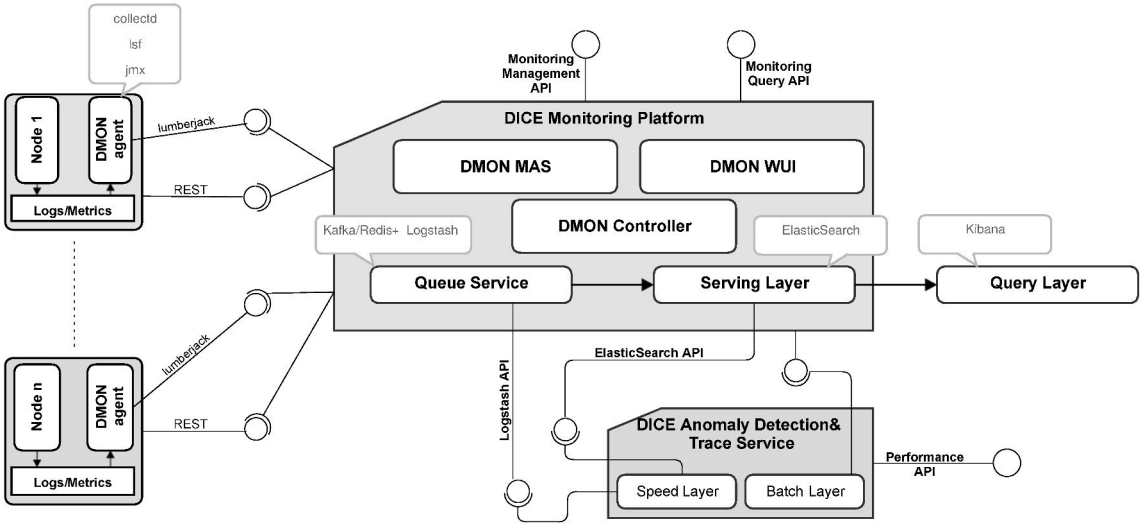


FIG. 2.1. *DMon architecture*

for gathering and processing logfile data. Kibana server provides a user-friendly graphical user interface. The main services composing DMon are the following: *dmon-controller, dmon-agent, dmon-shipper, dmon-indexer, dmon-wui* and *dmon-mas*. Each of these services will be used to control both the *core* and *node-level components*.

2.1. Core components. As the name suggests the *core components* are the backbone of the entire monitoring platform. They are used for collecting, processing, aggregating and transforming all the incoming monitoring data. One of the most essential features that is common for these components is ease of configuration, scalability and support for high throughput.

Elasticsearch [7] is an open-source, RESTful search engine based on top of Apache Lucene [12]. It is an inherently (horizontally) scalable solution which can perform near real-time processing with up to five-second latency. It also provides support for multi-tenancy, streamlined backup procedures as well as insuring data integrity. One of the most important capabilities of Elasticsearch is its ability to handle high throughput of tens or even hundred thousands of messages per second.

Logstash [17] is a tool developed in order to collect, process and forward events and log messages. Basically, it handles **E**xtract, **T**ransform and **L**oad (**ETL**) operations. It uses configurable plugins for input-output and filters in order to collect, process and load data. The input plugins can be configured to accept a wide range of inputs starting from TCP/UDP to Kafka [10] topics. Input plugins send the data for processing to the filter worker plugins. Finally, the processed data is routed to one or more output plugins such as Elasticsearch, Kafka, InfluxDB etc. Logstasg has the important property of being essentially stateless thus making it extremely scalable. For example it is possible for two Logstash instances to serve the same Elasticsearch endpoint.

DMon does not provide its own metrics visualization widgets, rather it leverages on Kibana server to create customizable dashboards for any number of metrics. Kibana [7] serves the role of browser based analytics and search interface for Elasticsearch. DMon platform includes support to graphically represent CPU, Memory and Network loads, as well as Big Data specific metrics in customizable Kibana dashboards. These visualizations are based on Elasticsearch queries and can be aggregated and plotted using a histogram based on their timestamps. Some of these visualizations are automatically created by dmon-controller service and saved into a dedicated index in Elasticsearch. It is possible to add additional visualizations manually to fit the needs of any developer.

All of the above mentioned components are part of the so called *Elasticsearch ELK* stack. This setup provides a very robust base for DMon and will be used as a proof-of-concept implementation.

2.2. Node-level components. DMon has to monitor a wide range of Big Data technologies each of which have different metrics and metrics systems. Because of these constraints we had to choose collectors that are flexible enough to accommodate a wide variety of technologies. Besides input restrictions the collectors must have a small computational footprint. By taking into account all the previous restrictions we limit the amount of "noise" produced by the presence of different collectors. Another critical feature that all the deployed collectors must obey is easiness of deployments. That is, it should be easy to deploy and configure them for thousands of physical and virtual machines.

Collectd [5] is an open-source POSIX daemon that collects, transfers and stores performance and network related data. Being a wide used tool, collectd provides the users with a great palette of options for collector plugins. This tool is used in DMon to collect system metrics, such as CPU utilization or memory / disk / networking load and throughput.

As previously presented the Logstash server is able to collect metrics and log files directly from the machine it is installed on. In this particular case it would mean that we need to have a Logstash instance on each of the monitored nodes. In the DICE context, this approach is not feasible especially because Logstash has a substantial computational footprint when using specialized filters such as grok[6]. Instead, we decided to use logstash-forwarder [7] to do the job of metrics forwarding. logstash-forwarder is designed for the purpose of log forwarding to one or more logstash server instances. By using this approach inside DMon we are eliminating node-level side effects caused by local processing of logs.

At this point it is important to note that there are several alternatives to logstash-forwarder and even collectd. Most notably there are the *Beats* [8] data shippers for Elasticsearch. Although Beats represents today an interesting alternative, this solution was not available at the design and implementation time of the DMon prototype.

Since most of Big Data frameworks are Java tools, we can use Java Management Extensions (JMX) to extract valuable metrics related to the JVM. In fact, a large number of Big Data frameworks already support exporting metrics via JMX. Thus, jmxtrans [9] tool is used in our architecture to collect attributes exported at JVM level. It's worth mentioning that both the core and node-level components of DMon may not be final and other solutions might be integrated. For example, it is possible to use rsyslog [10] instead of Logstash to process and load data into Elasticsearch. There are alternatives to Elasticsearch as well, such as NoSQL databases that support handling of time series data, such as InfluxDB [11] which is an emerging technology. Of particular interest is the collection of JMX metrics using collectd via a plugin. Although there are some available collectd plugins [12] that are able to accomplish this task, they have proven to be either slow or very resource hungry in preliminary tests.

2.3. Platform services. All components described in the previous sections have to be deployed and configured. The steps for configuration and deployment are accomplished by a number of Web services wrapping the core and the node-level components. The rest of this section details each of the wrapping services.

2.3.1. Core-level services. There are in total three core services: *dmon-controller*, *dmon-shipper* and *dmon-indexer*. All have been implemented using the Python programming language. Specifically with the Flask microframework [8]. The interface used by the services to communicate with each other takes the form of JSON encoded messages.

The **dmon-controller** service is essentially the service with which all other components communicate. It is in fact the main point of integration with the rest of the DICE solution. In particular, it will be used by all DICE components that require monitoring data. The REST API is split into two main parts: Monitoring Management API and Monitoring Query API. A swagger [13] based web UI is used for all developed services for

[5] https://collectd.org/
[6] https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html
[7] https://github.com/elastic/logstash-forwarder
[8] https://www.elastic.co/products/beats
[9] http://www.jmxtrans.org/
[10] http://www.rsyslog.com/
[11] https://influxdata.com/
[12] https://collectd.org/wiki/index.php/Plugin:GenericJMX
[13] http://swagger.io/

```
{
  ``DMON'':{
    ``query'':{
      ``size'':''<SIZEinINT>'',
      ``index'':''<indexname>'',
      ``ordering'':''<asc|desc>'',
      ``queryString'':''<query>'',
      ``tstart'':''<startDate>'',
      ``tstop'':''<stopDate>''
    }
  }
}
```

FIG. 2.2. *Example of query (JSON format)*

easy of use and a form of interactive documentation of all REST APIs.

The *Monitoring Management API*, namely *Overlord*, is used to register nodes, change configuration parameters and current status of all node-level components. It can also be used to deploy and configure node-level metrics on to registered nodes. Because of this, when registering nodes it is required that credentials for each node be supplied (username, password or key). If node-level components and services have already been deployed by other tools they only have to register the already deployed node-level service endpoints. In this scenario credentials are not needed.

Long term storage of metrics is of course a problem that has to be dealt with in all data warehousing solutions. In the case of DMon one may use the management API to create new indexes which store monitoring data. We can also export these indexes or even dump the entire dataset into a different format. By default, DMon creates an index every 24 hours. These indexes can be queried either on at a time or all at once. The exported indexes can be at any given time be reloaded into DMon or into a different Elasticsearch deployment for offline processing.

Metrics version annotation is also supported by the dmon-controller. By this we mean that metrics pertaining to a specific application version can be annotated using tags. This way we can easily query, aggregate or even compare metrics of the application. Creation of a separate index for each application version is also possible. However, it is not as versatile a solution and makes comparison of different application version performance more difficult. The dmon-controller is also responsible for generating and enacting configurations for all core components (Elasticsearch, Logstash and Kibana). The configuration is largely dependent on the data provided during the registration of each node. This data is then used to configure each component of DMon.

As already mentioned, the type of node-level component needed for monitoring is based on the Big Data service that run on each machine. During registration a list of services that are deployed on each node can be defined which are then used to setup and manage node-level services and components. Querying DMon is done using the *Monitoring Query API*, namely *Observer*. In contrast to the Management API, this one doesn't require authentication. A query example is included in Figure 2.2, showing attributes that can be specified: the size of the returned response, its ordering, or start and stop dates (in UTC). The *queryString*, which follows the same format and rules as Kibana queries, actually defines the predicate to be run on the Elasticsearch and can be used to aggregate data, or perform additional operations on the stored data [7]. Query's response may be returned in several formats, currently supported are: CSV, JSON, or plain text. Support for RDF+XML encoding using OSLC Perf. Mon 2.0 vocabulary [9] was also developed but only system metrics are curently exportable using this format. The CSV and RDF+XML query responses are generated using the dmon-controller service, which takes the JSON response from ElastiSearch and converts it to the target format.

The **dmon-shipper** microservice is meant to deploy, manage and configure logstash instances. In contrast to dmon-controller service, this service has to be located on the same machine with the controller logstash instance is located. This service is not meant to be used by external tools and services, being an internal component of DMon platform. The **dmon-indexer** microservice is used to control nodes comprising the Elasticsearch cluster.

Splitting the control of various core and node-level components into microservices we can easily separate the application logic of DMon from the code that actually drives and enacts them. Another important point is that all services besides the dmon-controller are essentially stateless. For example neither service stores the current state of the components it controls, rather it has to poll the status of the component. The dmon-controller stores some basic state and node-level information inside a relational database, which can be exported, imported, versioned or even backed up.

2.3.2. Auxiliary platform services. These are optional services that add support for scalability, availability and ease of usage of the platform. At the time of writing of this paper, these services are under development and they have not been validated yet.

Queuing service. In some instances of DMon platform, metrics that are sent by the node-level components might exceed the capabilities of Logstash to process them effectively. This might lead to data loss. There are ways to mitigate this problem. First, we could increase the number of workers assigned to the filter plugin. In the Logstash documentation it is specified that some filters (specifically grok) might cause slowdown in metrics ingestion. Second, if increasing the number of workers is not an option we could create a second instance of Logstash which can handle some of the load. The exact way Logstash would behave in these scenarios is still under investigation since Logstash and elasticsearch configurations are automatically generated and controlled by DMon. Because of this future work will involve creating performance models of all core components.

The third variant is to use a queuing service that receives all metrics and from which the Logstash instances can consume data. Certainly, this will mitigate the data loss problem but could potentially increase the time it takes for a specific metrics reading to be processed and indexed inside elasticsearch. This service is pictured in Figure 2.1, possible candidates for its implementation ranging from Kafka or Redis to a combination of MongoDB [5] and RabbitMQ [16]. The full technical stack is still an open question and will be addressed in future versions of DMon.

DMon's **dmon-wui** is a user-friendly web user interface that gives end-users an easy access to management operations of the platform. It will also include an overview of the metrics collected from the current Big Data deployment. The dmon-controller service is able to generate a dashboard definition file for Kibana [7] engine containing both system metrics (CPU, memory, network etc.) and the most popular metrics for supported Big Data frameworks.

Automatic scaling and management of the DMon platform is going to be possible using the envisioned **dmon-mas** service (DMon multi-agent service). The service is based on a multi-agent system architecture with scaling and management capabilities in mind. For achieving these goals it relies on monitoring data for the currently deployed monitoring solution so that scaling of various components and services are going to happen based on the platforms performance. By using this approach we intend in improving overall performance and resilience to the system by enhancing the DMon platform with self-healing capabilities.

Being a multi-agent system, *dmon-mas*, is going to use a variety of specialized agents. Some of the agents are going to be in charge of monitoring the current deployment, while others will be in charge of reasoning on the gathered data. Besides these two types of agents another category will comprise agents responsible of enacting the changes imposed by the performance analysis agents. Provisioning agents will be added to the platform in order to facilitate provisioning of new Virtual Machines (VMs) on a wide variety of cloud platforms like Flexiant Cloud Orchestrator (FCO), Amazon or OpenStack.

Most of the configuration and management task needed will be accomplished using the dmon-controller service. Due to the clear separation of roles between individual components, we mention that the dmon-mas service is not being part of the core services. Hence it is not mandatory, but rather recommended in order to improve performance, to start the dmon-mas in order to obtain correct functionality of the overall DMon. Currently the dmon-mas service is under investigation and a concrete working solution for the service is being implemented.

2.3.3. Node-level Services. The **dmon-agent** service is used to manage and configure all node-level components. Similarly to the dmon-shipper and dmon-indexer services, it is also stateless. The dmon-controller service issues request to each dmon-agent service with a JSON payload that contains all required information for controlling the node-level components. Each monitored node has to have a dmon-agent instance running on it.

As of writing this paper, the dmon-agent supports collectd, logstash-forwarder and jmxtrans as metrics collecting components. It is able to install all of these supported components. The installation is based on the type of big data services and the roles assigned to the node where the dmon-agent service is installed. The dmon-controller is not responsible for picking what component each dmon-agent is deploying and managing, it only sends the list of roles each node has. It is also able to interact with Hadoop, Spark and Storm deployments. This interaction is required to change or activate the metrics system for the supported big data services.

3. Initial Validation. This section details the Cloud deployments and reports on initial validation results of the platform against Oryx2 and Storm frameworks.

3.1. Cloud deployment. In order to validate the deployment of the platform in Cloud environments, we selected two setups: one public Cloud provider (namely Flexiant Cloud Orchestrator) and one private, OpenStack powered, Cloud environment hosted by West University of Timisoara. The deployment on Flexiant Cloud Orchestrator used one VM with 4 vCPUs, 8 GB RAM and 250GB hard disk.

The current version of the monitoring platforms installation procedure requires the use of *aptitude*[14] package management system present in Debian based Linux distributions. However, it can run on any POSIX compliant operating system as long as all dependencies have been installed manually.

For future development we are considering *Chef* recipes for deployment.

3.2. Deployment using Vagrant. For development purposes, Vagrant [14] deployment scripts for DMon, CDH and Storm were also created. The first script installs and configure a distribution of DMon where all core components and services are collocated on the same VM. The second vagrant script provisions 4 VMs on which it installs the newest version of the CDH together with a version of Oryx 2 toy application. The usage of all Vagrant scripts documentation can be found in the Github repository [15]. Using these scripts anyone can create a standard development/demo environment, which contains not only the latest version of DMon. All VMs are provisioned with 2 CPUs, 4 GB RAM and a HDD of 50 GB.

3.3. DMon validation against Oryx2 and Storm. In order to validate the platform against state-of-the-art Big Data technologies, we have deployed Cloudera Distribution for Hadoop (CDH) 5.4.7 and Oryx 2 on a cluster of 14 nodes on FCO as well. All VMs have the same specifications as the monitoring VM described in section 3.1. For this first round of validation we assume a worse case scenario in the sense that all DMon services and core components are collocated on the same VM sharing a common resource pool. Although, tests were done on a fully distributed deployment the majority of the testing and development was done on a single VM.

It is important to note that Oryx 2 is treated in this setup as a collection of different Big Data services not as a Big Data service. This means that Oryx2 metrics are comprised of metrics from HDFS, Yarn, Spark and Kafka (Kafka monitoring is still in early development). No Oryx 2 specific metrics are monitored.

Each time a new node is added to the DMon platform for monitoring, the platform automatically installs the monitoring agent on it. The monitoring agents collect the data from the local files and sends the data to DMon. Currently, metrics for the following technologies are collected: Apache YARN and Apache Spark [18]. System metrics (CPU, memory, network, disk, etc.) are also collected using collectd plugins. These technologies are used by Oryx 2 for both the batch and speed layer. All metrics are sent to a logstash server instance which uses the custom filters generated by DMon to transform and then load the metrics into the elasticsearch instance. During a one hour period DMon was able to collect extract the monitoring data and index it into elastic search over 337,200 events of which more than half are Big Data service specific. All metrics have been collected at a 10 second interval. Each event contains information extracted from the Big Data metrics systems and can contain as much as 20 different key value pairs.

Another set of empirical validation has been run on a smaller deployment for Apache Storm. This deployment consisted on 4 VMs with the same setup as before. A demo topology was loaded consisting of 1 spout and two bolts. DMon was able to automatically detect the running topology and dynamically adjust the mapping of pertinent metrics to each bolt and spout. The automatic topology detection is accomplished via the scanning

[14] `https://wiki.debian.org/Aptitude`
[15] `https://github.com/dice-project/DICE-Monitoring/blob/master/src/`

TABLE 4.1
Comparison of tools

	Nagios	Ganglia	SequenceIQ	Apache Chukwa	Sematex	DataDog	DMon
Scalability	Manual	Manual	-	Manual	Yes	-	Self-scaling
Elasticity	None	None	-	None	Yes	-	Yes
Deployment Model	VM	On-premise	Service	On-premise	On-premise Service	Service	On-premise Service
Installation	-	Manual	-	Manual	-	-	Service via REST API
Big Data frameworks support	Poor	Poor	Hadoop 2.x	Hadoop 2.x	Good	Good	Good and Extensible
Visualization	User Defined	Predefined	Predefined	Predefined	User-defined	User-defined	User-defined
Analytics	Alerts	-	ML Support	Anomaly detection	Alerts	Alerts, correlation	Anomaly Detection
Real-time data support	Yes	Yes	Yes	No	Yes	Yes	Yes
Licensing	Freemium	BSD	Commercial	Apache 2	Freemium	Freemium	Apache 2

for the Storm REST API. Once this is found the description of the current topology is used for the automatic mapping. In a one hour timespan DMon collected over 84,600 events.

4. Similar tools. The section presents a brief overview of different monitoring solutions that can be applied to Big Data frameworks. Some of most popular open-source and commercial solutions are contrasted to DMon on different dimensions in table 4.1.

In the context of monitoring tools, scalability is key as Big Data deployments may include thousands of nodes. Although the selected technologies (ELK stack) easily support horizontal scalability, sometimes the throughout of generated monitoring data may exceed Logstash's processing capacity. In order to cope with this, a message queue should be employed 'in front' of Logstash server(s). In our case, Kafka provides a distributed backbone and data pipeline that enables the integration into the DICE monitoring.

Elasticity, the ability to adapt to data throughput is another key design driver for our platform. DMon multi-agent (dmon-mas) service provides up/down scaling of the platform based on observed data throughput.

In terms of deployment and installation approaches, platforms may be either installed manually or automatically deployed using specialized software infrastructure, namely content management systems. The reviewed platforms all require manual installation, whereas DMon provides scripts for node provision and configuration. These may be included in orchestration frameworks. The node components are transparently installed upon node addition by the DMon controller service, thus requiring no specialised skills nor personnel.

Extensibility of the platform, i.e. easy integration of new frameworks, was central to our design. The platform provides a uniform interface to a number of Big Data frameworks. Including support for a new Big Data frameworks requires proper configuration of nodes' roles and adaptation of Logstash parsers. In this way, not only Big Data frameworks can ingest data to our platform, but we can also collect log data produced by any custom data intensive application.

In most of reviewed platforms, analytics against collected monitoring data is handled via user-defined alerts. Although these provide valuable data for Ops teams, they do not provide the level of insight required by Dev teams for optimization and validation purposes. More sophisticated, contextualized methods and tools are required. The DICE Anomaly Detection component is able to detect such anomalies and with the help of the DICE Enhancement tools will feedback this information into design-time models.

5. Conclusions and Future Work. This paper presents the architecture and initial validation of the DICE Monitoring platform, which is a distributed, highly available platform for monitoring Big Data technologies. The goal of the initial version of the platform is to demonstrate a working Proof-of-Concept that collects, stores and processes monitoring data from multiple Big Data technologies, currently supported frameworks being Apache HDFS, YARN and Spark. Designed using a microservices architecture, the platform is easy to deploy, and operate on heterogeneous distributed Cloud environments. We reported successful deployments on Flexiant Cloud Orchestrator and OpenStack using Vagrant scripts.

We are planning to extend the platform to better support scalability and elasticity in Cloud environments (e.g., Queuing service, dmon multi-agent service) and to offer a more intuitive and user-friendly Web interface (e.g., metrics visualization UI, customizable dashboard). Research-wise we will be investigating the integration of the platform with design-time artifacts, such as UML modes, meta-models and profiles in order to allow software engineers and designers to define the required metrics in their models and then propagate them along

the deployment pipeline down to DMon configuration files. Streamlining the deployment of the platform by integrating it with DevOps oriented tools, such as Cloudify and/or Chef, is going to be addressed in the future.

Acknowledgment. This work was partially funded by the European Union's Horizon 2020 Research and Innovation Programme through the DICE action (`http://www.dice-h2020.eu`) under Grant Agreement Number 643946.

REFERENCES

[1] G. Aceto, A. Botta, W. de Donato, and A. Pescapè. *Cloud monitoring: A survey.* Computer Networks, 57(9):2093–2115, 2013.
[2] K. Alhamazani, R. Ranjan, K. Mitra, F. A. Rabhi, P. P. Jayaraman, S. U. Khan, A. Guabtni, and V. Bhatnagar. *An overview of the commercial cloud monitoring tools: research dimensions, design issues, and state-of-the-art.* Computing, 97(4):357–377, 2015.
[3] L. E. Bautista Villalpando, A. April, and A. Abran. *Performance analysis model for big data applications in cloud computing.* Journal of Cloud Computing, 3(1):1–20, 2014.
[4] G. Casale, D. Ardagna, M. Artac, F. Barbier, E. D. Nitto, A. Henry, G. Iuhasz, C. Joubert, J. Merseguer, V. I. Munteanu, J. F. Perez, D. Petcu, M. Rossi, C. Sheridan, I. Spais, and D. Vladuic. *Dice: Quality-driven development of data-intensive cloud applications.* In 7th IEEE/ACM International Workshop on Modeling in Software Engineering, MiSE 2015, Florence, Italy, May 16-17, 2015, pages 78–83, 2015.
[5] K. Chodorow and M. Dirolf. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage.* O'Reilly Media, 2010.
[6] M. Fowler. *Microservice overview,* Guide. Available at http://martinfowler.com/microservices/, Jan. 2016.
[7] C. Gormley and Z. Tong. *Elasticsearch: The Definitive Guide.* O'Reilly Media, 2015.
[8] M. Grinberg. *Flask Web Development: Developing Web Applications with Python.* O'Reilly Media, Inc., 1st edition, 2014.
[9] S. Kennedy and L. Jiu. *Facilitating collaboration and interaction across the enterprise with oslc.* In Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '13, pages 374–375, Riverton, NJ, USA, 2013. IBM Corp.
[10] J. Kreps, N. Narkhede, and J. Rao. *Kafka: A distributed messaging system for log processing.* In Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece, 2011.
[11] N. Marz and J. Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems.* Manning Publications, 2015.
[12] M. McCandless, E. Hatcher, and O. Gospodnetić. *Lucene in Action.* Manning Pubs Co Series. Manning, 2010.
[13] S. Newman. *Building Microservices: Designing fine-grained Systems.* O'Reilly Media, Incorporated, 2015.
[14] M. Peacock. *Creating Development Environments with Vagrant.* Community experience distilled. Packt Publishing, 2013.
[15] D. Pop. *Machine learning and cloud computing: Survey of distributed and saas solutions.* Technical Report 2012-1, Institute e-Austria Timisoara, December 2012.
[16] G. Santomaggio and S. Boschi. *RabbitMQ cookbook.* Packt Publ., Birmingham, 2013.
[17] J. Turnbull. *The Logstash Book:.* James Turnbull, 2013.
[18] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. *Spark: Cluster computing with working sets.* In Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

Edited by: Viorel Negru
Received: June 15, 2016
Accepted: October 5, 2016

DOI 10.12694/scpe.v17i4.1204
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 4, pp. 323–330. http://www.scpe.org

EXPOSING HPC SERVICES IN THE CLOUD: THE CLOUDLIGHTNING APPROACH

IOAN DRĂGAN*, TEODOR-FLORIN FORTIŞ† AND MARIAN NEAGUL‡

Abstract. Nowadays we are noticing important changes in the way High Performance Computing (HPC) providers are dealing with the demand. The growing requirements of modern data- and compute-intensive applications ask for new models for their development, deployment and execution. New approaches related with Big Data, peta- and exa-scale computing are going to dramatically change the design, development and exploitation of highly demanding applications, such as the HPC ones. Due to the increased complexity of these applications and their outstanding requirements which cannot be supported by the classical centralized cloud models, novel approaches, inspired by autonomic computing, are investigated as an alternative. In this paper, we offer an overview of such an approach, undertaken by the CloudLightning initiative [9]. In this context, a novel cloud delivery model that offers the capabilities to describe and deliver dynamic and tailored services is being considered. This new delivery model, based on a self-organizing and self-managing approach, will allow provisioning and delivery of coalitions of heterogeneous cloud resources, built on top of the resources hosted by a cloud service provider.

Key words: autonomic computing; cloud computing; HPC as a service; resource modeling; service description language

AMS subject classifications. 68M14, 68Q10

1. Introduction. With the continuous development of cloud computing, and the integration of new resource types, novel methods are considered for application development. The diversity of available resources, coupled with the convergence of the different approaches (like fog and edge computing, Internet of Things and cloud of things, bare-metal clouds, and others), are moving the focus towards building of more complex and demanding applications, for which the classical centralized cloud approach – where a full set of information is made available to support decisions – is not enough anymore.

A different approach can be built in relation with the generic principles of autonomic computing [7] and which is centered around self-organization and management [10] to enable the construction of systems which are tailored to customers' needs. With such an approach one can equally serve the outstanding requirements of modern, large-scale distributed applications, as well as the needs of the specialized ones.

Inspired from the set of four major self-* principles (self-configuration, self-healing, self-optimization and self-protection [7]), the approach considered in the context of the CloudLightning project is primarily focused on self-configuration and self-optimization in order to organize and deliver coalitions of heterogeneous resources. The implied self-organization of resources imposes extensions to the notion of resource, by using the mechanisms of organization for 'basic' heterogeneous resources in order to offer so called 'mega-resources' (*i.e.*, CloudLightning coalitions), which are then able to respond to the management requirements of a diversity of large scale applications – like commissioning, deployment or execution –, tailored for high-performance computing.

The remainder of this paper is organized as follows: section 2 offers an overview of the CloudLightning initiative, its approach, objectives and considered use cases. Section 3 offers a description of the major components of the CloudLightning architecture. Conclusions and future work description are formulated in Section 4.

2. The CloudLightning initiative.

2.1. Concept. The CloudLightning initiative intends to offer a new delivery model that will go beyond the currently used centralized management approach. An Intersect360 report identifies "provider capability to meet performance and capacity requirements" as being one of the barriers faced by High Performance Computing (HPC) to adopting cloud computing [18]. The new delivery model developed by CloudLightning will allow to make important steps in dealing with this barrier, by allowing to provision and deliver heterogeneous cloud resources in order to be consumed by HPC applications, by the means of a specialized, but still extensible description language.

CloudLightning service descriptions enables services and resources discovery – represented by the resources which are hosted by the cloud service provider –, and decomposition mechanisms which are triggering, in

*"Victor Babes" University of Medicine and Pharmacy and Institute e-Austria Timisoara Romania, (idragan@ieat.ro).
†West University of Timisoara and Institute e-Austria Timisoara Romania, (florin.fortis@e-uvt.ro).
‡West University of Timisoara and Institute e-Austria Timisoara Romania, (marian.neagul@ieat.ro).

turn, a response from the infrastructure services resulting in a proposition of some resource gatherings (the CloudLightning coalitions) capable to cover the requirements of customers' services. Heterogeneous by their nature, the CloudLightning coalitions are composed of heterogeneous components chosen to offer a better quality of service.

Self-organization and self-management are key techniques used to discover, build and propose coalitions with minimal effort related to low-level service provisioning, and to support the automation of the cloud service lifecycle. Their successful adoption limit the over-provisioning of resources [5], currently used for high demanding services, while avoiding under-provisioning of resources and enabling important savings, both on cloud provider and cloud service consumer sides.

2.2. The main components of the solution. The CloudLightning approach is based on three major components, which are further used to build the supporting use-cases [9]. Although most of these components have been extensively researched in previous projects, it is still an outstanding challenge to implement all of them in close relation to each other.

2.2.1. The declarative approach to service provisioning. The first component of the project's approach is built in relation to some concepts coming from cloud service brokers (defined as entities which "concentrate on the negotiation of relationships between consumers and providers without owning or managing the whole cloud infrastructure" [3] and can build and offer some services created over a PaaS or IaaS support). Still, the CloudLightning approach is different, as it is based on the idea that one cannot make any assumptions about the number, type and availability of resources when self-management is in action.

With this assumption, the necessity of a CloudLightning-specific service description language (CL-SDL) was identified, such that a clear separation between the declaration on what services are required and how these services are provided can be clearly realized. This clear separation allows a high level description of requested services while the resources hosted by the cloud service provider have the capability to organize themselves and respond with several offerings, if any, based on current availabilities. The CL-SDL, which is compatible with OASIS TOSCA [14], allows users to create complex solutions that are further deployed in the CloudLightning system, and provides a detailed description of the identified SDL [19].

2.2.2. Decentralized self-management. Traditionally cloud computing was based on a centralized approach, where full information about the system and its components were made available for further decisions. Once we get more diversity and increased complexity in the cloud computing landscape, new approaches to cloud management must be considered: for example, in the case of edge computing, it "is pushing computing applications, data, and services away from centralized cloud data centre architectures to the edges of the underlying network" [4, 13].

Such an approach, which is highly relevant for the case of CloudLightning self-management, will also expose an increased complexity of the problem whilst offering support for cloud management at a greater scale than in the case of the traditional, centralized approach. In order to deal with this complexity, self-organizing is considered together with the self-management component, as similarly with the "biological self-organizing system, the global goals of the system are expected to emerge from local actions" [2, 10].

2.2.3. Exposure of heterogeneous resources. HPC relevant resource types, such as GPU and GPGPU, MIC, FPGA, programmable network routers, and others, are offering promising options for cloud computing deployments whenever compute- or data-intensive applications are expected to be deployed in a cloud environment [4, 12]. While the heterogeneity of resources may be relevant for both the loosely coupled cloud deployments (e.g., the Nebula system [4]) and the tightly coupled ones (like in the case of the HARNESS project uses-cases [6]), the self-organizing and self-management approach from CloudLightning builds a distinct approach, in order to get improved efficiency and simplified implementations regardless of the range of locally available resources or their distribution. The *declarative approach to service provisioning* is thus enriched with resource discovery mechanisms allowing easier incorporation, identification and consumption of a variety of heterogeneous resources.

2.3. Use-cases. Several use cases applications, data- or compute-intensive, are considered to demonstrate the benefits of the previously mentioned self-organizing and self-management approach and the usability of

its components. The considered use cases are HPC applications, which expose outstanding requirements in compute processing power and/or data processing capabilities. They are inspired by 1. *genomics*, where the computational cost is still an important barrier; 2. *fluid dynamics* (applied in oil and gas explorations), where large scale simulation currently require dedicated and highly expensive clusters; 3. *ray tracing*, where important steps are being made towards interactive visualizations; 4. *self-optimized scientific libraries*, where increased performance for several compute-intensive libraries of scientific functions (such as BLAS/MKL, cudaFFT, or FFTW) will be supported by the specificities of the underlying heterogeneous resources. A detailed description of the currently tackled use cases can be found in [1].

2.3.1. Genome processing. The genome processing problem that CloudLightning intends to exploit in this first use-case arises from the human gene sequencing. Set in the context of the central dogma of molecular biology ("DNA makes RNA makes proteins"), and considering the huge amount of data involved in the encoding of human DNA (approximately 3 billions base pairs, a total amount of data of up to 750MB), the problem of gene sequencing becomes highly demanding in what regards computational and storage resources. In fact, as specified in [17], "genomics is a Big Data science which is going to get much bigger".

CloudLightning intends to support this topic by demonstrating how to enable some genomics-oriented systems in a CloudLightning-enabled platform, and to benchmark their performance in the context of our approaches.

2.3.2. Fluid simulations. Sophisticated models for modeling the underground basins that store natural resources of oil and gas were developed for the industry of oil & gas exploration and exploitation. These models are based on readings coming from both seismic surveys and other means of survey. A recent interest exists in multiphase flow of fluids in porous media, typically rock formations. However, as current simulations are based on in-house solutions (workstations/clusters) that are usually used under their computing capacity, their total cost is growing.

Cost reduction is an important issue for this type of applications, and it can be addressed by parallel computing approaches. According to the study from [16], only parallelization cannot offer an alternative as I/O operations become a bottleneck, due to the huge amount of processed data and the dynamics of complex fluid simulations. An approach based on heterogeneous resources and cloud-specific developments becomes thus a promising alternative. By considering this use-case, in the context of the CloudLightning project will be identified best practices required for migrating such a demanding application type to a tailored cloud environment, while keeping costs under control.

2.3.3. Ray tracing. Ray tracing is heavily used in various industries or research areas where a photorealistic preview may offer valuable information. Once the realism of the rendered scene increases, the incurred computational costs of the ray tracing process becomes extremely large. Moreover, as the process is also time consuming, it is rather performed offline, leading to important issues related to under- and over-utilization of dedicated equipments.

There are promising experiments in using heterogeneous resources in order to deliver ray tracing in real-time [8, 15]. Thus, by migrating such a problem to a self-organizing, self-management cloud environment, it will be possible to create an environment where a tailored approach, exploiting heterogeneous resources, is being offered to fully support its timely execution.

3. The architecture of the CloudLightning solution. As identified in Section 2, CloudLightning is intended to provide customers with the possibility of deploying complex applications on a coalition of *heterogeneous resources* by using a *declarative approach to service provisioning*. In order to achieve this, the principles of self-organization and self-management are applied in a decentralized way, which allows achieving the goal of optimizing resource utilization and, as a consequence, energy consumption.

Based on the declarative approach to service provisioning, a set of design requirements were identified, such as: 1. lower service cost; 2. optimize utilization/multi-tenancy and reduce operational overhead by automation; 3. provide faster service provisioning and better performance; 4. supporting dynamic workload and resource management; 5. offering service placement optimization [11]. Based on these requirements, a succinct presentation of the core components of the CloudLightning architecture is realized in what follows.

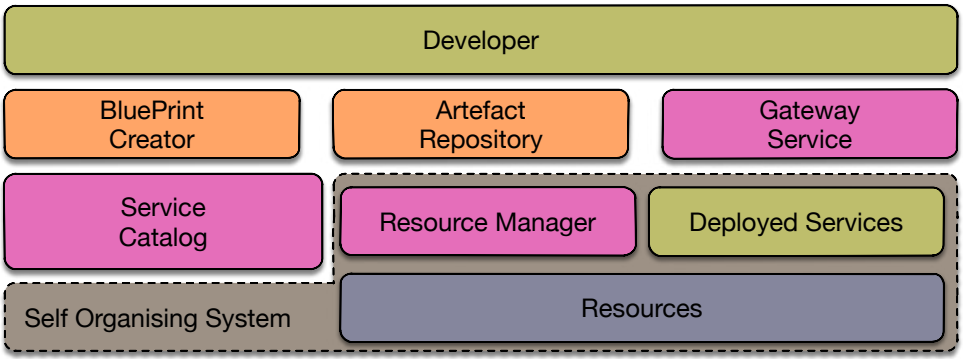


FIG. 3.1. *General overview of the CloudLightning components grouped as layers*

3.1. The layered architecture. The architecture of CloudLightning can be viewed as a three layered architecture, as depicted in Figure 3.1. Its first layer, which is represented by the *Developer*, is in charge of creating and designing application requirements that can be understood by the CloudLightning system by the means of the *CL-Blueprints*, as a representation of the CloudLightning *declarative approach to service provisioning*. Individualized blueprints are required for the deployment of different applications, and the *developer* will have to interact with components from the second layer of the architecture in order to provide such an individualization of blueprints.

The second layer of the architecture is constituted from a set of major components, namely the *Blueprint Creator*, *Gateway Service* and *Artefact Repository*, each having specific functionalities. The interactions from first level happens via the *Artefact Repository*, when a blueprint was already defined and its reuse is considered, or via the *Blueprint Creator*, when a new valid blueprint is required. In the latter case, the blueprint will be added to the repository prior to resource acquisition and application deployment. When an application is fully specified by its blueprint, it will be submitted to the *Gateway Service*, which acts as an entry point to the third layer of the architecture.

The *Self-organizing and Self-managing* level is at the core of the third level of CloudLightning architecture. At this level the system will trigger the self-organizing mechanisms to the adequate resources based on applications' requirements. Additionally, the system will provide some deployment options and will keep track of resource utilization and deployed applications, eventually by using additional repositories.

3.2. CloudLightning's architectural components. The *Blueprint* is at the heart of the CloudLightning architecture. It represents a formal description of the intended application, with some relevant annotations. Conceptually, a blueprint is usually a meaningful composition of several atomic services, a service being atomic if it cannot be decomposed into smaller services. However, a blueprint can also be imagined as a composition of both complex and atomic services.

From a technical point of view, inside CloudLightning a *Blueprint* is fully compatible OASIS TOSCA and fully supports Apache Brooklyn[1] blueprints, and includes information regarding topology, deployment and even scaling. Moreover, the *Blueprint* enables interactions between the different CloudLightning components, as they were identified in Figure 3.1, and drafted in Figure 3.2.

A sample blueprint description. Typical descriptions of CloudLightning blueprints are realized in YAML[2]. As a sample blueprint, we can consider the YAML description of a deployment scenario for a simple ray tracing application (see Section 2.3.3), which is composed of three basic components (as described in Listing 1):

- The ray tracing compute service, which will provide the required computing services;
- The controller for managing ray tracing services;
- A web portal, supporting client's access to computing services.

[1] `https://brooklyn.incubator.apache.org`
[2] `http://www.yaml.org/spec/1.2/spec.html`

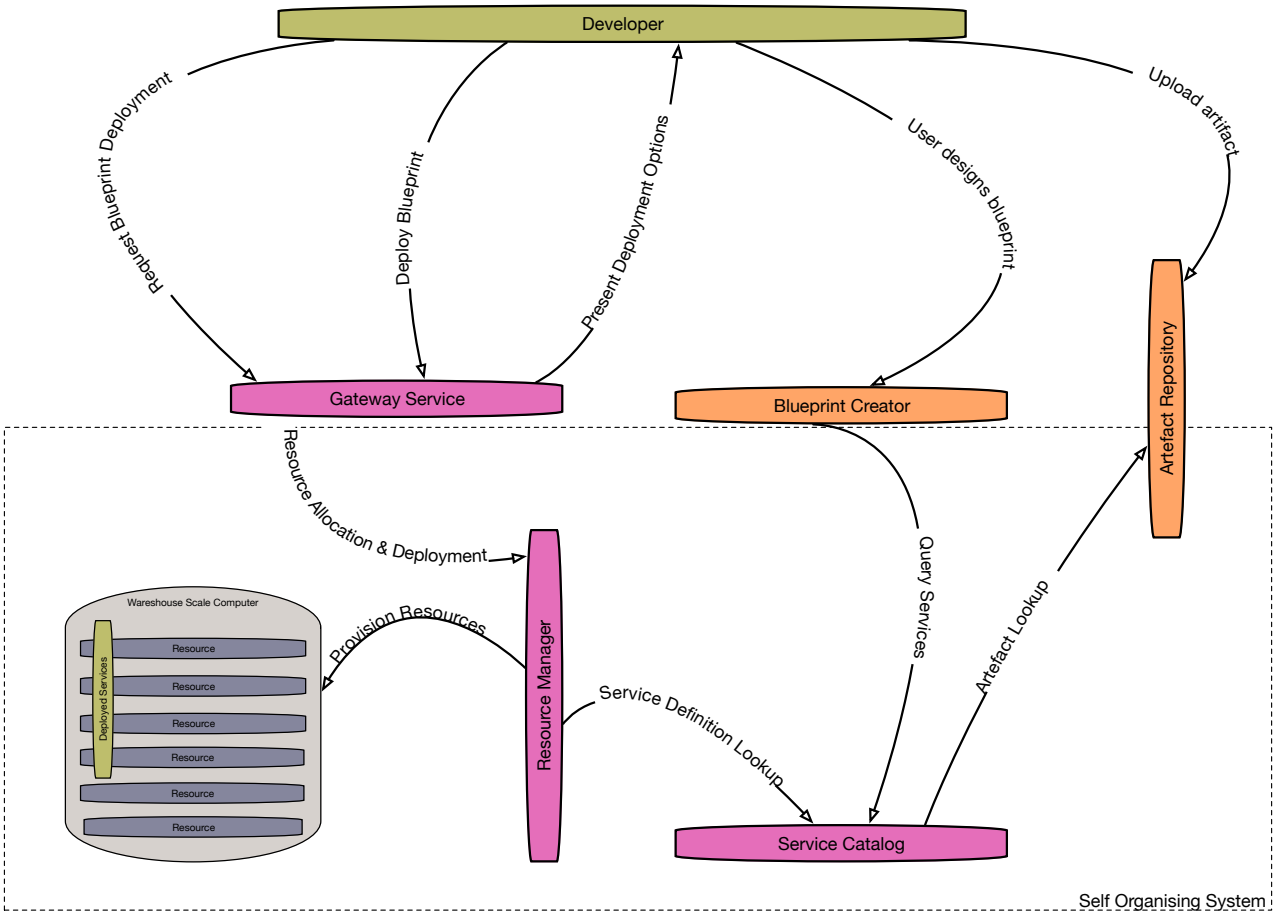


FIG. 3.2. *General overview of the CloudLightning components interactions*

For each of these components an identification, some Brooklyn configuration details, and a service type description is offered. The service type description will offer to the CloudLightning platform some additional information necessary for self-* activities, such as: the exact type of virtual machine or container that should be instantiated, and the configuration management rules to be applied.

Additionally, the blueprint specification offers the required details on components' relationships. In our sample blueprint, the *jetty_server* references the *rt_ctr* component which, in turn, references the *rt_cs* component. The latter component has some specific information: it has an abstract service type, which will serve as input for creating requests for tenders that are submitted by the service gateway to the self organizing meta-component of the CloudLightning system, thus triggering the autonomic capacities of the platform.

3.2.1. The Blueprint Creator. Acts as the component that empowers developers to easily create complex blueprints. Individual service description and definition is retrieved from a *Service Catalog* where they are defined. When some of the services needed are not defined, it provides the user with a concrete way of adding them to the *Service Catalog*. This component will also allow developers to load and reuse already defined blueprints from the previously identified *Artefact Repository*. One can view this component as being the liaison between the CloudLightning service description language (CL-SDL) and the internal components of the CloudLightning system.

3.2.2. The Service Catalog. Represents a core component of the CloudLightning architecture which is in charge of storing service definitions that can be eventually used for blueprint creation. This component is tightly connected with most of other major components of the CloudLightning architecture. Its major goal is

```
location: ieat-compute:timisoara
name: Sample Raytracing Service
services:
  - type: io.cl.entity.meta.RayTracingComputeService
    id: rt_cs
    brooklyn.config:
      cl.soso.min_performance: "100gflops"
      cl.config:
        hints:
          prefer: ['GPU', "FPGA", "CPU"]
  - type: io.cl.entity.meta.raytracing.controller
    id: rt_ctr
    brooklyn.config:
      lsf_cluster.head_node: $brooklyn:component("rt_cs")
  - type: io.cl.entity.java.jetty.web-portal-1
    id: jetty_server
    brooklyn.config:
      raytracing.controller: $brooklyn:component("rt_ctr")
```

Listing 1
Sample YAML description of a blueprint

to provide an easy to use component that keeps track of previously defined services that are made available for composing complex blueprints tailored for different applications. As previously specified, one can view the services as atomic components that can be interchanged between various users of the CloudLightning system. By doing so the process of creating custom blueprints for individual applications becomes a much easier task.

More precisely, in order to use a new service, the developer must add the atomic service(s) to the catalog such that they can be later used for blueprint creation. Additionally, the service catalog is used by the *Gateway Service* in order to decompose a blueprint in atomic services. The *Self Organizing System* will also interact with the *Service Catalog* in order to provide the *Resource Manager* with the formal description of the individual services.

3.2.3. The Resource Manager. This component is in charge with the management of the available underlying resources. Its responsibilities are related to resource reservation, provisioning, discovery and monitoring. The process of resource reservation is one of the core functionalities of the resource management system. Through this functionality, as soon as the *Self Organizing System* provides the user with a suitable set of resources to deploy its application the intended resources are reserved for this purpose and marked as being busy. An initial approach to the process of resource reservation is to reserve the resources as soon as they are presented to the user as a solution to the blueprint request. Besides reservation of resources, it is also in charge of discovering newly added resources to the infrastructure. The information about available and used resources are then aggregated and sent to the *Self Organizing System* so that they can be further used in the process of coalition formation for individual blueprints. During the deployment step, the *Resource Management* plays a crucial role by providing the means of interaction between physical resources and the CloudLightning system.

3.2.4. The Deployment Service. Represents the core component in charge of deploying an application on the reserved resources. Besides deployment of applications on readily provided coalitions the deployment service also takes care of monitoring the status of both the deployed application and of the underlying architecture. The collected monitoring information is aggregated and sent to the *Self Organizing System* where together with the information received from the *Resource Management* assists the underlying decision making of the Self Organizing System.

3.2.5. The Gateway Service. This component is in charge of creating the link between the outside world and the *Self Organizing System* from behind the scenes of CloudLightning. One of the core features that this component exposes is resource brokering and reservation. For achieving this the *Gateway Service* is exposing

a set of application programming interfaces (APIs) allowing the interaction between the outside world and the inner components of CloudLightning.

As a primary input the *Gateway Service* will receive a user defined Application Blueprint, based on which the Gateway service will call the *decomposition engine* in order to transform the user defined blueprint into a blueprint containing only atomic services. This step is required, as in the application Blueprint the developer may compose complex services in order to describe its application. The newly created blueprint is then sent to the Self Organizing System in order to provide with a coalition of resources meeting the requirements described in the decomposed blueprint. Once the *Self Organizing System* identifies the needed resources it will transmit them back to the Gateway Service which, in turn, provides them to the developer. Also, the Gateway Service provides a set of API's intended for controlling the deployed services and managing the commissioned resources.

3.2.6. The Self Organizing System. Finally, this system can be viewed as a meta-component which deals with the creation and management of coalitions where different applications run, and running at the core of the entire system. Being one of the core components of CloudLightning architecture, it is tightly coupled with other components in order to provide with adequate solutions for different application requests and it enables the decentralized self management approach and the self-organization mechanisms. The *Self Organizing System* communicates by the *Resource Manager* with individual resources in order to provide with answers to the independent application requests that arrive as blueprints in the system. It keeps a tight connection with the *Deployment* component so that it can integrate the information about already deployed services as respond to individual requests.

4. Conclusion and future work. We have shortly presented the approach and architecture of the CloudLightning system under development. The CloudLightning approach is built over a set of three central principles: (a) a declarative approach to service provisioning; (b) a decentralized, self-management approach; (c) integration of heterogeneous resources. The major components that are exposed by the CloudLightning architecture are constructed around the *Self Organizing System* and the CloudLightning service description language (CL-SDL), which offer important links with the other components of the system.

While the requirements and initial development of the CL-SDL were already delivered, the language is continually evolving to reflect the development of the *Gateway Service*. Additionally, it will support the core information required to build the various catalogues, as they are required by the CloudLightning system.

Acknowledgment. This work was partially funded by the European Union's Horizon 2020 Research and Innovation Programme through the CloudLightning action (`http://www.cloudlightning.eu`) under Grant Agreement Number 644869.

REFERENCES

[1] T. Becker, G. Gaydadjiev, P. Kuppuudaiyar, A.C. Elster, M.M. Khan, G. Gravvanis, C. Papadopoulos, T. Lynn, and D. Kenny. *D2.1.1: Use case requirements report.* Deliverable, CloudLightning Project Consortium, 2015.

[2] B.A. Caprarescu, N.M. Calcavecchia, E. Di Nitto, and D. J. Dubois. *SOS cloud: Self-organizing services in the cloud.* In Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 48–55. Springer Science + Business Media, 2012.

[3] J. Cave, N. Robinson, S. Kobzar, and H. R. Schindler. *Regulating the cloud: more, less or different regulation and competing agendas.* Less or Different Regulation and Competing Agendas (March 30, 2012), 2012.

[4] A. Chandra, J. Weissman, and B. Heintz. *Decentralized edge clouds.* IEEE Internet Computing, 17(5):70–73, September 2013.

[5] C. Delimitrou and C. Kozyrakis. *Hcloud: Resource-efficient provisioning in shared cloud systems.* SIGPLAN Not., 51(4):473–488, March 2016.

[6] O. Pell (editor). *D2.3: Validation plan.* Deliverable, HARNESS project, 2013.

[7] M.C. Huebscher and J.A. McCann. *A survey of autonomic computing: Degrees, models, and applications.* ACM Comput. Surv., 40(3):7:1–7:28, August 2008.

[8] A. Keller, T. Karras, I. Wald, T. Aila, S. Laine, J. Bikker, C. Gribble, W.-J. Lee, and J. McCombe. *Ray tracing is the future and ever will be... ACM SIGGRAPH 2013 Courses on - SIGGRAPH 13, 2013.*

[9] T. Lynn, H. Xiong, D. Dong, B. Momani, G. Gravvanis, C.F. Papadopoulos, A.C. Elster, M.M. Zaki Murtaza Khan, D. Tzovaras, K. Giannoutakis, D. Petcu, M. Neagul, I. Dragan, P. Kuppudayar, S. Natarajan, M. McGrath, G. Gaydadjiev, T. Becker, A. Gourinovitch, D. Kenny, and J. Morrison. *CLOUDLIGHTNING: A framework*

for a self-organising and self-managing heterogeneous cloud. In 6th International Conference on Cloud Computing and Services Science, CLOSER 2016, 2016.

[10] D.C. Marinescu, J.P. Morrison, and A. Paya. *Is cloud self-organization feasible?* In Adaptive Resource Management and Scheduling for Cloud Computing: Second International Workshop, ARMS-CC 2015, pages 119–127. Springer International Publishing, 2015.

[11] J. Morrison, H. Xiong, D. Dong, and B. Momani. *D3.1.2: Architecture.* Deliverable, CloudLightning Project Consortium, 2016.

[12] K. C. Nunna, F. Mehdipour, A. Trouvé, and K. J. Murakami. *A survey on big data processing infrastructure: evolving role of FPGA.* International Journal of Big Data Intelligence, 2(3):145, 2015.

[13] C. Pahl and B. Lee. *Containers and clusters for edge cloud architectures – a technology review.* In Proc. 3rd Int Future Internet of Things and Cloud (FiCloud) Conf, pages 379–386, August 2015.

[14] D. Palma and T. Spatzier. *Topology and orchestration specification for cloud applications* version 1.0, 2013.

[15] D. Pohl. *Experimental cloud-based ray tracing using intel mic architecture for highly parallel visual processing.* Intel Software Network Article, 21, 2011.

[16] E Rodrigues, JM Segura, P Vargas Mendoza, R Ausas, K Das, U Mello, MR Lakshmikantha, et al. *Exploring efficient alternatives for high performance computing requirements in coupled fluid-flow and stress simulations for the oil & gas industry.* In SPE Large Scale Computing and Big Data Challenges in Reservoir Simulation Conference and Exhibition. Society of Petroleum Engineers, 2014.

[17] Z.D. Stephens, S.Y. Lee, F. Faghri, R.H. Campbell, C. Zhai, M.J. Efron, R. Iyer, M.C. Schatz, S. Sinha, and G.E. Robinson. *Big data: Astronomical or genomical?* PLoS Biol, 13(7):e1002195, Jul 2015.

[18] Christopher G. Willard, Addison Snell, Sue Gouws Korn, and Laura Segervall. *Cloud computing in HPC: Barriers to adoption.* Research report, Intersect360 Research, 2011.

[19] H. Xiong, D. Dong, J. Morrison, I. Antoniadis, M. Neagul, K. Giannoutakis, T.-F. Fortiş, and I. Drăgan. *D5.1.1: Service description format specification.* Deliverable, CloudLightning Project Consortium, 2016.

Edited by: Viorel Negru
Received: June 15, 2016
Accepted: October 5, 2016

DOI 10.12694/scpe.v17i4.1205
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 4, pp. 331–349. http://www.scpe.org

TILING AND SCHEDULING OF THREE-LEVEL PERFECTLY NESTED LOOPS WITH DEPENDENCIES ON HETEROGENEOUS SYSTEMS

EBRAHIM ZAREI ZEFREH*, SHAHRIAR LOTFI†, LEYLI MOHAMMAD KHANLI‡, AND JABER KARIMPOUR§

Abstract. Nested loops are one of the most time-consuming parts and the largest sources of parallelism in many scientific applications. In this paper, we address the problem of 3-dimensional tiling and scheduling of three-level perfectly nested loops with dependencies on heterogeneous systems. To exploit the parallelism, we tile and schedule nested loops with dependencies by awareness of computational power of the processing nodes and execute them in pipeline mode. The tile size plays an important role to improve the parallel execution time of nested loops. We develop and evaluate a theoretical model to estimate the parallel execution time of tilled nested loops. Also, we propose a tiling genetic algorithm that used the proposed model to find the near-optimal tile size, minimizing the parallel execution time of dependence nested loops. We demonstrate the accuracy of theoretical model and effectiveness of the proposed tiling genetic algorithm by several experiments on heterogeneous systems. The 3D tiling reduces the parallel execution time by a factor of 1.2× to 2× over the 2D tiling, while parallelizing 3D heat equation as a benchmark.

Key words: Dependence loop, tiling, load balancing, communication, heterogeneous system

AMS subject classifications. 65Y05, 68M14

1. Introduction. Today, there are so many scientific applications in various fields such as meteorology, biology, medical research, signal and image processing, military industry, etc. that need high performance computing to be solved. These problems are either computationally intensive, or working on large-scale multi-dimensional data or both [1, 2]. Nested loops are one of the most time-consuming parts and the largest sources of parallelism in these problems [3, 4]. In order to meet the ever-increasing computing requirement of scientific applications, it is necessary to use high-level computational capacity and optimization techniques.

A heterogeneous computing system is a set of multiple computing nodes connected via a high-speed network interconnection, used for executing parallel and distributed scientific applications [5, 6, 7]. A homogeneous computing system is a special case of a heterogeneous computing system, in which all computing nodes have the same computing capabilities [5]. There are several ways to enhance the computational capacity of parallel computing systems such as (1) scaling up by adding more processing nodes, (2) replacement of all processing nodes with newer, faster ones, (3) upgrading computing systems by adding newer, faster nodes, (4) combing multiple clusters into a bigger computational system, known as multi-cluster systems, (5) using hybrid CPU-GPU architectures, etc. [8, 9, 10, 11]. In cases 1 and 2, the computing system remains homogeneous, but it can be very costly. In other cases, the computing system becomes heterogeneous. According to the Top500 list (http://www.top500.org), we could witness an increasing trend to heterogeneous computing systems from a 3.4% to 18.0% between June 2010 and June 2015 [12]. Hence, heterogeneity is one of the most important and challenging issues in parallel computing systems [13].

Loop optimization and parallelization have always been an important role to achieve higher performance [4]. A lot of loop optimization techniques have been developed to decrease the execution time of the nested loops and improve the performance. Loop tiling is an important loop optimization technique in scientific applications, used to improve data locality, expose fine-grained and coarse-grained parallelism, enhance cache reuse, etc. [14, 15, 16, 17, 18, 19].

In this paper, we consider the parallelization problem of perfectly nested loops with dependencies on heterogeneous systems. In order to achieve the maximum performance of nested loops with dependencies on heterogeneous systems, two issues must be adequately addressed:

- Load balancing: it is a technique that tries to distribute the data and computation across the computing resources of the parallel machine so that all tasks terminate at approximately the same time. In fact,

*Department of Computer Science, Faculty of Mathematical Sciences, University of Tabriz, Tabriz, Iran (zarei@tabrizu.ac.ir).
†Department of Computer Science, Faculty of Mathematical Sciences, University of Tabriz, Tabriz, Iran (shahriar_lotfi@tabrizu.ac.ir).
‡Faculty of Electrical and Computer Engineering, University of Tabriz, Tabriz, Iran (l-khanli@tabrizu.ac.ir).
§Department of Computer Science, Faculty of Mathematical Sciences, University of Tabriz, Tabriz, Iran (karimpour@tabrizu.ac.ir).

the goals of load balancing are optimization of resource utilization, maximization of throughput and minimization of response time [17, 20]. Processing nodes of heterogeneous systems may have different computational powers that depend on CPU speed, cache size, RAM size etc. So, load balancing is an important concept in heterogeneous systems that guarantees the amount of data and computation of any processing node correspond to their computational power [21].

- Communication: In a distributed-memory parallel architectures, each processing node has its own memory and nodes communicate together to exchanging data during program execution. Since accessing to the local memory is much faster than the remote memory, the cost of intra-node communication is much less than inter-node communication. Due to network latency of inter-node communication, data should place as close as possible to computation, referred to as the data locality [20, 22, 23, 24]. So, data and computation can be partitioned into blocks and distributed across the processing node to improve data locality and reduce communications during program execution.

In order to parallelize perfectly nested loops with dependencies on heterogeneous systems, the loop's iteration space partitioned into a series of small chunks of given tile size, executed one after another in pipeline mode. At the runtime, processing nodes communicate each other to exchange data while executing tiles. The number of inter-node communication (or tiles communication) is corresponding to the inter-tile dependency. Since communication is one of the most important reasons for performance degradation of the parallelized loops with data dependencies on heterogeneous systems, inter-tile dependency should be minimize as much as possible. To overcome communication overhead and improve the pipeline parallelism, we should determine the optimal tile size. The problem of determining the optimal tile size is NP-Hard [25]. There are many approaches that attempt to determine the near-optimal tile size in homogeneous platforms such as analytical, auto-tuning and evolutionary approaches [25, 26]. In heterogeneous platforms, tile size determined by the computational power awareness of the processing nodes [21, 27, 28]. The 3D tiling of the nested loop with dependencies for heterogeneous systems has not been given enough attention so far.

We believe that the use of 3D tiling and scheduling of perfectly nested loops with dependencies and taking into account the characteristics of heterogeneity in heterogeneous systems can enhance the execution time of the scientific applications. For this purpose, we first calculate the computational power of the processing nodes by running 3D benchmarks. Then, with computational power awareness of the processing nodes, we tile and schedule perfectly nested loops with data dependencies. Therefore, loop tiling combined with heterogeneity feature and a pipeline-like execution could help to decrees the execution time and improve efficiency of computation on heterogeneous computing systems.

In this paper, we propose a 3D tiling and scheduling approach for three-level perfectly nested loops with data dependencies on heterogeneous systems using the computational power awareness of the processing nodes. Our idea is to exploit the computational power of the processing nodes of heterogeneous platforms in order to achieve higher computing power for executing nested loops. In addition, we use loop tiling to partition the iteration space into chunks and subchunks with equal and unequal size such that the load balancing between the computational nodes increases and the internode communication is minimized as much as possible. Then, we use pipeline approaches to achieve the maximum degree of potential parallelism and consequently, the improved execution time of programs. We provide a theoretical model to estimate the parallel execution time of nested loop with dependencies and propose a tiling genetic algorithm to determine the near-optimal tile size.

The main contributions of our paper are as follows:

- We propose a 3D tiling and scheduling approach for three-level perfectly nested loops with dependencies on heterogeneous systems.
- We develop a theoretical model to estimate the parallel execution time.
- We propose a tiling genetic algorithm to determine the near-optimal tile size.

The rest of the paper is organized as follows. Section 2 describes the program model and notation and discusses an overview of related works. Section 3 describes the proposed method. Section 4 is concerned with simulations and experimental results. Finally, Section 5 is conclusions and future works.

2. Background and related work.

2.1. Program model and notation. An n-nested loop, a nested loop of depth n, is defined as a set of n loops where each loop is contained in its previous loops. If all statements are nested inside the innermost loop,

then it is called perfectly nested loop. Each iteration of n-nested loop is represented as $J = (j_1, j_2, \cdots, j_n) \in Z^n$. When a data dependency exists in a nested loop, the result of one loop iteration affects the results of other loop iterations. In fact, dependencies impose precedence constraints in the execution order of loops iterations [29]. In an n-nested loop, data dependencies are denoted by a distance dependence vector. Suppose that the matrix $D = [d_{ij}]_{n \times m}$ shows the m dependency vectors of the n-nested loop. Intra-iteration dependence occurs in the same iteration between the statements of nested loop while inter-iteration dependence occurs in different iterations [30]. Figure 2.1(a) illustrates a three-level perfectly nested loop and its iteration space denoted by $J = \{(j_1, j_2, j_3) | 1 \leq j_1 \leq N_1, 1 \leq j_2 \leq N_2, 1 \leq j_3 \leq N_3\}$. Figure 2.1 (b), (c) and (d) illustrate the iteration space, the intra-iteration and inter-iteration dependencies and the dependency matrix for the following nested loop. Nested loops categories in parallel and dependence loops. If there are no inter-iteration dependence among their loops iterations, the nested loop is called a parallel loop otherwise the nested loop is called a dependence loop [21, 31].

One of the most important of loop optimization techniques is loop tiling that could improve data locality and expose parallelism. Loop tiling decomposes an n-nested loop into a 2n-nested loop where the outer n loops move between tiles and the inner n loops traverse iteration within a tile. Suppose that the $H \in Q^{n \times n}$ be the tiling matrix that each row is a normal vector and shows the edges of the tile. V_{comp} expresses the number of iterations within a tile and V_{comm} expresses the number of iterations that need to send data to the neighboring tiles (the number of dependences exit from the tile). V_{comp} and V_{comm} are calculated by the following formulas [25, 27, 32]:

$$V_{comp}(H) = \frac{1}{|det(H)|} \tag{2.1}$$

$$V_{comm}(H) = \frac{1}{|det(H)|} \sum_{i=1}^{n} \sum_{k=1}^{n} \sum_{j=1}^{m} h_{i,k} d_{k,j} \tag{2.2}$$

Figure 2.1(e) shows the code after loop tiling transform. Figure 2.1(f) shows a $2 \times 2 \times 2$ parallelepiped tiling of the 3-nested loop and Fig. 2.1(g) illustrates the tiling matrix H for the parallelepiped tiling in Fig. 2.1(f). So, $V_{comp}(H) = 8$ and $V_{comm}(H) = 12$.

Pipeline parallelism can improve the efficiency of the nested loop with dependencies. In pipeline parallelism, each node performs its tasks, then passes its set of data along to the next node and receives the next set of data from the previous node [17]. In distributed-memory parallel systems, communication and synchronization overhead between the nodes are the important reasons of the performance degradation when running dependence loops. So, we use coarse-grain pipeline parallelism to balance trade-offs between parallelization, communication and synchronization overhead [20, 33].

2.2. Related work. There are a lot of research efforts on determining the optimal partitioning (tiling) of nested loops without dependencies on heterogeneous systems ([34, 35, 36] and references therein). However, there are a few research efforts targeting tiling problem for nested loops with dependencies on heterogeneous systems. Most of these works are bounded into 2D tiling.

Boulet et al. [37, 38] used loop tiling on heterogeneous systems for the first time. Iteration space is divided into tiles with same size and assigned column blocks with more tiles to the faster node. Then nodes execute the tiles in a row-wise order within each block to minimize latency between starting of blocks. The authors target fully permutable 2-nested loops with horizontal and vertical dependencies.

Chen and Xue [27] proposed the 2D partitioning and scheduling loops for a network of heterogeneous workstations (NOWs). As shown in Fig. 2.2, to consider heterogeneously of NOWs, the iteration space is partitioned into 2D tiles of the same shape and different sizes according to computational powers of theirs workstations. The same colored tiles can be executed simultaneously. The authors consider the doubly nested loop or two adjacent loops of nested loop with constant data dependency.

Ciorba et al. [31, 39] proposed enhancing self-scheduling algorithm for loops with dependencies on heterogeneous systems. The self-scheduling algorithms such as chunk self-scheduling, guided self-scheduling, trapezoid self-scheduling and factoring self-scheduling are dynamic scheduling algorithms that are used to schedule nested

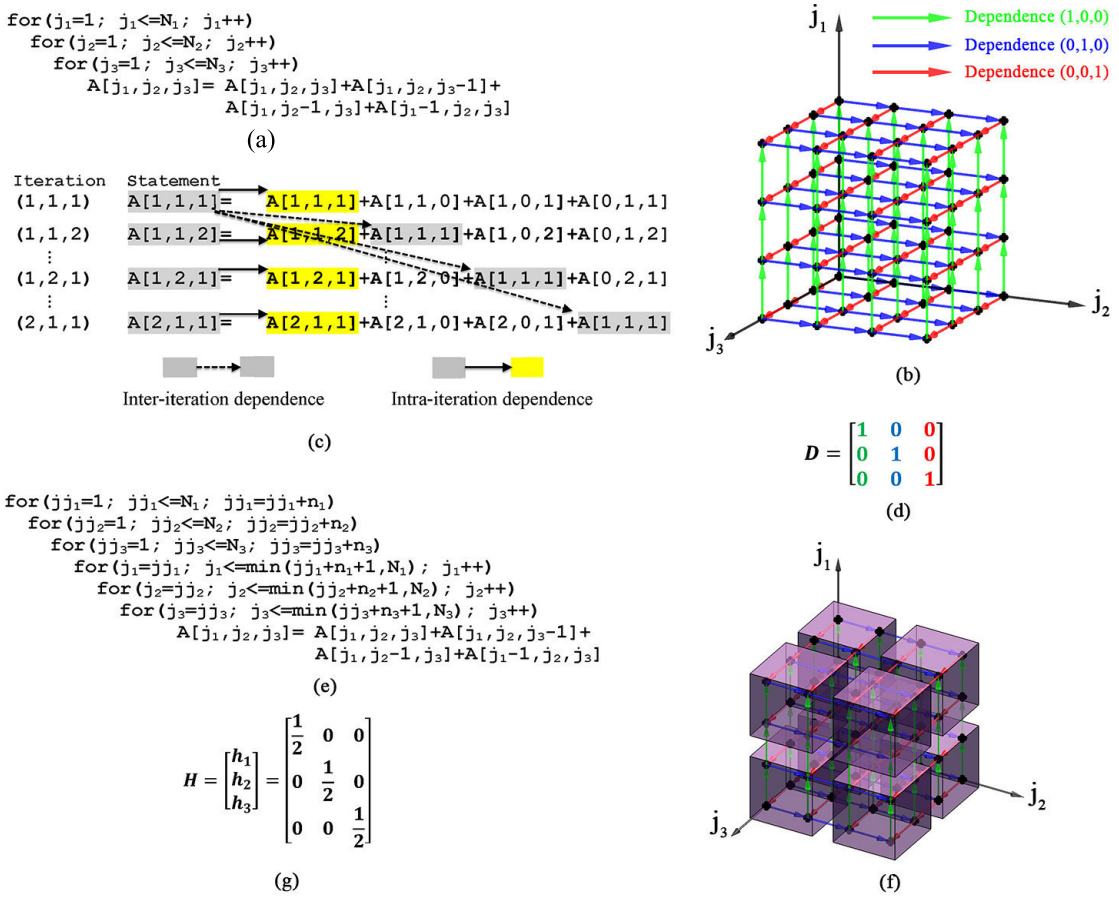


FIG. 2.1. *(a) Three-level perfectly nested loop, (b) iteration space and dependencies between iterations, (c) the intra-iteration and inter-iteration dependencies, (d) dependency matrix, (e) the tiled perfectly nested loop, (f) 3D tiling and (g) the tiling matrix*

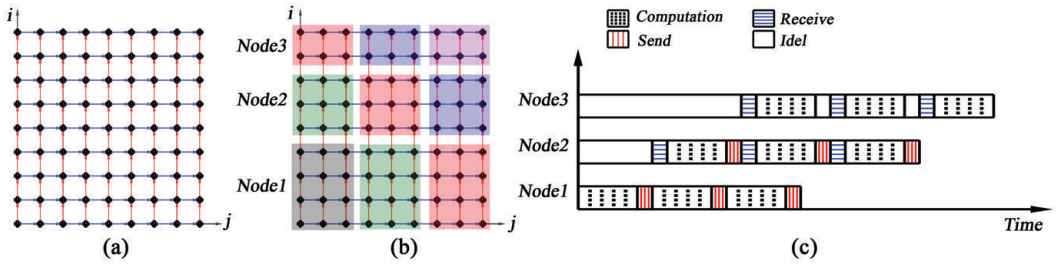


FIG. 2.2. *(a) Iteration space of nested loop with two constant data dependency, (b) 2D heterogeneous tiling and (c) parallelization strategy [27]*

loops without dependencies on homogeneous systems. They enhance self-scheduling algorithm to handle nested loops with dependencies by inserting synchronization points to enable inter-node communication. They also consider a weighted mechanism for self-scheduling algorithms to improve the performance and make them suitable for heterogeneous systems. Therefore, the iteration space is divided into chunks according to the computational power of nodes.

Andronikos et al. [21, 33, 40] claimed that the problem of finding the optimal partitioning of nested loops

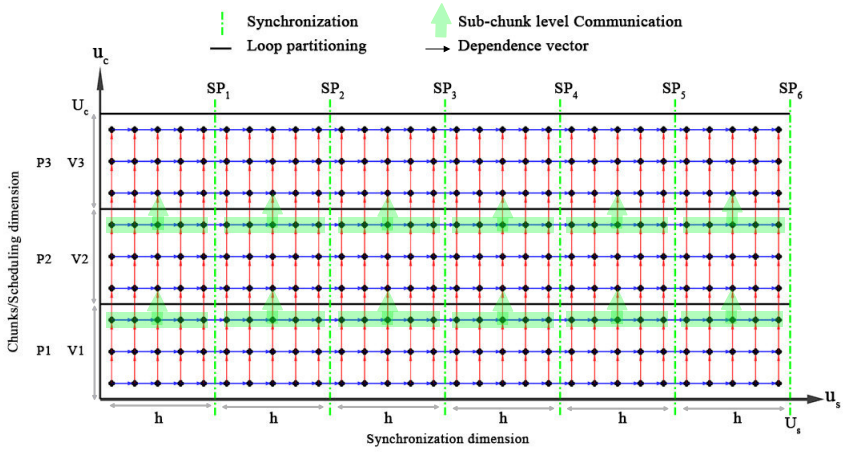


FIG. 2.3. *Tiling of a 2-nested loop with dependencies on a homogeneous system [21]*

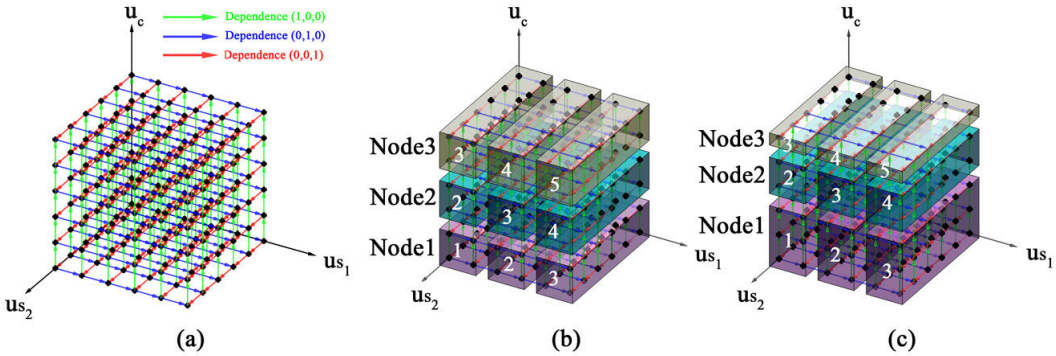


FIG. 2.4. *(a) Iteration space and data dependencies of three-level nested loop, (b) and (c) partitioning and scheduling of nested loop with proposed methods in [21, 33, 40] on homogeneous and heterogeneous systems, respectively. The tiles with the same number can be executed simultaneously.*

with dependencies for heterogeneous systems has not been given enough attention. Therefore, they proposed a theoretical model to estimate parallel execution time as a function of the synchronization frequency for nested loops with dependencies on heterogeneous systems. As shown in Fig. 2.3, the iteration space partitioned into chunks along chunk/scheduling dimension based on the computational powers of nodes using self-scheduling schema. The chunks are divided into subchunks along synchronization dimension by inserting synchronization points. They find the optimal subchunk size based on the theoretical model. This paper targets n-nested loops ($n >= 2$) with dependencies where the outer loop is considered as synchronization dimension and another loop as scheduling dimension. Figure 2.4 shows how to use this method for three-level nested loop with dependencies on homogeneous and heterogeneous systems. In this case, the u_c-dimension partitioned into chunks corresponding to the computational power of the processing nodes, the u_{s_1}-dimension partitioned into subchunks and the u_{s_2}-dimension executed as serial. Then subchunks are executed in a wavefront fashion to exploit the potential parallelism.

As mentioned above, these works are generally focused on 2D tiling of the nested loop with dependencies on heterogeneous systems. The 3D tiling of the nested loop with dependencies for heterogeneous systems has not been given enough attention so far. In this paper, we address this issue.

3. Proposed methods. In this section, we propose an approach to 3D tiling and scheduling of three-level perfectly nested loops with dependencies on heterogeneous systems. In the paper, we use the notation in [21, 33], indicated in Table 3.1. Algorithm 1 outlines the main steps of proposed method.

TABLE 3.1
Notations used within the proposed method

Parameter	**Description**
P	The number of processing nodes
p_i	The i^{th} processing node
$N = U_c \times U_{s_1} \times U_{s_2}$	The size of iteration space
U_c	The upper bound of u_c dimension
U_{s_1}	The upper bound of u_{s_1} dimension
U_{s_2}	The upper bound of u_{s_2} dimension
vp_i	The computational power of i^{th} processing node
V_i	The size of chunk i in the u_c dimension
h_1 and h_2	The size of tile in the u_{s_1} and u_{s_2} dimensions
c_{p_i}	The execution cost per iteration of i^{th} processing node
t_{p_i}	The computation time of a tile in node i
c_d	The start-up latency cost
c_c	The transfer cost per unit of data
t_s	The send time of message between a pair of nodes
t_r	The receive time of message between a pair of nodes

Algorithm1: 3D tiling and scheduling

Input:

A heterogeneous system consist of P nodes $p_1, \ldots, p_P$ with computational powers $vp_1, \ldots, vp_P$
c_p: The execution cost per iteration of nodes
c_c and c_d: Communication parameter
$U_c \times U_{s_1} \times U_{s_2}$: The size of iteration space

- Sorting computational power of nodes such that $vp_1 \geq vp_2 \geq \ldots \geq vp_P$
- Partitioning the u_c scheduling dimension into chunks of given size V_i by $V_i = U_c \times vp_i$ (Figs. 3.1(b) and 3.2(b))
- Partitioning each chunk into subchunks with unknown sizes h_1 and h_2 along the u_{s_1} and u_{s_2} synchronization dimensions (Figs. 3.1(c) and 3.2(c))
- Calculating the computation time of a tile by $t_p = V_i h_1 h_2 c_p$
- Calculating the communication time of a tile by $t_s = t_r = c_d + h_1 h_2 c_c$
- Estimation the parallel execution time, $T_P(h_1, h_2)$, based on parallel execution flow of subchunks with unknown sizes h_1 and h_2 (Fig. 3.3)

$$T_P(h_1, h_2) = (t_p + t_s) + \left(\sum_{i=2}^{P-1}(t_r + t_p + t_s)\right) + \frac{U_{s_1}U_{s_2}}{h_1 h_2}(t_r + t_p) + \left(\frac{U_{s_1}U_{s_2}}{h_1 h_2} - 1\right) t_{idle} + T_{wa}$$

- Formulate the problem of finding the optimal tiling as follows (Eq. 3.5):

$$\begin{aligned} &Minimize \;\; T_P(h_1, h_2) \\ &Subject\ to \;\; V_i h_1 h_2 \leq CacheSize_i, \;\; i = 1, 2, \ldots, P \\ &\qquad\qquad h_1 \text{ and } h_2 \text{ are integer} \\ &\qquad\qquad 1 \leq h_1 \leq U_{s_1} \\ &\qquad\qquad 1 \leq h_2 \leq U_{s_2} \end{aligned}$$

- Solving the above optimization problem using:
  1- NOMAD (Nonlinear Optimization using the MADS Algorithm)
  2- The proposed tiling genetic algorithm

Output: optimal tile sizes

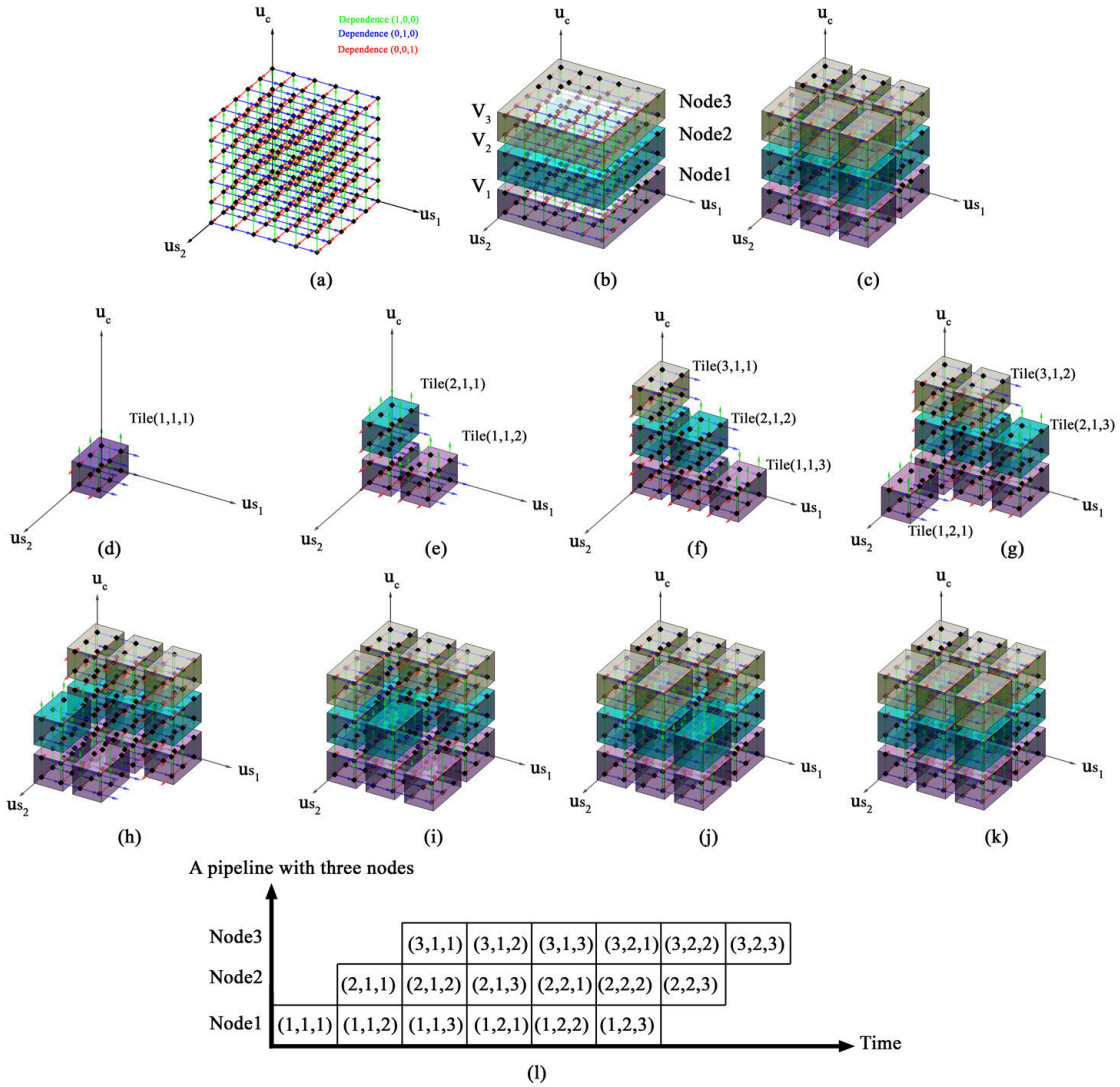


FIG. 3.1. *(a) Iteration space and dependencies vectors, (b) partitioning iteration space into three equal chunks in a homogeneous system (c) partitioning each chunk into subchunks, (d) to (k) the execution process of tiles in node 1, 2 and 3, and (l) pipelined execution of tiles in time*

Suppose, there exists P processing nodes $p_1, p_2, \cdots, p_P$ of the computational powers $vp_1, vp_2, \cdots, vp_P$ in the heterogeneous system such that $\sum_{i=1}^{P} vp_i = 1$ and $vp_1 \geq vp_2 \geq \cdots \geq vp_P$. In this paper, we consider the three-level perfectly nested loops with uniform dependencies in three dimensions. We partition the iteration space into chunks along one dimension by using self-scheduling algorithms. This dimension is called the scheduling dimension and is denoted by u_c. Let V_i be the size of the chunk i in the u_c dimension assigned to i^{th} processing node of the heterogeneous system. It should be noted that the size of each chucks is corresponding to the computational power of the processing nodes. If the distributed system has homogeneous nodes, then the sizes of chunks are equal (see Fig. 3.1(b)), otherwise the sizes of chunks are unequal (see Fig. 3.2(b)). The two other dimensions are denoted by u_{s_1} and u_{s_2} consider as synchronization dimensions. Each chunk is partitioned into subchunks with setting synchronization points along u_{s_1} and u_{s_2} dimensions (see Fig. 3.1(c) and Fig. 3.2(c)). Each of 3D boxes of points in the iteration space is considered as a tile.

In the execution flow, the tile first receives the needed data from other tiles, then does computation and

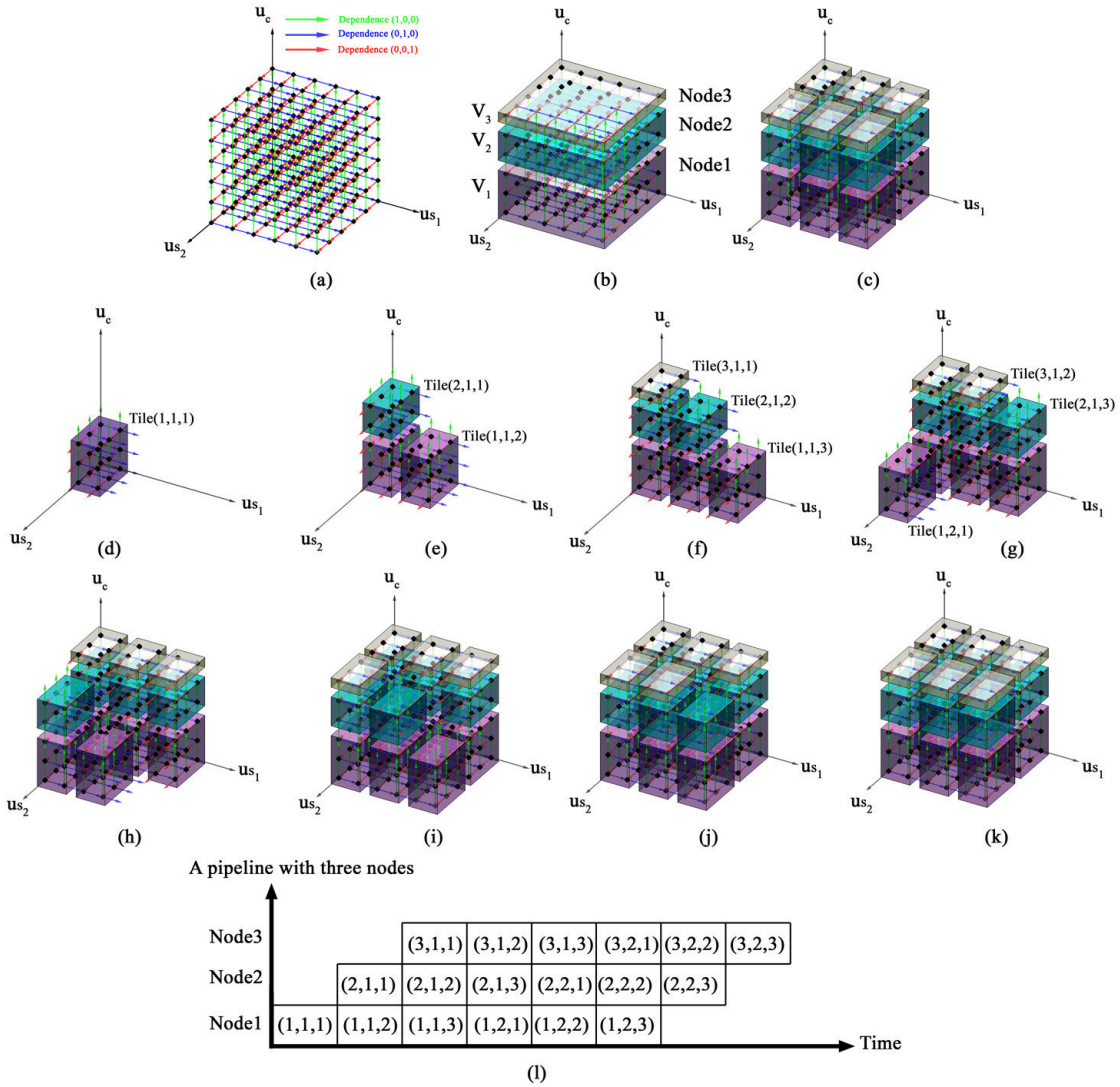


FIG. 3.2. *(a) Iteration space and dependencies vectors, (b) partitioning iteration space into three unequal chunks in a heterogeneous system, (c) partitioning each chunk to subchunks, (d) to (k) the execution process of tiles in node 1, 2 and 3, and (l) pipelined execution of tiles in time*

finally sends data to other tiles that needed it. Due to the presence of dependencies, no nodes can start the execution at the same time and we should consider a precedence order. Notice that according to the partitioning of the iteration space, dependencies $(0,1,0)^T$ and $(0,0,1)^T$ occur in each node and the dependency $(1,0,0)^T$ occurs between two neighboring nodes. As shown in Fig. 3.1(d), node 1 runs tile (1,1,1) and then sends necessary data to tile (2,1,1) that schedule on node 2. Then node 1 and node 2 simultaneously run tiles (1,1,2) and (2,1,1) respectively, as shown in Fig. 3.1(e). After that node 1, node 2 and node 3 simultaneously runs tiles (1,1,3), (2,1,2) and (3,1,1) respectively, as shown in Fig. 3.1(f). This process continues until the node 3 runs tile (3,2,3). Actually, the tiles establish a communication and synchronization mechanism between the processing nodes.

Idle time is an important factor that affects the execution time in the tiled loop. At any time during the execution of the tiled nested loops, some nodes are active and some are idle. Idle time represents the time when the node is in idle mode during the execution of the tiled iteration space. The idle time can arise due to two reasons: (1) because of the presence of dependence, a node may have to wait for the necessary data from other

nodes; (2) some nodes may have completed their works and are waiting for the last node to finish its work [41]. In homogeneous platforms, the size of tiles is the same, so choosing the shape of tiles is very important to reduce the idle time in the parallel execution. However, in heterogeneous platforms, both size and shape of the tiles have significant effect to reduce the idle time. Load balancing can reduce the idle time and guarantee that the amount of workloads of any processing node corresponds to its computational power. Therefore, in heterogeneous platforms, we use tiles with the same shape and different sizes such that nodes complete the execution of their tiles at the same time. Figure 3.2 shows the heterogeneous tiling for a heterogeneous platform with normalized computational powers $VP = \{0.5, 0.33, 0.17\}$.

To estimate the parallel execution time of nested loops with dependencies on heterogeneous systems, we need a communication and computation cost model. We use the notations in [21] and extend them.

3.1. Computation cost model. The computation time of a tile in node i is defined as a function of the number of iterations within a tile, $V_{comp}(H)$, multiplied by the execution cost per iteration, c_{p_i}. We can calculate it as follows:

$$t_{p_i} = V_{comp}(H) \times c_{p_i} \tag{3.1}$$

3.2. Communication cost model. We consider heterogeneous computing systems of P processing nodes p_1, p_2, $\cdots$, p_P that is connected with homogeneous communication links. In this work, we use the one-port model as the communication cost model to quantify the communication overhead between the processing nodes. In one-port model, a node can either send or receive a message at each time step and distinct node pairs communicate simultaneously. There are two different costs to transfer a message from one node to another: (1) the start-up latency cost between a pair of nodes, c_d; (2) the transfer cost per unit of data between a pair of nodes, c_c [17, 21]. We suppose that the send (t_s) and receive (t_r) times of a message between each pairs of nodes are equal since the number of message elements are the same in the process of sending and receiving. The communication time of a tile is defined as a function of the start-up latency cost and the number of iterations that need to send data to the neighboring tiles and the transfer cost per unit of data as follows:

$$t_s = t_r = c_d + V_{comm}(H) \times c_c \tag{3.2}$$

3.3. The proposed theoretical model. In this paper, we consider parallelizing the three-level perfectly nested loops with dependencies in three dimensions on heterogeneous computing systems. We tile and schedule these loops with the computational power awareness of the processing nodes and execute them in pipeline mode. To estimate the parallel execution time of nested loops, we build a theoretical model as a function of tile sizes.

As shown in Figs. 3.1 and 3.2, we partition the iteration space of nested loop into 3D tiles. Let V_i be the size of one side of the tile (i, j, k) along the u_c dimension assigned to i^{th} processing node. To satisfy load balancing, we calculate V_i as a function of the computational power of processing node by $V_i = U_c \times vp_i$. Suppose that h_1 and h_2 are the size of other sides of the tile along the synchronization dimensions u_{s_1} and u_{s_2}. h_1 and h_2 are the same for all tiles. So, the computation time of a tile in node i is calculated by $t_{p_i} = V_i h_1 h_2 c_{p_i}$ and the communication time of a tile in node i is calculated by $t_s = t_r = c_d + h_1 h_2 c_c$ because only dependency $(1, 0, 0)^T$ occurs between two neighboring nodes.

Figure 3.3 shows the parallel execution flow of tiled nested loops on homogeneous and heterogeneous platforms for Figs. 3.1 and 3.2. In the following, we consider the parallel execution flow and construct a formula to estimate the parallel execution time. Here, we use the master-worker model. The processing nodes (or workers) send a request message for assigning the work to the master. Master, that has all the information about the nodes, receives the requests, calculates the chunk sizes and assigns them to the processing nodes. The duration between sending a request and assigning a chunk to nodes is considered as the work assignment time and is denoted by T_{wa}. The nodes are responsible for executing the assigned chunk. Node 1 starts the execution of the tile $(1, 1, 1)$. Due to the presence of dependence, node 2 should be expected to receive data from node 1. This is the idle time and is shown by white strip in Fig. 3.3. Node 1 completes the execution of the tile $(1, 1, 1)$, sends the necessary data to node 2 and starts the execution of tile $(1, 1, 2)$. Node 2 after receiving the necessary data from node 1 executes tile $(2, 1, 1)$ and sends the necessary data to node 3. Node 3 is the last node and does not need to send data. Node 3 should be expected to receive data from node 2, so there is an idle time between the operations of execution and receiving.

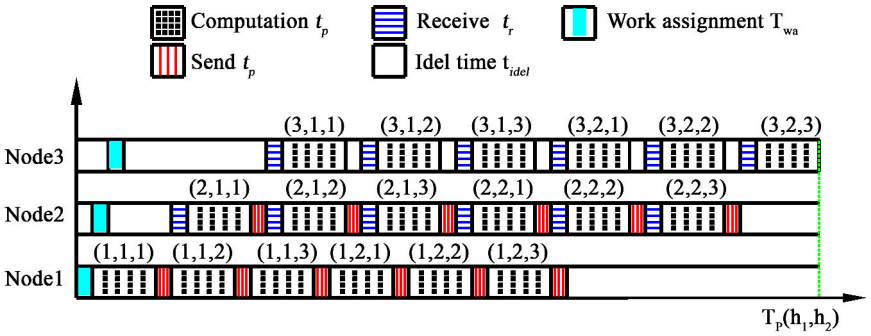


FIG. 3.3. *Parallel execution flow for three nodes*

Suppose that the theoretical parallel time, $T_P(h_1, h_2)$, is the parallel execution time of the last tile that is carry out by node P. All nodes have to *receive*, *compute* and *send* except for the first and last nodes. Node 1 only computes and sends data. So, the time required to compute each tile in node 1 and send the necessary data to node 2 is $t_{p_1} + t_s$. The time needed to receive data, compute and send data of the first tile in node $2, 3, \cdots, P-1$ is $\sum_{i=2}^{P-1}(t_r + t_{p_i} + t_s)$. The last node, node P, only receives data and computes. So, the time needed to receive the necessary data from node $P-1$ and compute all tiles in node P is $\frac{U_{s_1}U_{s_2}}{h_1h_2}(t_r + t_{p_P})$. Node P also spent $(\frac{U_{s_1}U_{s_2}}{h_1h_2} - 1)t_{idle}$ idle time for receiving data from node $P-1$. t_{idle} approximately equals to t_s. Therefore, the total parallel execution time is

$$T_P(h_1,h_2) = (t_{p_1} + t_s) + \left(\sum_{i=2}^{P-1}(t_r + t_{p_i} + t_s)\right) + \frac{U_{s_1}U_{s_2}}{h_1h_2}(t_r + t_{p_P}) + \left(\frac{U_{s_1}U_{s_2}}{h_1h_2} - 1\right)t_{idle} + T_{wa} \tag{3.3}$$

Since the processing nodes of homogeneous computing systems have the same computational power, V_i and c_{p_i} are the equivalent for all nodes. In the heterogeneous computing systems, processing nodes have different computational power. Therefore, the execution costs per iteration c_{p_i} of nodes are different. According to load balancing, the best state is when all nodes execute their assigned tiles at the same time, in the other words $t_{p_1} = t_{p_2} = \cdots = t_{p_i} = \cdots = t_{p_P}$. It is noticed that a perfect load balancing is not always possible. In this case, we want that all nodes execute their tiles at approximately the same time as much as possible $t_{p_1} \cong t_{p_2} \cong \cdots \cong tp_i \cong \cdots \cong t_{p_P}$. When running multiple tiles in parallel, maybe a node, which finishes the execution of its tile, has to wait for the other one to complete its execution before they could exchange data. To control the situation in heterogeneous systems, we consider $t_p = max\,(t_{p_1}, t_{p_2}, \ldots, t_{p_i}, \ldots, t_{p_P})$. Therefore, we have

$$T_P(h_1,h_2) = (t_p + t_s) + \left(\sum_{i=2}^{P-1}(t_r + t_p + t_s)\right) + \frac{U_{s_1}U_{s_2}}{h_1h_2}(t_r + t_p) + \left(\frac{U_{s_1}U_{s_2}}{h_1h_2} - 1\right)t_{idle} + T_{wa} \tag{3.4}$$

h_1 and h_2 require fine-tuning so that nodes can start their computation as soon as possible and achieve minimum parallel execution time. We also consider two constrains:

1. If we want to improve data locality in each node, then data items should stay in the cache between successive uses. In order to get a good performance, tile sizes are better to fit in the cache of nodes.
2. Since the sides of the tile are positive integer, we need integer solutions for h_1 and h_2.

Considering these observations, we have a nonlinear pure integer-programming problem (NLIP) as follow:

$$\begin{aligned} &\textit{Minimize} && T_P(h_1,h_2) && \\ &\textit{Subject to} && V_ih_1h_2 \le CacheSize_i, && i = 1,2,\ldots,P \\ &&& 1 \le h_1 \le U_{s_1}, 1 \le h_2 \le U_{s_2}, h_1 \text{ and } h_2 \text{ are integer} && \end{aligned} \tag{3.5}$$

Nonlinear integer programming problems are NP-complete. These problems can solve using nonlinear integer programming solvers or evolutionary approaches. In this paper, we use the NOMAD (Nonlinear Optimization using the MADS Algorithm) [42] as a nonlinear integer programming solver and proposed an evolutionary approach based on the genetic algorithm to find a near-optimal solution that minimize $T_P(h_1, h_2)$.

3.4. The proposed tiling genetic algorithm. In this section, we use Genetic Algorithm (GA) to solve the nonlinear integer-programming problem, Eq. 3.5, derived from the 3D tiling of nested loops with dependencies on heterogeneous systems.

The GA is a population-based heuristic search that follows an iterative process toward better solutions. The GA begins with an initial random population of the problem solution, called chromosomes. In each iteration, the fitness of every chromosome in the population is evaluated by using objective function. The fitter chromosomes are stochastically selected and then evolutionary operators such as crossover and mutation are used to generate new population. The GA is terminated for a maximum number of generations [25, 43].

Problem encoding. Each problem solution is represented by a chromosome. Here, chromosome is specified as a pair of integer number $\langle h_1, h_2 \rangle$ where $1 \leq h_1 \leq U_{s_1}$ and $1 \leq h_2 \leq U_{s_2}$.

Initial population. We use a random integer number generator to create the initial population of chromosomes. To generate a chromosome, the h_1 and h_2 are defined randomly by using formulas $h_1 = Round\,(1 + U_{s_1} \times Rand())$ and $h_2 = Round\,(1 + U_{s_2} \times Rand())$ where U_{s_1} and U_{s_2} are the upper bound of h_1 and h_2, respectively. The function $Rand()$ returns standard uniform distribution on the interval $(0, 1)$ and the function $Round(x)$ returns rounding of the elements of x to the nearest integer. So, $h_i = Round(1 + U_{s_i} \times Rand())$ generates integer values from the uniform distribution on the interval $[1, U_{s_i}]$ for $i = 1, 2$.

Fitness function. The main objective is to find integer values h_1 and h_2 such that the parallel execution time $T_P(h_1, h_2)$ of the heterogeneous system with P processing nodes is minimized. In addition, we have a constraint to fit the tiles into the cache memory of the processing nodes of the heterogeneous system. When the requiring space for the iteration points within tiles is not exceed the cache size of the processing nodes, the tiles are feasible (on the other hand, the chromosomes are feasible). According to Eq. 3.6, we consider the objective function as a summation of two positive numbers, the parallel execution time and the penalty value computed for the chromosomes. We use a constant value for penalty which is zero for feasible chromosomes and $c > 0$ for an infeasible one.

$$
\begin{aligned}
Objective(\langle h_1, h_2 \rangle) &= T_P(\langle h_1, h_2 \rangle) + Penalty(\langle h_1, h_2 \rangle, M) \\
M &= min_{i=1,\ldots,P} \frac{Cache\ size\ of\ node\ i\ in\ byte}{V_i \times (\#Byte\ of\ data\ type)} \\
Penalty(\langle h_1, h_2 \rangle, M) &= \{ \begin{array}{ll} 0 & if h_1 \times h_2 \leq M \\ c & if h_1 \times h_2 > M \end{array}
\end{aligned} \tag{3.6}
$$

We assign a fitness value to each chromosome in the population, calculated by Eq. 3.7. The better chromosome, the bigger fitness value.

$$
Fitness(\langle h_1, h_2 \rangle) = \frac{1}{Objective(\langle h_1, h_2 \rangle) + 1} \tag{3.7}
$$

Selection, crossover and mutation operators. After assigning the fitness value to each chromosome in the current population, the roulette wheel selection method is used to choose a couple of parent chromosomes for the crossing over operation. The bigger the fitness value of chromosomes are, the more chances to be chosen they have. Crossover and mutation are two important genetic operators. Crossover is an exploitation operator that is used to create new population by combining a couple of parent chromosomes. Mutation is an exploration operator that is used to maintain diversity in the new population [43]. Here, the crossover operator is applied to the selected parent chromosomes using an arithmetic crossover. The crossover operator is done with the combined probability, $P_{Crossover}$, as follows:

$0 \leq \lambda \leq 1$ is chosen randomly
If $Rand() \leq P_{Crossover}$

$\text{ChildChromosome}_1 = \lambda\times \text{ ParentChromosome}_1 + (1-\lambda)\times \text{ ParentChromosome}_2$
$\text{ChildChromosome}_2 = (1-\lambda)\times \text{ ParentChromosome}_1 + \lambda\times \text{ ParentChromosome}_2$
else
$\text{ChildChromosome}_1 = \text{ParentChromosome}_1$
$\text{ChildChromosome}_2 = \text{ParentChromosome}_2$
end

After applying the crossover operator, the mutation operator with the probability, $P_{Mutation}$, is applied to newly generated chromosomes. It replaced the value of the chosen chromosomes, $\langle h_1, h_2\rangle$, with integer values from the uniform distribution between the upper and lower bounds of h_1 and h_2.

Replacement Scheme. After generating the new population using selection, crossover and mutation operations, the GA replaces the current population with the new one. We use elitism in the replacement scheme. If the fittest chromosome in the current population is better than the fittest chromosome in the new population, then it is moved to the next population directly. Elitism is important since it allows preserving the fittest chromosome over the time.

4. Experiments and results. In this section, our simulation and experimental results are presented. We evaluate the performance of the proposed theoretical model and tiling genetic algorithm by using the 3D heat equation, three-level perfectly nested loops with dependencies, as a benchmark. Table 4.1 shows the specifications of nine classes of processing nodes used in experiments. They are multi-core processors. A 100 Mbits/s fast Ethernet network is used to interconnect processing nodes. The benchmark is implemented in C using OpenMP for intra-node communication and MPI for inter-node communication.

TABLE 4.1
Specifications of processing nodes

	Processing nodes								
	1	2	3	4	5	6	7	8	9
Name of processors	Intel Core 2 Duo T5870	Intel Pentium E5300	Intel Core 2 Duo E7500	Intel Core i3 2350M	Intel Pentium G620	Intel Pentium G2020	Intel Core i5 2410M	Intel Pentium G2030	Intel Core i7 4710HQ
#Processors	1	1	1	1	1	1	1	1	1
CPU Speed (GHz)	2.00	2.60	2.93	2.30	2.60	2.90	2.30	3.00	2.50
#Cores	2	2	2	2	2	2	2	2	4
L1 Cache (KB)	2 x 32	2 x 32	2 x 32	2 x 32	2 x 32	2 x 32	2 x 32	2 x 32	4 x 32
L2 Cache (KB)	2048	2048	3072	2 x 256	2 x 256	2 x 256	2 x 256	2 x 256	4 x 256
L3 Cache (MB)	-	-	-	3	3	3	3	3	6
Memory type	DDR2	DDR2	DDR3	DDR3	DDR3	DDR3	DDR3	DDR3	DDR3
RAM (GB)	4	2	2	4	2	4	4	4	8
Normalized Computational Power for 3D Heat Equation	0.0529	0.0531	0.0920	0.1029	0.1197	0.1383	0.1405	0.1438	0.1568

We use hierarchical tiling to exploit the computational power of all cores in multi-core nodes. For this purpose, we first partition the iteration space of nested loops with dependencies into chunks and assign each chunk to each node. Due to the dependence, each assigned chunk is partitioned to subchunks and run in pipeline mode to achieve the maximum degree of parallelism between nodes of a heterogeneous system. In multi-core node, the subchunk is tiled again and assign to their cores. Figures 4.1(a) and (b) show the pseudo code of a subchunk of size $n_i \times n_j \times n_k$ of the 3D heat equation and the wavefront-parallel 3D heat equation for a subchunk of size $n_i \times n_j \times n_k$, respectively [44].

We execute the 3D heat equation on each node several times, measure the average execution time and calculate the computational power of the processing nodes. These values, which are used as weights that scale

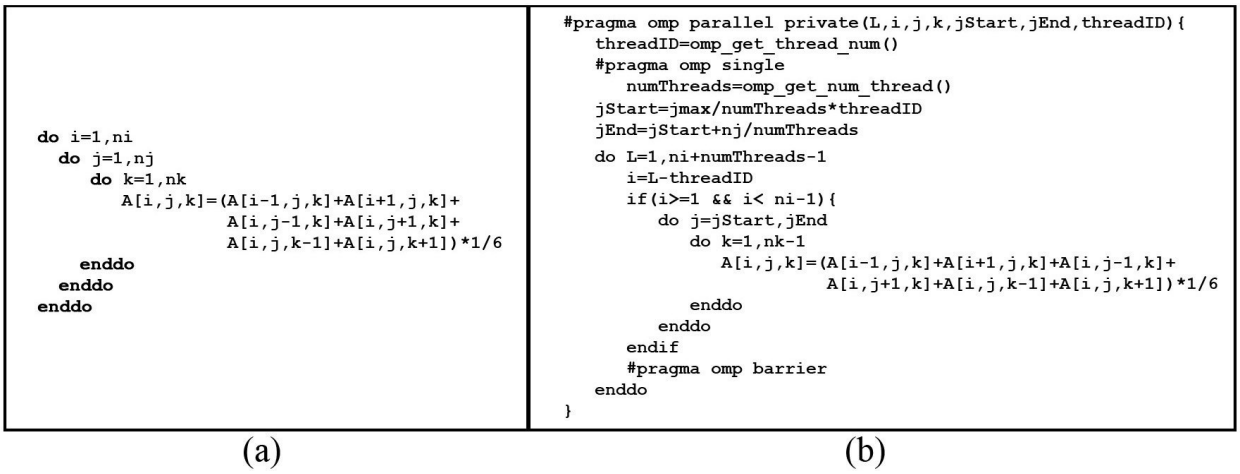


FIG. 4.1. *(a) Pseudo code of 3D heat equation and (b) the wavefront-parallel 3D heat equation [44]*

TABLE 4.2
Specifications of experiments

Experiment	Node type											
#1	1	1	1	1	1	1	1	1	-	-	-	-
#2	3	3	3	3	1	1	1	1	-	-	-	-
#3	8	8	8	8	1	1	1	1	-	-	-	-
#4	7	7	7	7	4	4	4	4	1	1	1	1
#5	9	8	7	6	5	4	3	2	1	-	-	-

the size of each chunks assigned to each processing node, are normalized and showed in the last row of Table 4.1.

Simulations and experimental results are presented for one homogeneous and several heterogeneous computing systems to evaluate the performance of the proposed theoretical model for estimating the parallel execution time and the tiling genetic algorithm for finding the near-optimal tiling. Table 4.2 describes the specification of experiments.

All nodes of computing systems connected together with homogeneous communication links. An MPI program in C used to exchange data with different sizes between every pair of processing nodes. We measured the average time to send and receive messages. The estimated value of the start-up latency, c_d, and the transfer cost per unit of data, c_c, between each pairs of nodes are 300e-06 and 0.80e-06, respectively.

We approximate the execution cost per iteration of each node as a function of tile size (namely, the constant value V_i and variable integer values h_1 and h_2) to consider processor heterogeneity, the heterogeneity in memory structure, and the effect of paging [45]. To do so, we run the benchmark for several integer values h_1 and h_2, then the execution cost for all integer values of $1 \leq h_1 \leq U_{s_1}$ and $1 \leq h_2 \leq U_{s_2}$ was predicted using bilinear interpolation methods. The execution time of each tile is measured once and is used several times in practice. So, the cost of calculating the execution time of each tile will be amortized on the total execution time. Since intra-node communication cost is negligible compared to inter-node communication cost, we did not directly consider intra-node communication cost in Eq. 3.4. In fact, intra-node communication cost indirectly have regarded in c_{p_i} parameter.

4.1. Evaluation of the theoretical model. In this section, we evaluate the proposed theoretical model for estimating the parallel execution time and genetic tiling algorithm for finding near-optimal tiling. In experiment 1, we consider a homogeneous computing system consists of eight same processing nodes of type 1 as mentioned in Table 4.2. First, the computational powers of these nodes are normalized such that the summation of them equals one. The size of the iteration space is $U_c \times U_{s_1} \times U_{s_2} = 1024 \times 1024 \times 1024$. The size of the assigned

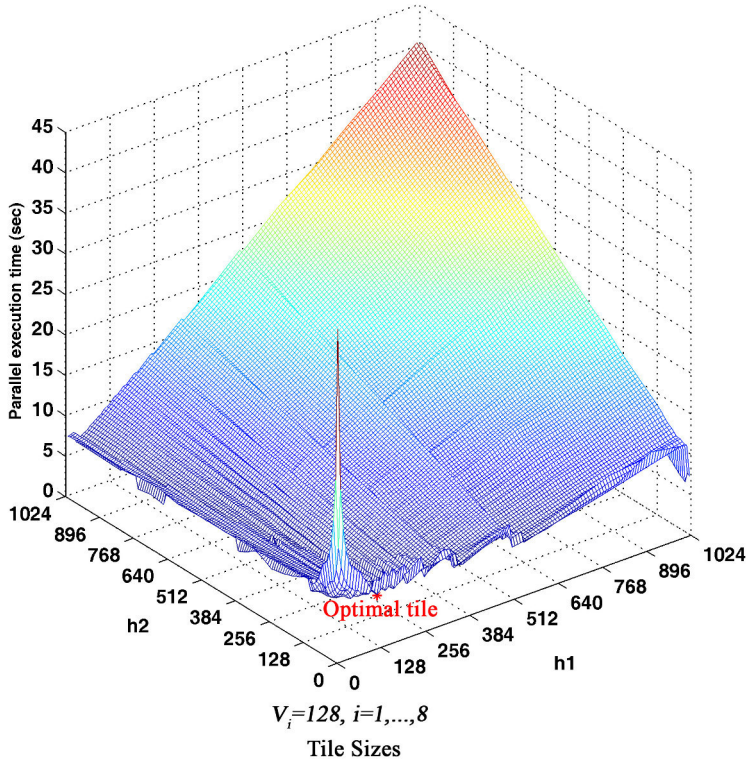


FIG. 4.2. *Parallel execution time for different tile sizes in experiment 1*

chunk to the respective nodes in the u_c dimension is $V_i = U_c \times vp_i = 1024 \times 0.125 = 128$ for $i = 1, \ldots, 8$. Now, we can determine the optimal size of other sides of 3D tile, h_1 and h_2, along the synchronization dimensions u_{s_1} and u_{s_2}. Figure 4.2 shows the parallel execution time for various tile sizes (the V_i, h_1 and h_2). By searching the entire space of solutions of h_1 and h_2, the optimal value of $\langle h_1, h_2 \rangle$ are $\langle 128, 16 \rangle$. As theoretically expected, when the tile sizes fit into the cache of nodes, the cache utilization and data locality maximize and it would lead to improvement in the parallel execution time.

It is to be noted that searching the entire solution space of tile sizes can be very time consuming, especially in the large solution space. So, we use proposed tiling genetic algorithm and the nonlinear integer programming solver, NOMAD, to find the near-optimal value of h_1 and h_2 from the theoretical model, Eq. 3.5. Table 4.3 shows the results of 30 runs of the tiling genetic algorithm and NOMAD to solve Eq. 3.5 in experiment 1. The comparison of the average and standard deviation of $\frac{|AT-OPT|}{OPT}$ indicate that the reliability of the proposed tiling genetic algorithm to find the near-optimal value of h_1 and h_2 is better than NOMAD algorithm. On the other hand, the accuracy of value $\langle h_1, h_2 \rangle$ of the tiling genetic algorithm and NOMAD is achieved by the error less than 0.0055 and 0.0248 in 30 runs, respectively.

4.2. Comparison of 3D and 2D Tiling. As mentioned in the related work, the proposed methods in [21, 33, 40] could find the near-optimal partitioning of 3-nested loop with dependencies for homogeneous/heterogeneous computing systems. It targets two loops of the nested loop and considers the outer loop as synchronization dimension and another loop as scheduling dimension. We refer to this work as the 2D tiling. In the following, we compare the proposed 3D tiling with the 2D tiling for the 3-nested loop with dependencies on homogeneous/heterogeneous computing systems. We find the near-optimal 3D tiling and 2D tiling for one homogeneous and several heterogeneous computing systems. Table 4.4 shows the near-optimal tile sizes of 2D and 3D tiling with/without considering heterogeneity feature and Fig. 4.3 plots their corresponding execution time. The results presented in Tables 4.5 and 4.6 show the speedup of execution time for the 3D tiling versus

TABLE 4.3
The results of 30 runs of the tiling genetic algorithm and NOMAD Algorithm in experiment 1

Run	NOMAD Algorithm					Genetic Tiling Algorithm					
	h_1	h_2	TT	AT	$\frac{\lvert AT-OPT\rvert}{OPT}$	Generation	h_1	h_2	TT	AT	$\frac{\lvert AT-OPT\rvert}{OPT}$
1	128	16	6.0560	6.0643	0.0000	68	128	16	6.0560	6.0643	0.0000
2	227	9	6.3812	6.3904	0.0538	97	128	16	6.0560	6.0643	0.0000
3	136	15	6.2110	6.2199	0.0257	500	128	15	6.2429	6.2518	0.0309
4	128	16	6.0560	6.0643	0.0000	225	128	16	6.0560	6.0643	0.0000
5	256	8	6.2871	6.2954	0.0381	522	128	16	6.0560	6.0643	0.0000
6	127	16	6.1178	6.1271	0.0104	494	128	16	6.0560	6.0643	0.0000
7	255	8	6.3724	6.3828	0.0525	109	128	16	6.0560	6.0643	0.0000
8	135	15	6.2147	6.2237	0.0263	173	128	16	6.0560	6.0643	0.0000
9	128	16	6.0560	6.0643	0.0000	422	128	16	6.0560	6.0643	0.0000
10	256	8	6.2871	6.2954	0.0381	500	127	16	6.1178	6.1271	0.0104
11	127	16	6.1178	6.1271	0.0104	389	128	16	6.0560	6.0643	0.0000
12	128	16	6.0560	6.0643	0.0000	398	128	16	6.0560	6.0643	0.0000
13	64	32	6.3656	6.3739	0.0511	239	128	16	6.0560	6.0643	0.0000
14	146	14	6.3077	6.3172	0.0417	67	128	16	6.0560	6.0643	0.0000
15	128	16	6.0560	6.0643	0.0000	288	128	16	6.0560	6.0643	0.0000
16	128	16	6.0560	6.0643	0.0000	473	128	16	6.0560	6.0643	0.0000
17	156	13	6.3014	6.3104	0.0406	500	128	15	6.2429	6.2518	0.0309
18	227	9	6.3812	6.3904	0.0538	500	128	16	6.0560	6.0643	0.0000
19	136	15	6.2110	6.2199	0.0257	500	129	15	6.2387	6.2476	0.0302
20	128	16	6.0560	6.0643	0.0000	79	128	16	6.0560	6.0643	0.0000
21	136	15	6.2110	6.2199	0.0257	112	128	16	6.0560	6.0643	0.0000
22	128	16	6.0560	6.0643	0.0000	500	128	15	6.2429	6.2518	0.0309
23	128	16	6.0560	6.0643	0.0000	500	129	15	6.2387	6.2476	0.3022
24	128	15	6.2429	6.2518	0.0309	121	128	16	6.0560	6.0643	0.0000
25	227	9	6.3812	6.3904	0.0538	152	128	16	6.0560	6.0643	0.0000
26	128	16	6.0560	6.0643	0.0000	168	128	16	6.0560	6.0643	0.0000
27	63	32	6.4037	6.4125	0.0574	206	128	16	6.0560	6.0643	0.0000
28	119	16	6.2917	6.3011	0.0390	224	128	16	6.0560	6.0643	0.0000
29	136	15	6.2110	6.2199	0.0257	178	128	16	6.0560	6.0643	0.0000
30	145	14	6.3114	6.3210	0.0423	453	128	16	6.0560	6.0643	0.0000
	Average of $\frac{\lvert AT-OPT\rvert}{OPT}$=0.0248 Standard Deviation of $\frac{\lvert AT-OPT\rvert}{OPT}$=0.0212					Average of $\frac{\lvert AT-OPT\rvert}{OPT}$=0.0055 Standard Deviation of $\frac{\lvert AT-OPT\rvert}{OPT}$=0.0114					

Comment:
TT is the Theoretical Time for $\langle h_1, h_2\rangle$.
AT is the Actual Time for $\langle h_1, h_2\rangle$.
OPT is the Optimal Time for $\langle h_1, h_2\rangle$.
Optimal values for $\langle h_1, h_2\rangle$ via searching the entire space of feasible solutions is $\langle 128, 16\rangle$ with the actual time 6.0643

2D tiling with/without considering heterogeneity feature.

In experiment 1, the homogeneous computing system consists of eight similar nodes of type 1. So, the parallel execution time in 2D tiling with and without considering heterogeneity feature is the same and similarly for 3D tiling. In this case, the 3D tiling achieves 1.65× speedup of execution time compared to the 2D tiling.

In experiment 2, the heterogeneous computing system consists of eight processing nodes, four nodes of type 1 and four nodes of type 3, as mentioned in Table 4.2. Since nodes 1 and 3 have the computational power close to each other, the resulting speedup of execution time is almost close to experiment 1.

In experiment 3, the heterogeneous computing system consists of eight processing nodes, four nodes of type 1 and four nodes of type 8, as mentioned in Table 4.2. The nodes 1 and 8 have the computational power very different from each other. In this case, the 3D tiling achieves 1.74× speedup of execution time compared to the

TABLE 4.4
Near-optimal tile sizes of 2D and 3D tiling with/without considering heterogeneity feature

Exp	The sides of the tile	Without considering heterogeneity		With considering heterogeneity	
		2D Tiling	3D Tiling	2D Tiling	3D Tiling
Exp.1	V_i	{ 128,128,128,128,128,128,128,128 }		{ 128,128,128,128,128,128,128,128}	
	h_1	135	128	135	128
	h_2	1024	16	1024	16
Exp.2	V_i	{ 128,128,128,128,128,128,128,128 }		{ 163,163,163,163,93,93,93,93}	
	h_1	135	128	150	74
	h_2	1024	16	1024	32
Exp.3	V_i	{ 128, 128, 128, 128, 128, 128, 128, 128 }		{ 188, 188, 188, 188, 68, 68, 68, 68}	
	h_1	135	128	165	82
	h_2	1024	16	1024	25
Exp.4	V_i	{ 86,86,86,86,85,85,85,85,85,85,85,85 }		{ 122,122,122,122,89,89,89,89,45,45,45,45}	
	h_1	140	192	144	84
	h_2	1024	16	1024	38
Exp.5	V_i	{ 114,114,114,114,114,114,114,113,113 }		{ 161,148,144,142,122,105,94,54,54}	
	h_1	132	32	152	147
	h_2	1024	64	1024	18
Exp.6	V_i	{ 114,114,114,114,114,114,114,113,113 }		{ 152,152,152,152,152,66,66,66,66 }	
	h_1	132	32	156	128
	h_2	1024	64	1024	20
Exp.7	V_i	{ 114,114,114,114,114,114,114,113,113 }		{ 152,152,152,152,101,101,100,57,57 }	
	h_1	132	32	162	80
	h_2	1024	64	1024	32

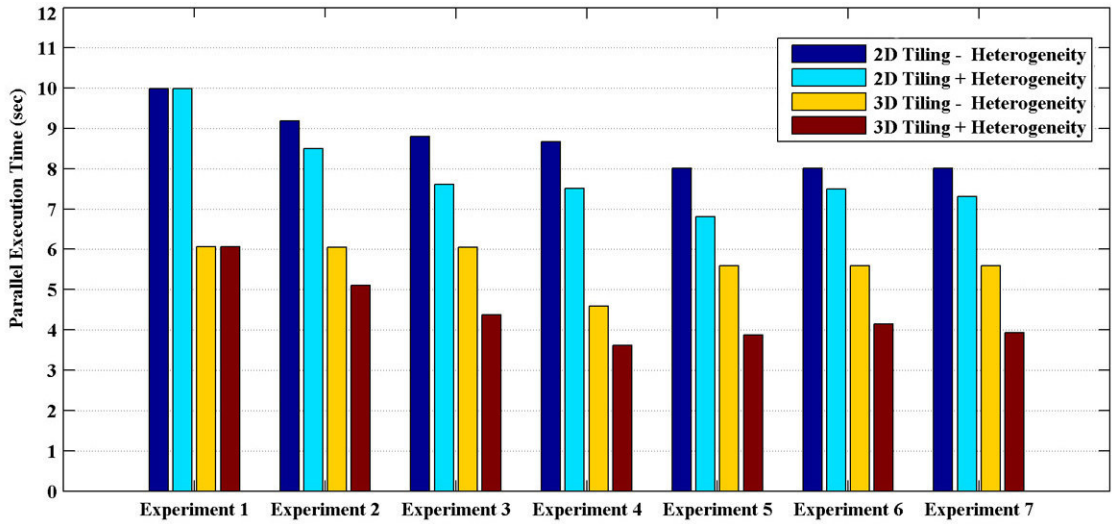


FIG. 4.3. *Comparison of 3D tiling and 2D tiling*

TABLE 4.5
The speedup of execution time of 3D tiling vs 2D tiling without considering heterogeneity feature

	2D Tiling-Heterogeneity						
	Exp.1	Exp.2	Exp.3	Exp.4	Exp.5	Exp.6	Exp.7
3D Tiling-Heterogeneity	1.65	1.52	1.45	1.89	1.43	1.43	1.43

TABLE 4.6
The speedup of execution time of 3D tiling vs 2D tiling with considering heterogeneity feature

	2D Tiling+Heterogeneity						
	Exp.1	Exp.2	Exp.3	Exp.4	Exp.5	Exp.6	Exp.7
3D Tiling+Heterogeneity	1.65	1.67	1.74	2.08	1.76	1.81	1.86

TABLE 4.7
Partitioning nodes in experiment 5 into two or three groups of similar performance

Computational power of nodes	Group 1	Group 2	Group 3
0.1568, 0.1438, 0.1405, 0.1383, 0.1197, 0.1029, 0.0920, 0.0531, 0.0529	0.1568, 0.1438, 0.1405, 0.1383, **0.1197**	0.1029, 0.0920, 0.0531, **0.0529**	-
	0.1568, 0.1438, 0.1405 **0.1383**	0.1197, 0.1029, **0.0920**	0.0531, **0.0529**

2D tiling with considering heterogeneity feature.

In experiment 4, the heterogeneous computing system consists of 12 processing nodes of three types 1, 4 and 7 as mentioned in Table 4.2. In this case, the 3D tiling can achieve 1.89× and 2.08× speedup of execution time compared to the 2D tiling without and with considering heterogeneity feature, respectively.

In experiment 5, the heterogeneous computing system consists of nine nodes of fully different computational powers as mentioned in Table 4.2. In this case, the 3D tiling can achieve 1.43× and 1.76× speedup of execution time compared to the 2D tiling without and with considering heterogeneity feature, respectively.

The heterogeneity is an important feature in parallel and distributed computing systems but considering fully heterogeneity in practice is very difficult. Therefore, we partition nodes of experiment 5 into two or three groups of almost similar performance in terms of their computational power and consider the weakest node in each group as the representative. Table 4.7 show the results of the grouping that was done with *fastclus* procedure on SAS software. The weakest node in each group is bold. The parallel execution times in experiment 6 and 7 are very close to experiment 5.

According to the experimental results, the parallel execution time of the 2D tiling and 3D tiling with considering heterogeneity feature is less than the 2D tiling and 3D tiling without considering heterogeneity feature. Therefore, loop tiling combined with the heterogeneity feature could help to improve the efficiency of computation on heterogeneous systems. Overall, the results show the minimum parallel execution time for the 3D tiling with considering heterogeneity feature in all experiments.

As already mentioned, Fig. 4.3 shows the cost to implement the obtained solution for 2D and 3D tiling. The proposed genetic tiling algorithm takes, on average, less than one second to find a solution. Therefore, the cost to obtain the solution for 3D tiling using the genetic algorithm is higher than 2D tiling, because it involves the cost of the evolutionary process. However, the results presented in Fig. 4.3 shows that the 3D tiling might lead to a more parsimonious solution in terms of implementation cost.

5. Conclusions and future work. This paper addresses the problem of 3D tiling and scheduling when parallelizing three-level perfectly nested loop with dependencies on heterogeneous systems. The tile size plays an important role to improve the parallel execution time of nested loops. Searching the entire feasible solution space of tile size can be very time consuming, especially in cases where the solution space is large. We build a theoretical model to estimate the parallel execution time with the computational power awareness of the nodes of computing systems. We use the proposed tiling genetic algorithm and nonlinear integer programming solvers, NOMAD, to find the near-optimal value of tile size from the theoretical model. Experiment results by 3D heat equation on heterogeneous systems show the accuracy and efficiency of the proposed theoretical model and the tiling genetic algorithm in estimating the parallel execution time and finding the near-optimal 3D tiling. Furthermore, we show that the 3D tiling combined with heterogeneity feature and a pipeline-like execution could exploit the potential parallelism and improve the parallel execution time of perfectly nested loop with dependencies on heterogeneous systems.

The plans for future work include: (i) extend the 3D tiling algorithm for the imperfectly nested loops with

dependencies on heterogeneous computing systems; and (ii) extend the 3D tiling algorithm to handle partially connected network.

Acknowledgments. The authors would like to thanks the editor and the reviewers for their helpful and constructive suggestions, which considerably improved the quality of the paper. They would also like to thanks Nasrin Nasrabadi and Fateme Karimi, PhD Students, for all very valuable comments.

REFERENCES

[1] S. Fide and S. Jenks, *A middleware approach for pipelining communications in clusters*, Cluster Computing, 10 (2007), pp. 409-424.
[2] I. Riakiotakis and P. Tsanakas, *Dynamic scheduling of nested loops with uniform dependencies in heterogeneous networks of workstations*, 8th International Symposium on Parallel Architectures, Algorithms and Network, ISPAN 2005, 2005.
[3] R. L. Cariño and I. Banicescu, *A load balancing tool for distributed parallel loops*, Cluster Computing, 8 (2005), pp. 313-321.
[4] X. Zhou, M. J. Garzarán, and D. A. Padua , *Optimal parallelogram selection for hierarchical tiling*, ACM Transactions on Architecture and Code Optimization, 11 (2015), pp. 1-23.
[5] M. I. Daoud and N. Kharma, *An efficient genetic algorithm for task scheduling in heterogeneous distributed computing systems*, IEEE Congress on Evolutionary Computation, CEC, pp. 3258-3265, 2006.
[6] G. Wang, Y. Wang, H. Liu, and H. Guo, *HSIP: A Novel Task Scheduling Algorithm for Heterogeneous Computing*, Scientific Programming, 2016 (2016), pp. 1-11.
[7] K. Qinma and H. He, *Honeybee mating optimization algorithm for task assignment in heterogeneous computing systems*, Intelligent Automation & Soft Computing, 19 (2013), pp. 69-84.
[8] C.-T. Yang and L.-H. Cheng, *Implementation of a performance-based loop scheduling on heterogeneous clusters*, Algorithms and Architectures for Parallel Processing, Springer, pp. 44-54, 2009.
[9] J. Dongarra and A. L. Lastovetsky, *High performance heterogeneous computing*, John Wiley & Sons, 2009.
[10] K. Hwang, J. Dongarra, and G. C. Fox, *Distributed and cloud computing: from parallel processing to the internet of thing*, Morgan Kaufmann, 2013.
[11] R. Bleuse, S. KedadSidhoum, F. Monna, G. Mounié, and D. Trystram, *Scheduling independent tasks on multicores with GPU accelerators*, Concurrency and Computation: Practice and Experience, 27 (2015), pp. 1625-1638.
[12] M. G. Lopez, J. Young, J. S. Meredith, P. C. Roth, M. Horton, and J. S. Vetter, *Examining recent many-core architectures and programming models using SHOC*, Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems, 2015.
[13] P. A. La Fratta and P. M. Kogge, *Heterogeneity in parallel and distributed computing*, Journal of Parallel and Distributed Computing, 73 (2013), pp. 1523-1524.
[14] M. E. Wolf and M. S. Lam, *A data locality optimizing algorithm*, ACM Sigplan Notices, pp. 30-44, 1991.
[15] G. Rivera and C.-W. Tseng, *Tiling optimizations for 3D scientific computations*, ACM/IEEE Conference in Supercomputing, pp. 32-32, 2000.
[16] M. E. Wolf and M. S. Lam, *A loop transformation theory and an algorithm to maximize parallelism*, IEEE Transactions on Parallel and Distributed Systems, 2(1991), pp. 452-471.
[17] D. Padua, *Encyclopedia of parallel computing* , Springer Science & Business Media, 2011.
[18] M. Kowarschik and C. Weib, *An overview of cache optimization techniques and cache-aware numerical algorithms*, Algorithms for Memory Hierarchies, LNCS 2625, Springer, pp. 213-232, 2003.
[19] S. Parsa and M. Hamzei, *Locality-Conscious Nested-Loops Parallelization*, ETRI Journal, 36 (2014), pp. 124-133.
[20] I. Riakiotakis, F. M. Ciorba, T. Andronikos, and G. Papakonstantinou, *Distributed dynamic load balancing for pipelined computations on heterogeneous systems*, Parallel Computing, 37 (2011), pp. 713-729.
[21] T. Andronikos, F. M. Ciorba, I. Riakiotakis, G. Papakonstantinou, and A. T. Chronopoulos, *Studying the impact of synchronization frequency on scheduling tasks with dependencies in heterogeneous systems*, Performance Evaluation, 67 (2010), pp. 1324-1339.
[22] U. Bondhugula, *Compiling affine loop nests for distributed-memory parallel architectures*, International Conference in High Performance Computing, Networking, Storage and Analysis (SC), pp. 1-12, 2013.
[23] H. El-Rewini and M. Abd-El-Barr, *Advanced computer architecture and parallel processing*, John Wiley & Sons, 2005.
[24] C. L. Abad, Y. Lu, and R. H. Campbell, *DARE: Adaptive data replication for efficient cluster scheduling*, International Conference on Cluster Computing (CLUSTER), IEEE, pp. 159-168, 2011.
[25] S. Parsa and S. Lotfi, *A new genetic algorithm for loop tiling*, The Journal of Supercomputing, 37 (2006), pp. 249-269.
[26] S. Mehta, G. Beeraka, and P.-C. Yew, *Tile size selection revisited*, ACM Transactions on Architecture and Code Optimization, 10 (2013).
[27] S. Chen and J. Xue, *Partitioning and scheduling loops on NOWs*, Computer Communications, 22 (1999), pp. 1017-1033.
[28] F. M. Ciorba, I. Riakiotakis, G. K. Papakonstantinou, T. Andronikos, and A. T. Chronopoulos, *Studying the impact of synchronization frequency on scheduling tasks with dependencies in heterogeneous systems*, PACT, 2007.
[29] J. Ramanujam and P. Sadayappan, *Nested loop tiling for distributed memory machines*, Proceedings of the Fifth Conference in Distributed Memory Computing, pp. 1088-1096, 1990.
[30] D. Liu, Y. Wang, Z. Shao, M. Guo, and J. Xue, *Optimally maximizing iteration-level loop parallelism*, IEEE Transactions on Parallel and Distributed Systems, 23(2012), pp. 564-572.

[31] F. M. Ciorba, I. Riakiotakis, T. Andronikos, G. Papakonstantinou, and A. T. Chronopoulos, *Enhancing self-scheduling algorithms via synchronization and weighting*, Journal of Parallel and Distributed Computing, 68 (2008), pp. 246-264.
[32] J. Xue, *Communication-minimal tiling of uniform dependence loops*, Journal of Parallel and Distributed Computing, 42 (1997), pp. 42-59.
[33] I. Riakiotakis, F. M. Ciorba, T. Andronikos, G. Papakonstantinou, and A. T. Chronopoulos, *Towards the optimal synchronization granularity for dynamic scheduling of pipelined computations on heterogeneous computing systems*, Concurrency and Computation: Practice and Experience, 24 (2012), pp. 2302-2327.
[34] P. Crandall , M. J. Quinn, *Three-Dimensional Grid Partitioning for Network Parallel Processing*, ACM Conference on Computer Science. Citeseer, pp. 210-217, 1994.
[35] O. Beaumont, V. Boudet, F. Rastello and Y. Robert, *Matrix multiplication on heterogeneous platforms*, IEEE Transactions on Parallel and Distributed Systems, 12 (2001), 1033-1051.
[36] E. Z. Zefreh, S. Lotfi, L. M. Khanli, and J. Karimpour, *3D data partitioning for three-level perfectly nested loops on heterogeneous distributed systems*, Concurrency and Computation: Practice and Experience, accepted, 2016.
[37] P. Boulet, J. Dongarra, F. Rastello, Y. Robert, and F. Vivien, *Algorithmic issues on heterogeneous computing platforms*, Parallel processing letters, 9 (1999), pp. 197-213.
[38] P. Boulet, J. Dongarra, Y. Robert, and F. Vivien, *Static tiling for heterogeneous computing platforms*, Parallel Computing, 25 (1999), pp. 547-568.
[39] F. M. Ciorba, T. Andronikos, I. Riakiotakis, A. T. Chronopoulos, and G. Papakonstantinou, *Dynamic multi phase scheduling for heterogeneous clusters*, 20th International in Parallel and Distributed Processing Symposium, IPDPS, 2006.
[40] F. M. Ciorba, I. Riakiotakis, T. Andronikos, A. T. Chronopoulos, and G. Papakonstantinou, *Optimal synchronization frequency for dynamic pipelined computations on heterogeneous systems*, International Conference on Cluster Computing, IEEE, pp. 410-415, 2007.
[41] F. Desprez, J. Dongarra, F. Rastello, and Y. Robert, *Determining the idle time of a tiling: new results*, International Conference on Parallel Architectures and Compilation Techniques, pp. 307-317, 1997.
[42] S. Le Digabel, *NOMAD: Nonlinear optimization with the MADS algorithm*, ACM Transactions on Mathematical Software (TOMS), 37 (2011).
[43] M. Gen and R. Cheng, *Genetic algorithms and engineering optimization*, John Wiley & Sons, 2000.
[44] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers*, CRC Press, 2010.
[45] A. Lastovetsky and R. Reddy, *Data partitioning with a functional performance model of heterogeneous processors*, International Journal of High Performance Computing Applications, 21 (2007), pp. 76-90.

Edited by: Dana Petcu
Received: May 28, 2016
Accepted: August 2, 2016

DOI 10.12694/scpe.v17i4.1206
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 4, pp. 351–368. http://www.scpe.org

A SELF-HEALING ARCHITECTURE BASED ON RAINBOW FOR INDUSTRIAL USAGE

ALI FARAHANI*, ESLAM NAZEMI* AND GIACOMO CABRI†

Abstract. Over recent decades computer and software systems become more and more complex because of the applications' and user's requirements. The complexity makes the software systems more vulnerable to the error and bugs. Also, environmental situations affect software systems which do not react to the environmental activities. Self-healing architectures have been proposed in order to make systems defeat these problems and to make systems capable of reacting to the environmental activity. Hence, these architectures help system to become dynamic and more robust, but finding a proper architecture which can support and cover system's requirements is an issue. This is particularly true in industrial environments, which consist of some known and some unknown parameters.

This paper presents an architecture that can be used in some industrial environment to facilitate the process of adapting the system to unpredicted situations. This architecture has been developed over the base of RAINBOW infrastructure and it is compliant to the MAPE control loop (*Autonomic Computing* control loop). The paper reports also about the practical experience of implementing this architecture for a painter robot in an automotive factory, which deals with problems in painted part by itself. The proposed architecture uses rule-based reasoning and it actualizes the method of environmental modeling by using a rule-based system as the model extractor. The results of the implementation shows huge benefits in reusability and even in the quality of painting process.

Key words: Self-healing, RAINBOW infrastructure, Rule-Based

AMS subject classifications. 68M14, 68N19

1. Introduction. Computer systems play an unavoidable role in every field. In this paper, we focus on industrial environments, where they are widely exploited. Industrial environments have a unique property which makes it different from other environments: they have so many parameters and the value of some of them could not even be defined. Dealing with this kind of situations needs a software system that can manage the unpredicted situations. The main goal of the research experience presented in this paper is to propose an architecture and its implementation for industrial applications, which is able to adapt to environmental unpredicted situations by means of the self-healing property.

As mentioned in [1], to face the size and complexity of software systems, design has become more important than algorithms and data structures. In this context, defining an architecture is necessary for developing software systems. Another issue which should be considered is addressing failures [2]. Fault management can be considered and categorized with following reasons and cases:

- The scope of the systems could change during the process and also changes in system's specification will occur because of early system's requirements analyzing.
- Because of the system's environment characteristics, the system could face changes when executing in a non deterministic environment.
- Failures in system's part could occur.

Systems having unknown, incorrect and/or improvable behavior are not used by any developer and user of computer systems, because of their unreliability in covering the users' needs [2].

Starting from these considerations, our aim is to propose an architecture to react to failures in an industrial system. Previous study in fault management considered fault management as gathering and analyzing alerts and failures in a service [3]. According to the explanation in [3], "service" means *tasks* and *functions* presented by an industrial IT system, although this definition can be considered for the usage of any other software systems as well.

The main steps in fault management process are described as below [4]:

- Detect
- Diagnose
- Decide

*Computer Science and Engineering Faculty, Shahid Beheshti University, Tehran, Iran ({a_farahani,nazemi}@sbu.ac.ir)
†Department of Physics, Informatics and Mathematics, Università di Modena e Reggio Emilia Modena, Italy (giacomo.cabri@unimore.it)

- Respond

Beside this viewpoint for remediating the systems from fault state, there are several solutions for fault management and responding to failure in the software systems context. Similar to [3], we are looking for the capability of adapting the system to the environment. For a long time, self-adaptation property's implementations were used in computer systems and it was blended with other system functionalities. In fact, a part of the system is usually responsible for responding to events (e.g. failure) but not separately from the part responsible for the main system functions. As also proposed in software engineering, it is better to have separation of concerns. For supporting this separation, it has to be applied in both system design and implementation. Separation is mentioned as one of the views in [6]. If the change and fault management functionality domain remains at the code level (high coupling), the domain of changes detection and responding to changes will be small and inadequate, introducing another weakness.

The solution and approach that can be conceived to face these problems, also emphasized in [7], is to address the response to changes and failures at the architecture level. In another word, the architecture of the system should be designed for reacting against failures and changes.

In the case of lack of an architecture, the developed system will no longer carry the benefits of an architectural approach to systems development. Some of these benefits are [8]:

- *reusability*: reusing developed modules;
- *easy improvement*: continuous and simple improvement of each section;
- *extendibility*: simple extension capability;
- *changeability*: ability to make simple changes to different parts of the system.

Various architectures have been presented for supporting the reacting to changes. One of these architectures is derived from *Autonomic Computing* (AC), which is described in [9, 10]. Also, architectural solutions for fault-management are used for confronting and reacting against failures. Examples of these researches are mentioned in [11]. The main feature of the architecture in [11] is that it supports responding to changes at the architecture level. Also, *self-healing* is one of the properties and capabilities supported by the architecture presented in [11]. In [11] the task for self-healing is to recover the system from errors and responding to failures. Self-healing is defined as "To discover, diagnose and react to disruptions" in [10]. As it is clear, the viewpoint of this feature is aligned with the *fault-management* approach and both seek to react to events that take the system out of a correct state. By having in mind the researches mentioned in this section, it could be an option to consider self-healing and fault-management as two features that have similar goals and they are trying to reach their goals with different views and paths.

For achieving a system capable of managing and responding to changes and failures, self-healing should be considered as an architectural aspect is systems architecture [5]. In order to implement an autonomic architecture for self-healing a *model of failure* must exist [7]. So there will be a knowledge about states in which failures can happen in the system. Also, there should be the information about which state requires responding and which does not. Due to the differences between architecture and implementation abstraction level, a solution is needed for realizing architecture level views in the implementation. Thus, an architecture should be implemented which supports the AC at the architectural level.

Another research field in AC which can have benefit in fault management is policy-based autonomic systems, which are mentioned as a solution for making systems react to the changes [12]. For implementing a policy-based autonomic system, there is an approach that uses a rule-based viewpoint to make a rule-based engine for systems' reaction task [13]. The rule-based approach is mentioned as a type of policy-based system implementation and also rule-based reasoning as a solution for defining policies [14]. Humans are also reasoning in the way rule-based systems reason. This could facilitate the process of using the self-healing systems (rule-based self-healing systems) for humans (system's user).

The contribution of this paper is to present a *self-healing architecture* in the industrial environment, which takes inspiration from the fault management view and based on Autonomic Computing. In this architecture, the reason of using rule-based approaches is to achieve a more specific architecture definition and also to facilitate the process of human dealing with systems specification and implementation.

There are some general proposed architectures that can be found in self-healing software, self-healing systems and autonomic systems research fields. The proposed architectures have been used to facilitate the adaptation

of a robot's software system to its new task. The architecture presented in this paper does not rely on any specific platforms or implementation's language. This architecture can be used in a wide range of software systems.

Another contribution of this research is that the prosed architecture is integrated for all phases in the software development (from designing a general architecture to its implementation). Namely, it could cover the process from the high-level architecture development step (like AC architecture) to deciding about the implementation solution (like policy-based system) and implementation algorithm (like rule-based implementation).

By presenting the literature and previous work in Section 2, we become familiar with achievements in this field and we can define our architecture. Section 3 introduces a real-life case study that has been used for explaining and also examining the architecture implementation. In Section 4 the proposed architecture is presented. In Section 5 the results of implementation and examination is reported and Section 6 discusses the conclusions and further work.

2. Related Work. The previous researches which are related to architectures of reacting systems in the industrial usage (based on the knowledge from software engineering, Autonomic Computing and fault-tolerant) will be presented in different three categories:

- Autonomic Computing and self-healing systems;
- Fault-management with self-healing viewpoint;
- RAINBOW architecture [1] as a base for proposed architecture.

According to the literature, there are papers including the combination of more than one of these categories which will be discussed in detail in the following.

2.1. Autonomic Computing and Self-healing Systems. In [21], entitled *Towards architecture-based self-healing systems*, the goal is presenting an architecture with a self-healing capability in order to execute `repair system` in running time and without human interference. In order to reach this goal, different concepts and tools are used which are mentioned in the following:

- Elements of this architecture are formed by components and connectors between them; this makes performing fault management changes in the architecture very flexible, because of the loosely coupled connection point feature.
- Connections between these components are established through "independent messages or events" instead of "shared memory between components". Moreover, benefiting from the events will provide the possibility to separate components from each other and eventually it will provide the system with the ability to remove, add or replace the components easily in run time without changing the code.
- In order to obtain a vast range of applications supporting specific types of repairs, domains, or implementation platforms, middleware, and languages, it is necessary to use a general architecture description language (such as xADL 2.0 in this work).

The software architecture description is the basis of the system implementation, in other words, the components are located and loaded (also generating of links and connector) based on implementation knowledge contained in the architecture description. In the situation that a fault happened and there is a need to perform some reactions, there is an engine that takes the description of two xADL 2.0 architecture described (the first is the current architecture, and the second is the architecture proposed as a remediation) as input and tries to extract the difference between them and gives a new architecture description as an output. In another part, this new architecture, which is derived from the differences, will be analyzed. After deciding that is a good solution for the system, it can be executed.

From these explanations, it is obvious that infrastructures that do not support adding or removing components in a software system are unsuitable for this approach. Also, we know that although this policy is not sufficient for all types of repair, it is suitable for many applications, like those with no strict timing constraints. But considering software architecture descriptions as integral parts of the deployed software system described by them is an incomparable aspect of the approach given here.

As it is obvious, discussed research in [21] is carried out at the architecture level and it tries to support healing by changing the architecture (without interfering the change ability inside the system implementation). Also, the detail level remains at the architecture level. As mentioned, this solution is not applicable in a wide range of systems.

Being the change in management part outside the system implementation (except for changes in relationships and architectural order), the ability to change the implementation will be taken away from the system. This point questions the self-healing capability.

In [22], easing the self-healing in software based on software control principle is discussed. It focuses on remediation of failures in software based on reacting to the failure with the help of non-internal component of the system. It prepares Final State Machine for software and inserts remediation states into the software execution flow. This research adds some states in order to control the failure in the system. This research does not interfere in the main system architecture and just tries to remediate failure with adding some remediation state. By the way, interfering the architecture could improve the result of remediation but it has more difficulties.

Another research brings architectural patterns to support self-adaptation in architectural level (SimSOTA) [23]. **SimSOTA** is an integrated Eclipse plugin that brings self-adaptation into the system based on a feedback loop. It uses model-driven viewpoint and implemented by a case study in the cooperative electric vehicle. It does not deal with software architecture in requirement analysis or design phase. Need for a reference architecture for easier implementation of the self-adaptive system in industrial usage is not covered by this research.

2.2. Fault-management with Self-healing. As mentioned in [16], autonomic fault management can be done with the usage of IBM's reference architecture for AC. This architecture is mentioned in the previous section. This article tries to show this that in order to create an automatic fault management system it is necessary that the knowledge capable of reasoning exists in all steps from detection to response to the fault.

This research in [16] provides an engineering process in service level for fault management. But this technique only applies to the service oriented systems. Also, there is no detail on the system architecture and how components are placed in the system and the internal system architecture in [16]. In another research [20], an investigation is done for solutions of self-healing in software systems. Here it is indicated that one of the solutions in this field is obtaining knowledge from the environment based on the model. Meaning that, the method presented in this paper for identifying and understanding the environment is carried out by creating and processing the model using the environment and its conditions. But in this research details and specifications about the system architecture and also the implementation and the solution to fulfill it, are not given. As it will be mentioned in future papers of this research, a possible alternative could be investigating first for more intelligent repair policy mechanisms. It means what the structure basics and building blocks of the self-healing capability implementation are.

Also in a fault management viewpoint, a structure for fault-management systems is presented in a mentioned architecture and commercial used in [20]. This structure is used for fault management systems in network management systems. It includes a few elements provided generally in all systems; `Modeling event/alerts`, `Parsing`, `Correlation`, `Validation rules & Filters`, `Fault DB`.

These sections' function provides input to enter the `parse` procedure. In the `parse` and its internals information is extracted (for example information are placed in a structure like XML and need parsing for recovery). Now this information is correlated and handed over to `Modeling event/alert` in order to draw the current system condition model. This model is checked considering all rules and filters and its required decision is extracted and the decision is executed by `Action`. This isolation that which models require a response and which do not be stored in fault database (DB). Information about conditions which must be assumed as a fault by the system is stored in this database.

The architecture in [20] was presented with enough information about the details of implementation of each components and internal modules. However, it does not mention how the knowledge is maintained and checked the models based on knowledge. Also, this architecture is designed for a specific area of systems (network management). This architecture requires implementing conditions and environment policies.

2.3. RAINBOW Architecture. In [15], RAINBOW as a framework/infrastructure is presented including a structure capable of being reused and also solutions for applying it. RAINBOW infrastructure gives the executor the capability of explaining necessary motivations and performing the suitable instructions. On the other hand, this presented structure in [15] generates the ability for reusing solutions and methods (codes and implemented system) which are prepared in the first place.

This generation is formed by two main modules, *adaptation infrastructure* and *system layer*. The system layer indicates a running system (without change capability) to which self-adaptive property must be added.

This task which will be inserted into the system will be done by adaptation infrastructure. RAINBOW framework also includes translation and architecture parts in infrastructure module. The translation module is used for transforming the structure of the information that comes from the environment into the information which can be understood by another part of self-healing architecture. This transformation of information is the reason of RAINBOW framework reusability feature. In architecture module, there are four main elements; Model manager, constraint evaluator, adaptation engine and adaptation executor. The required knowledge for changing the system is known as system specific knowledge; Using this knowledge and also defining mappings, types and properties, rules, strategies and tactics and operators will make the system capable of responding to the case[15]. This information and data are necessary to this system and could be considered as knowledge. In RAINBOW architecture, after detecting information and conditions according to resources by the probe, they are reported to the architecture layer by knowledge and resource discoveries and in between; transforming this information into comprehensible information for the architecture layer is carried out using existing rules in mapping module. Then, information is aggregated by gauges and is transformed to the environment model by the model manager. Next, limitations are checked by the analyzer and if necessary, it will send an adaptation request to the adaptation manager and after explaining the adaptation strategy, adaptation manager will provide the strategy to the change executer. The applied result to the system layer will be translated into the system level by the translator and then it will be sent.

Clearly, in RAINBOW framework few features are emphasized:

- Reusability: having the ability to use some implemented part in future;
- Specified architecture: defining an architecture and roadmap for creating a system;
- Based on a reference architecture: being based on a well-known control loop (MAPE loop);
- General propose: not being domain specific.

But besides these features, there could be different ways to implement the knowledge detection method and also knowledge levels, system details and controlling details and the system implementation. These kinds of information about implementation of RAINBOW architecture/infrastructure have been introduced in [17, 18].

The architecture presented in [17, 18], considers AC and self-adaptation generally, and could be specialized for special-purpose functions and attributes, like self-healing in order to gain more performance and reusability in that domains.

Similar to RAINBOW, there is a research [19] that also takes decisions on the base of known probability of a failure. It provides failure avoidance based on changing systems component (services) based on scenarios and situations. The cost of each change in system and failure rate of each service is calculated for each scenario and adaptation plan.

According to these researches and our aim of having a self-healing architecture for industrial usage, the following issues must be considered in our work:

- The RAINBOW framework includes reusability and also other capabilities for having an architecture for a system and we can guarantee these abilities by underlying on it. Furthermore, this framework is based on the AC presented by IBM and this point is an acknowledgment to the performance and the verification of this framework and adaptive architectures based on this framework.
- In order to obtain a usable architecture as mentioned in [15] the solution for implementing the architecture must be considered in the architecture itself.
- As given in [9], including knowledge about the environment and the system also knowledge on adaptation in architecture is necessary. In addition, as in [16], reasoning from the knowledge base on the situation should be considered, and also how the knowledge is implemented must be provided as other elements of the architecture.

Considering these aspects, in the next section we introduce a real case study and then we will present our architecture and prove this architecture by referring to previous works.

3. Case Study. To explain our approaches we introduce a real case study, which is taken from an industrial project about using a robot for painting automotive parts. This robot had been programmed to paint a specific automotive part. The software consists of a program P that was written in C# language. The program P can run a software code G that was written in G-code; the execution happens on the robot through an API. G-code is also known as RS-274, which is the common name for numerical code (NC) programming language. The

main usage of the G-code is in the industries that use computer-aided robots. An example of a G-code for our robot and its description from starting the painting process is the following (code descriptions will follow '#'):

```
# version
3
# title
first side buck
# Start Delay
# 36.5
60
# VerticalOffset
0
#iteration delay
6
#GUN Mult
0.75
#GEAR Mult
1.0

COMMAND
# delay action [data]
# delay can be a double number represent an absolute delay passed from start delay
# delay can be as this format: $index,offset == index*iterationDelay + offset
# examples :
# 0.1 HOMINGHEAD
# $1,0.5 MOVE 1 0.5 0.5
#
# delay MOVE vertical gear gun
# 0.2   MOVE 10.3     0.5  0.5
#
# delay COLOR vertical gear gun sprayStartDelay sprayDuration
# 0.5   COLOR 20.5     0.5  0.5 0.1             1.8
#
# delay COLORPOS vertical gear gun

-2.5 MOVE  -3  0.15 0.55

$0,0 COLOR  120   0.15     0.40      0.200 1.600
&0.1 COLOR  -3    0.16    0.63      0.050 1.700
&3.8 COLOR  120   0.15     0.30      0.000 1.300
&0.5 HOMINGHEAD
```

Fig. 3.1 shows the robot. The robot consists of 5 main parts:

1. the *body* which the hand is installed on;
2. the *hand* which carries the wrist;
3. the *wrist* which controls the nozzle;
4. the *nozzle*;
5. the *camera* which can capture the state of the painting position on the automotive part.

The body can lift up and put down the hand which carries the wrist. The wrist is the most important part. It has 2-dimensional freedom degree for changing the position of the nozzle (this grants the robot 3 type of different movements) and of the camera, which takes the picture from the painting position on the automotive part.

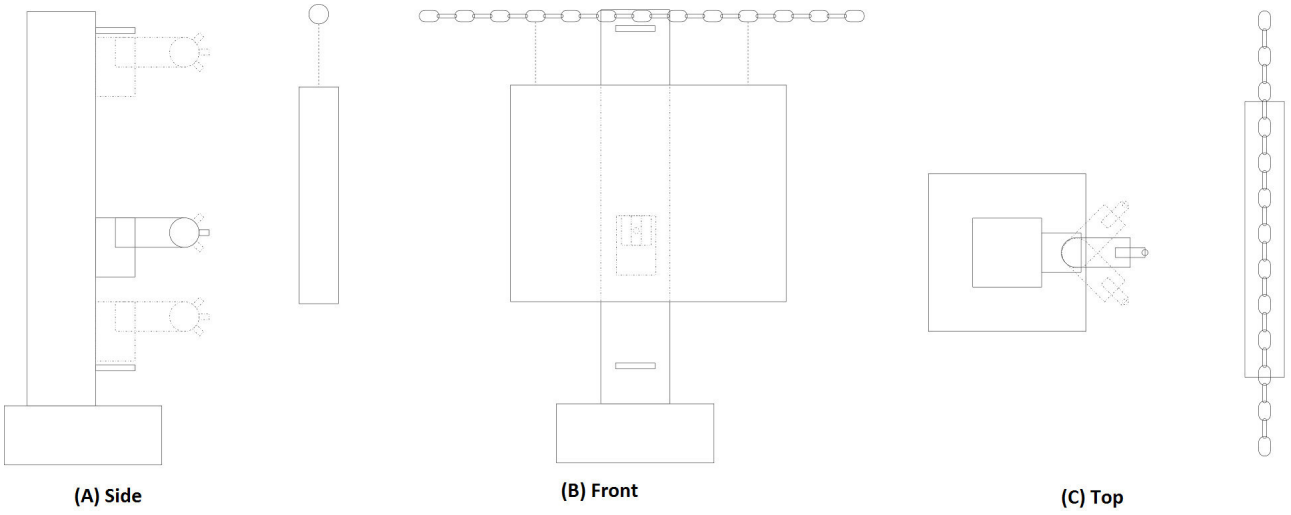


FIG. 3.1. *Painter Robot*

The automotive part to be painted is chained into the conveyor and comes from one side and goes to another side. The speed of the conveyors could be considered slow. The actual speed in the factory is 20 cm per minute. The robot control code (G) was written based on a best practice, which states that the painting is better to be in not all time slots. This means that, for example, it is better to start painting in t=0s and in t=10s the painting stop. After 30 seconds of pause, another 10-second painting slot is started.

Stops between each painting slot make the robot capable of remedying the painting process that has had faults: the corrections are provided in between two painting slots. To remedy the painting processes there must be a feedback from the painting process so the remediation program for those slots is going to be prepared. This process needs some heuristics from humans that are familiar with G-codes and know the object to be painted and the environment and also know about constraints in the painting process. For this purpose, the painting of an object ran so many times and the images had been processed by computer for colorless spots. The experts introduce some remediation in the code G for each situation that shows that the system did not do its job well enough after analyzing the situation (for example a problem on the color density of some automotive parts).

The main problem is that changing this G-codes for programming and remediation is a heavyweight process and task for software developers and will cost so much resource for the project. Any kind of enhancement in this area could be considered as a huge benefit for the company.

This real case study is going to be used to present the proposed architecture.

4. Proposed Architecture. In this section, a self-healing architecture is proposed to address fault management introduced in the previous sections, by means of self-healing. This architecture passed through two phases of improvement. The first phase is going to reduce the time for changing the software for different situations and tasks; the second one is to even make it easier to change and also more general to use.

We did not start our work from scratch, instead we take inspiration from the RAINBOW architecture previously presented. However, some issues drove us to propose a *new* architecture based on RAINBOW; the main needs can be summarized in the following reasons:

- No need for online feedback: differently from the RAINBOW architecture which supports online feedback, in our architecture we do not need online feedback because of the environment and problems nature. In industrial usage, not all the factors that are going to influence the process are known and this will make the online feedback not only unnecessary but also misleading for the context. Knowing the cause of the fault in nondeterministic environment (such as industrial usage) is impossible (even implicit cause) and grabbing the result, analyzing the data and importing it into the knowledge could be a bad idea.

- Nondeterministic environment: there will be so many unknown situations and parameters in industrial usage. The example in this paper is one of them. In this kind of environment, presenting the probability function for each action and also knowing the result set o the environmental variable is not doable. So, simplifying the architecture will help the modeling of the problem.

The following architecture will address the industrial problem easier with less complexity and in an enough complete way rather than the RAINBOW architectural-based solution.

Starting from the existing implementation, we report the improvement we applied to it in two phases, detailed in the next subsections.

4.1. Phase 1. As mentioned before, we aim at proposing an architecture for software systems that can heal themselves against faults and keep achieving their tasks. Considering the case study, an example of the fault is when a small portion of automotive parts has some areas which have oil drop on it so there will be the need for repainting those areas. The robot should recognize these situations and perform a sequence of tasks in order to repaint the unpainted areas. This remediation is going to be the self-healing part for the first phase of the proposed architecture. Besides the functional requirements as the example mentioned above, below we mention some non-functional requirements that have led our work:

- Being general-purpose. Meaning that it is not developed for a specific application.
- It must be developed based on reference architectures or best practice; this will help the architecture and so the system to have the benefits of those best practice or reference architecture in them.
- User of this proposed architecture must be supported in all phases of system development, from strategy to design. Meaning that when creating a macro design, detailed designing, and implementation methods must be considered.
- Bringing the benefits of the architecture for the software system (for instance, reusability).

The first idea that comes to mind about proposing the architecture is to go with best practice and well-known architecture. There are so many works that adopt the feedback loop, but the general purpose and the most well-known reference architecture is the Monitor-Analyze-Plan-Execute (MAPE-K) loop which was introduced by IBM [9]. This control loop is known thanks to its properties: it is a general purpose architecture for supporting all the AC properties (like self-adaptation, self-configuration and so on), it also supports self-healing as one of the four main self-managing properties.

There are some works that introduce an architecture or extension of architecture over the MAPE-K loop, the most well-known is the RAINBOW architecture [1]. It actualizes the MAPE-K loop with a model-based viewpoint.

As presented in the Section 2, the RAINBOW architecture can support the capabilities and advantages of using an architectural approach for the developed system. Also, this architecture is based on the AC architecture reference model (MAPE-K loop) presented by IBM in [9]. Hence, it can guarantee the accuracy and solidity of the proposed architecture based on this framework. Our architecture also inherits the mentioned characteristics. Our architecture is based on the RAINBOW framework and adopts the four main phases in the mentioned framework. We remark that the fact that our architecture is based on RAINBOW can grant reusability to the elements of the developed system based on our architecture. In addition, our architecture is based on the AC reference architecture.

Fig. 4.1 shows the proposed conceptual architecture. In this architecture, we have four main components (*Model Manager*, *Model Analyzer*, *Self-healing Engine* and *Executor*) similar to the RAINBOW and IBM's MAPE loop. A knowledge-base component supports these four components and all the data that are supposed to be transferred in or out of the system should pass through the *Environment Adaptor*, which enables the interactions between the environment and the self-adaptation parts of the system.

An aspect of the RAINBOW framework that must be considered is the needed knowledge and information for reacting to the environment. In the RAINBOW framework, this information is seen as part of the architecture puzzle which is added to each part when needed. It means that each module of the architecture somehow maintains and uses its specific knowledge and attributes. This technique in RAINBOW is different from the MAPE-K IBM reference architecture, where an integrated database is used for maintaining knowledge of all modules. In our architecture, we spent an effort in committing to the information and knowledge described in each RAINBOW framework module, but in order to simplify the implementation further and also to use a

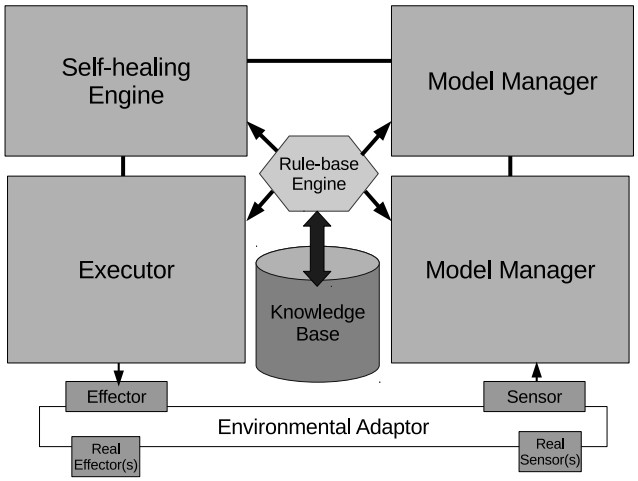


FIG. 4.1. *Self-healing architecture (Abstract view)*

knowledge structure for the all of the system's component, we use a centralized database in our architecture. Using the same knowledge structure throughout the whole architecture can help us in implementing the system easier and it will not carry on different and sometimes inconsistent methods for storing, recovering and using knowledge.

The mentioned knowledge component includes a reasoning capability for managing the knowledge at the run time. Decision making based on this knowledge needs knowledge management capabilities. Hence, the mentioned knowledge component must be able to inference. The existing method for storing, recovering and using the knowledge also having the ability of inference is called "rule-based knowledge" and the "rule-based knowledge base" in [24]. So, in the proposed architecture the knowledge is maintained in a centralized manner in a rule-based type knowledge base and inferences are done based on this type of knowledge.

After the definition of this conceptual architecture, the program P for running the painter robot went through a refactoring process and the new project can count on six main packages:

- `ModelExtractor`: represents Model Manager in the self-healing architecture in the refactored code.
- `ModelAnalyzer`: represents Model Analyzer in the self-healing architecture in the refactored code.
- `SelfHealingEngine`: represents Self-healing Engine in the self-healing architecture in the refactored code.
- `Executor`: represents Self-healing Engine in the self-healing architecture in the refactored code.
- `Knowledge`: represents Knowledge Engine in the self-healing architecture in the refactored code.
- `Main System`: represents Environment which self-healing Architecture deals with.

As mentioned in the packages' presentation, these packages are derived from the component in the proposed architecture (Figure 4.1). Also, having `Model Extractor`, `Environmental Adapter`, and `Self-healing Engine` in the proposed architecture are similar to `monitoring`, `interpretation`, `resolution` and `adaptation` which are mentioned in [25]. This similarity could be considered as compliance of the proposed architecture with previous works.

In the following, we are going to explain the classes and their relationships.

4.1.1. Model Extractor. This package extracts some models from the environmental situations and reports them to next packages. This package consists of the following classes:

- `PainterSensor`: This class is an implementation of a `Sensor` interfaces. It senses the environment (through the `Main System`) and extracts data from robot position, robot occupancy, and automotive part position. Also, it uses `CameraImageProcessing` for knowing about the unpainted areas of automotive parts.
- `CameraImageProcessing`: This class processes the images from the area of painting and find out about

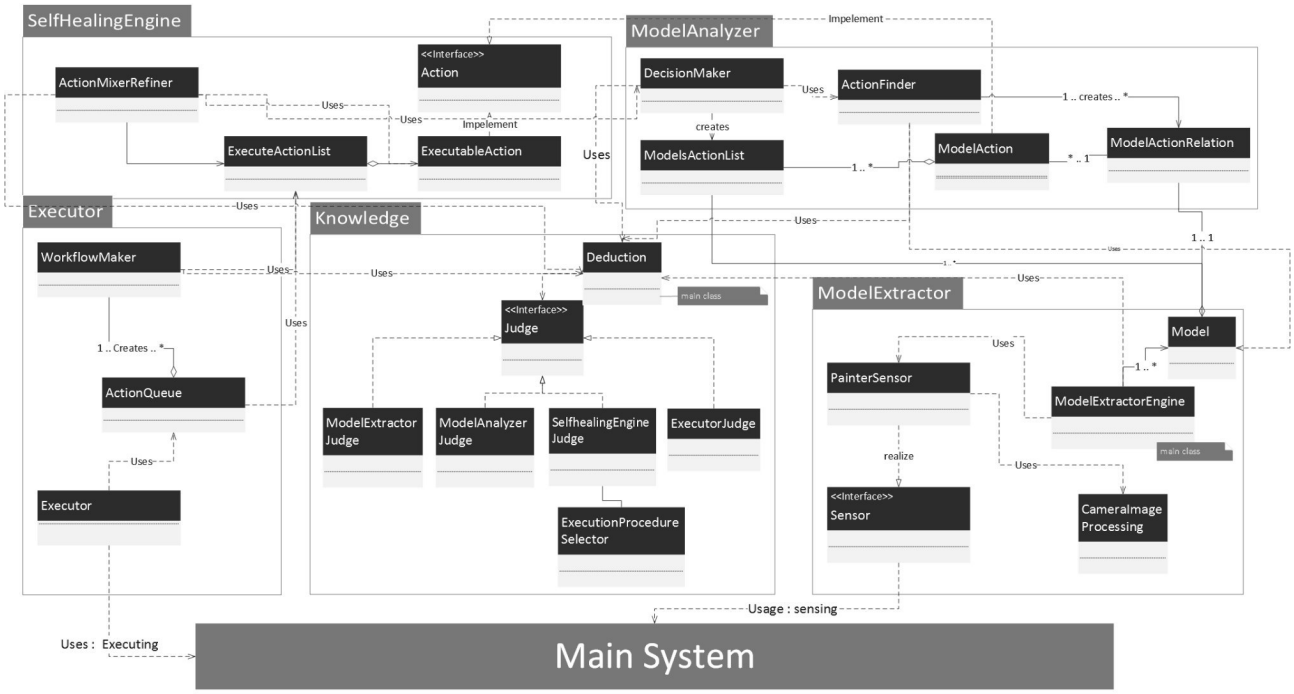


FIG. 4.2. *Class diagram of Phase1's system implementation*

the unpainted areas in the automotive parts.

- `Model`: This class represents the object model in the system. It contains information about the situation which is detected. For example, recognizing the spot with bad painting could be a model.
- `ModelExtractorEngine`: This class based on the information from a sensor and also knowledge from `Knowledge` package, concludes about the models that can fully or even partly represent the current environment and robot states.

4.1.2. Model Analyzer. This package analyzes the situations of the environment and the robot. Suitable actions for each model are the results of the model analyzing the process. The classes in this package are:

- `ModelActionList`: This class represents the list of actions that are concluded for the extracted models or situations.
- `ModelAction`: It represents the action which could be mapped into a model. For example spraying color from above can be an action.
- `ModelActionRelation`: This class represents the relationship between `ModelAction` and `Model`.
- `ActionFinder`: This class creates instances of `ModelActionRelation` between each `ModelAction`'s instances and `Model`'s instances.
- `DecisionMaker`: This class decides about what action should be performed for each model. This class uses the knowledge from the `Knowledge` package. For example, `DecisionMaker` takes decision about repainting the area changing the nozzle angle.

4.1.3. Self-healing Engine. This package will analyze and combine (if needed) the actions for reacting to models. This package will consider the impact and the influence of each action on the robot and environment for combining and scheduling the actions. Also, refinement for actions is applied for action to prepare the applicable and understandable actions for the robot.

- `ExecutableAction`: This class implements the `Action` interface. It represents the action to be executed. For example, changing the nozzle angle in X axle of +22 degrees is an executable action.
- `ExecutableActionList`: This class represents the list of `ExtecutableAction`. For example, a list

consist of these 3 actions are a ExecutableAction list: 1) change the nozzle angle in X axle to +22 degree, 2) spray color for two seconds and 3) change the nozzle angle to 0 degree.

- `ActionMixerRefiner`: Some actions are going to be mixed based on their impact and influence and these actions also should be refined into some fine-grain actions. These tasks are implemented by an `ActionMixerRefiner` class's instance. For example, changing the nozzle direction in X axle of +22 degrees and after that reversing it and after that changing the nozzle direction in X axle for +10 degrees and reversing it could be optimized: first changing direction to +10 degrees and after that +12 degrees more and returning the nozzle to +0 in X-axle.

4.1.4. Executor. This package provides a doable action list and workflow and sends it to the executor for applying the actions to the environment (`MainSystem`). It contains the following classes:

- `Executor`: Instance of this class will execute all the actions (from the `ActionQueue`'s instance) on the environment (`Main System`). This class contains the business logic for executing the actions.
- `WorkflowMaker`: Maybe some actions and some tasks need some prerequisite or maybe it is better to perform them in a specific order. This class will take care of ordering and timing of actions. This class uses the `Knowledge` package and also the `DecisionMaker` class to carry out its tasks. Making a set of changing of robot's position with better efficiency is a goal for this class.
- `ActionQueue`: This queue contains the tasks and actions that can be enacted in the environment.

4.1.5. Knowledge. This package responds to the request (questions and queries) of other packages (actually of their classes). It responds to the queries based on the knowledge that is hard-coded in it. The classes are:

- `Judge`: This interface is implemented by each of the four main components of the control loop (Model Extractor, Model Analyzer, Self-healing Engine and Executor). These classes will answer to the `Deduction`'s instance in order to make it able of deducting about situations and models.
- `Deduction`: This class will answer to the query about situations and models which come from any of four main components of the control loop.

4.1.6. Main System. This package contains the rest of system, which works and paint automotive parts based on its simple work-chain. The other packages see this package as `Environment` and `Main system`.

The improvements applied in this first phase reduced the time for preparing the robot for another automotive part painting. Results are discussed in *Implementation and Experimentation* section.

4.2. Phase 2. The proposed architecture is abstract and conceptual; in order to make it more concrete, each of the architecture elements will be discussed with more details. This explanation should be based on four main RAINBOW framework phases and the IBM reference architecture. These internal elements should be general purpose and not only applicable to one solution. So these elements differ from the packages and classes in the 4.1.

Because the implemented system (which is implemented based on presented abstract architecture in phase 1) and the detailed architecture (which is presented in phase 2) derive from one source, the implemented system in Section 4.1 should be able to fit into the detailed architecture. Based on the related work (RAINBOW, self-healing architecture, fault-tolerant system architecture, etc.) elements in the proposed architecture will be explained with more detailed. Because of that, our architecture must tend to a self-healing architecture based on fault managing. Hence we have used pure fault management architecture, derived elements and details are adapted to our problem's context.

According to these descriptions, we detail our architecture elements. For a better explanation, we are going to continue using the example as a tool for description and clarification of the research.

The *detailed architecture* could be found in Fig. 4.3. The description and specification of the components are presented in the following subsections , referring to the case study.

4.3. Model Manager. In this phase, input information from the environment is transformed into a model of the current environment situation using the related knowledge about the system state and input data. It means that the raw input data is separated, verified and aggregated. The information is given to the *Rule-Based Situation Monitor* module based on the rules. Situations resulted from the rules are transformed into a model

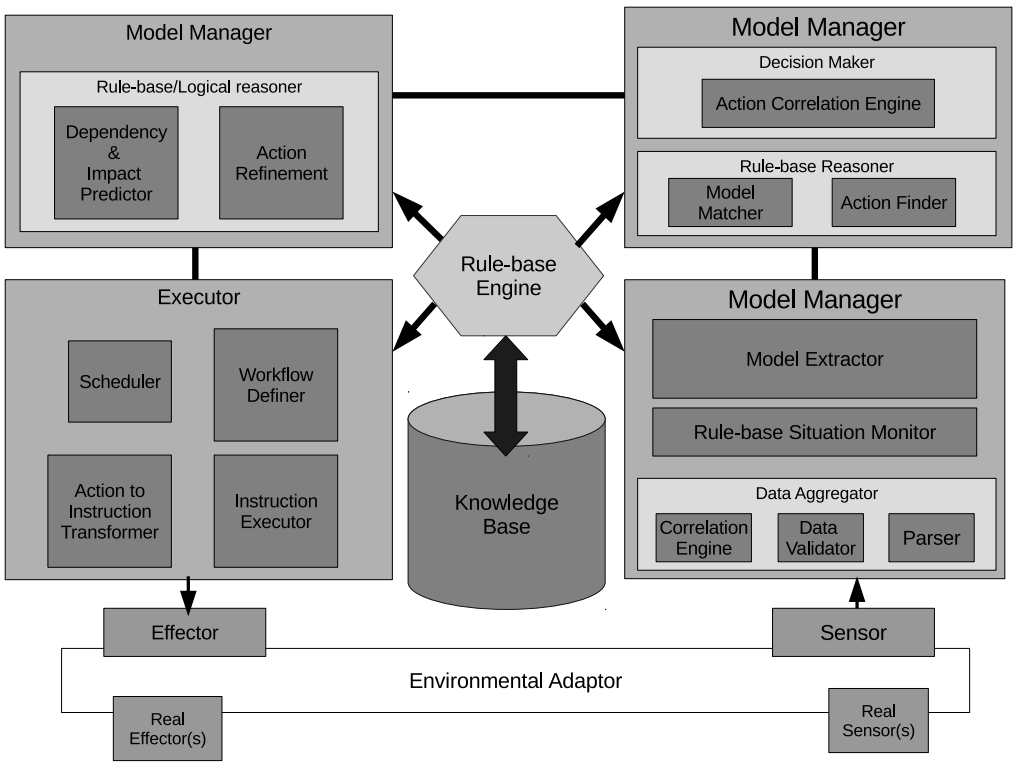


FIG. 4.3. *Self-healing Architecture (Detailed View)*

by the *Model Extractor*. The management of input data is in charge of the *Data Aggregator*, which is composed of the following parts:

- *Parser*: input data from the environment are different depending on context and data structures. Using a parser (which can be inside the *Sensor*), data within these data structures are extracted and changed into a predefined data structure. The *Parser* is implemented by "regular expression matcher" in the case study.
- *Validation Engine*: in case input data are inconsistent with the primary assumptions, the *Validation Engine* will make decisions about these situations (dismissing the data, some changes into the data). For example, in the painting problem, if the position of the area which was not painted correctly is calculated outside the painting area, this information will be ignored. This module examines primary assumptions. This module is implemented as a code execution engine which runs a predefined code on the data in order to validate the data.
- *Correlation Engine*: data received from different elements are aggregated and packed based on their relationship with each other. Data will be correlated with each other in order to construct a more abstracted data (which is usable for understanding the situation) and will cover the details which are not important for the problem. In the example, correlating the four different data about the locations of the body, hand, wrist and the nozzle into a single data structure (which is called position).

Others components of the *Model Manager* are:

- *Rule-based Situation Monitor*: This module is responsible for checking if the information can trigger some rules existing in the database. These rules determine the state according to the input data. In the case study, this module is implemented as a procedure that queries the knowledge base. Query consist of data such as success position (robot and automotive's part position) and failure positions in the painting process.

- *Model Extractor*: Extracted states are examined and the environment model is extracted based on these states. This step integrates and combines extracted situations from the Rule-based module into a set of models. In the case study, failure in past cycle painting (position 0,0), robot's position (50,10,10) and automotive part's position (-90) produce the model (0,0,50,10,10, need_to_reposition(true,true,true)) The model is translated into the following call:
  `method signature:`
  `model (failure's position X, failure's position X, robot's hand position, robot writ's angle in X direction, robot writ's angle in Y direction need_to_reposition( hand's position (true/false), wrist's position (true/false), nozzle's position (true/false)))`

4.4. Model Analyzer. In Model Analyzer subsystem, first models are handed over to the system as inputs. It must be determined whether, based on these models that describe states of the system, any event has occurred which requires a reaction. Hence, the existing model in the *KB* (Knowledge Base) is searched for and checked using inference methods so similar models to the current one are extracted.

In the case study, extracted situations and models will be examined and the suitable actions for these situations and model will be extracted for responding to situations and models. Also, if it is possible, these actions will be merged together.

The modules are the following:

- *Model Matcher* In this module models and situations which are reported by *Model Manager* are examined with the knowledge in the KB and if there is an exact or even similar situation reported in the KB would be found. In the example examining the `(0,0,50,10,10,need_to_reposition(true,true, true))` will match `bad_painting_near_position`.
- *Action Finder*: This component will use the knowledge in KB and also the model in order to find appropriate set of actions for the situation. This set of actions could be extracted from different knowledge in the KB. In the Example for previous situation based on knowledge (because the area is in the center not in the corner, maybe the area is not flat and needs to be painted from above and beyond) will result this action sets.
  `paint (repaint, from_above, 0,0,-10), paint (repaint, from_beyond, 0,0, -10)`
  the description of paint procedure is as follows:
  `paint (repaint/first_time_paint/paint_over_paint, direction of painting (from_above/ from_beyond/front/back), position X, position Y, position in Z)`

These two modules together are called *Rule-Based Reasoner*.

- *Action Correlation Engine*: This module, if there is an option available for correlating some actions with each other, it will reduce the actions set. In the case study, if action set is like this: `paint (repaint, from_above, 0,0,-10), paint (repaint, from_above, 0,-10,-10)` it will change into `paint (repaint, from_above, 0,0,-10,-10)` because these two procedures have so many similar moves in themselves. This mixed data will be refined in the next component.

After preparing the action sets, these action sets can be called "*decisions*". These decisions are entered the Self-healing Engine module.

4.5. Self-healing Engine. Responding to the changes by performing doable decisions is the goal of the architecture presented in this paper. In order to prepare doable decisions from the system's knowledge, it is necessary to first determine the relationships and dependencies between the decisions and the required resources for the decisions using an inference engine and existing knowledge in *action detailer*. After refining actions into detailed actions (i.e., a set of detailed actions could be extracted from an action) and extracting the relation between decisions and required resources for each detailed actions, these data will be used in the process of dependency and impact checking. Afterward, the list of doable actions will take places as the result of this module.

- *Action Refinement*: In this module, overall decisions of the previous phase are transformed to refined and precise decisions applicable to environment using the existing knowledge in the Knowledge Base and logical rules. In the case study, the action set `paint (repaint, from_above, 0, 0, -10, -10)` is translated into this set of actions:

```
1. go(0,-10,0)
2. shoot(1000)
3. go(0,-10,-5)
4. shoot(1200)
```

- *Dependency and Impact Predictor*: In this module the impact and influence of each step are gathered and, if there is a conflict, this module will take decision about doing one of these three options for those actions: 1) stop *both* of them, 2) stop the *second* one, 3) run both of them accepting the *risk*.
  The work of this module is similar to that in RAINBOW that defines a utility function for knowing about the future impact of each decision on the adaptation goal [17]. In the case study, if the time slot is free for 3 seconds and if action 1+2+3+4 are going to take more than 3 seconds, the action 3 and 4 will be eliminated. This decision making is stored in the knowledge base in a simple rule form.

These resulting decisions are sent to the *Executor*.

4.6. Executor. In order to be executed, decisions must include a priority planning and a scheduling, and then they can be executed. The responsibility for scheduling the priority of these decisions and with which precedence they are executed is of the *Workflow Definer* module. Also, planning and determining schedules for these decisions is done by the *Scheduler* module. After scheduling, based on the knowledge, the outputs are transformed from these actions to a series of instructions in system level. This task is carried out by the *Action to Instruction Transformer* module. These instructions are executed in a row by the *Instruction Executer* and the help of the *Effector*.

As happens in the RAINBOW framework, for separating internal system knowledge from the system which adaptive section affects, there is an *Environment Adaptor* with knowledge about translating events and inputs. This module translates the instruction to applicable instructions and also as a part for importing data to the presented adaptive loop, it will translate the information and inputs into comprehensible information into the understandable information for *Model Manager*. This task gives the possibility to reuse the developed system in different applications. For example, if there is another production line with different products, different robots with different tasks (for example welding instead of painting), the *Self-healing Engine* module could be informed of the changes by making a change in the *Knowledge*. The self-healing section of the system will start to act properly in a new situation. This is possible by translating the environment state into an understandable language for the self-healer section of the system. For example, if robot's location is (-10,10,0) and the automotive part's location is -90 so the distance will be translated from these data as -A ($70 \leq A \leq 80$).

The modules in this subsystem are:

- *Scheduler*: This module will do the scheduling based on the knowledge in the KB. In the case study, painting the location which is far from the nozzle sooner than another location is an example of the knowledge in the KB. Results will be like the following.
  ```
  1. go(0,-10,-5)
  2. shoot(1200)
  3. go(0,-10,0)
  4. shoot(1000)
  ```
- *Workflow Definer*: This component will decide about the order and timing of the tasks that are going to be done based on the knowledge in the KB. In the case study, the following tasks will be the results.
  ```
  1''   -> go(0,-10,-5)
  5.5'' -> shoot(1200)
  7''   -> go(0,-10,0)
  10''  -> shoot(1000)
  ```
- *Action to Instruction Translator*: Translations of the instructions from high-level instructions into the understandable instructions for robot will be done in this module. After this translation, the instructions are handed over to the *Effector* and will be execute. In the case study, the previous four tasks are translated into the following G-code, based on the knowledge in the KB.

  ```
  COMMAND
  1.0    MOVE   0  -10  0
  ```

```
#(absolute delay) (command) (x)  (y)  (z) (pre_delay) (task_time)
5.5    COLOR  0  -10  0   0.000   1.200
#(absolute delay) (command) (x)  (y)  (z)
7.0    MOVE   0  -10 -5
#(absolute delay) (command) (x)  (y)  (z) (pre_delay) (task_time)
10.0   COLOR  0  -10 -5   0.000   1.000
END COMMAND
```

- *Instruction Executer*: This module will take responsibility for running prepared instruction on the environment.

4.7. Environmental Adaptor. In this module, if there are any changes (for better understanding or unification of data models) needed, these modifications will be applied. For example, if the idle situation of the robot is called (-10,0,0) in the system and we know that the robot needs floating point number for input (-10.00, 0.00, 0.00) output data will be change into floating point type. Input data from the environment are also comprehended by the *Sensor* and after applying necessary changes will be sensed by a virtual sensor and they enter the *Model Manager* module.

In the case study, the previous four tasks are translated into the following G-code, based on the knowledge in the KB.

Given these explanations and getting familiar with the presented architecture, in the next section, the results of the implementation of the case study with the proposed architecture will be discussed.

5. Implementation and Experimentation (Evaluation and Measurement). In this section all the results from the implementation of the painter robot system will be discussed in two phases, corresponding to the two main improvements we made starting from the original system. Also, for each phase, some implemented module will be discussed. We report the results of the experiments in Fig. 5.1, 5.3 and 5.2; in all figures, the first three items are from the phase 0 (system without improvement), the second three items are from phase 1 and the other items (the last four items) are from phase 2.

5.1. Phase 1. For phase 1 the code had been refactored in order to obey the architecture of a self-adaptive system. The architecture has derived from the RAINBOW architecture. Refactoring procedure took 14 days and after that period refactored code become capable of being adapted easily to the changes and able to support new automotive part easily.

Fig. 5.1 reports the time for preparing the code for each automotive part which this robot supports. Fig. 5.1 shows that the preparation time for changing the code in order to support a new automotive part has been significantly reduced from phase 0 to phase 1, from 1 person for a month to 6 days of a person's time. The reduction in preparation time also will be discussed in next section for phase 2.

Hence, the reduction in preparation time was the main improvement project goal, other improvements also came up from the code refactoring. As it is shown in the Fig. 5.2, the quality of painting after refactoring has seen an improvement. It is because that remediation of known problems in painting becomes easier by just calling some predefined procedures. For example, by running the robot for a new automotive part and finding the area that needs more painting, the predefined procedure for repainting the area (in robot's free time slot) could be exploited easily so the robot paints a larger area and the quality of the painting improves.

As it could be guessed, these improvements will consume some other resources. As it is obvious from Fig. 5.3, the portion of the time slot in which the robot is moving and painting increased after the code refactoring (phase 1). Having predefined procedure for remediation of the painting problems makes preparing of the remediation code much easier and this will lead into more remediation per time slot. Also, this predefined procedure will increase the time of the remediation (even the same remediation) because of eliminating the ability to optimize the path and painting of the robot by integrating two or more predefined procedures in one set of actions.

5.2. Phase 2. In phase 2 the refactored code had faced a significant change. The code's architecture changed into a more mature architecture. The deduction and conclusion which before were taken on the base of if-then-else code changed into a set of knowledge which is implemented in the packages related to the deduction. Also, data and information that traverse through the system were unified into predefined information. These

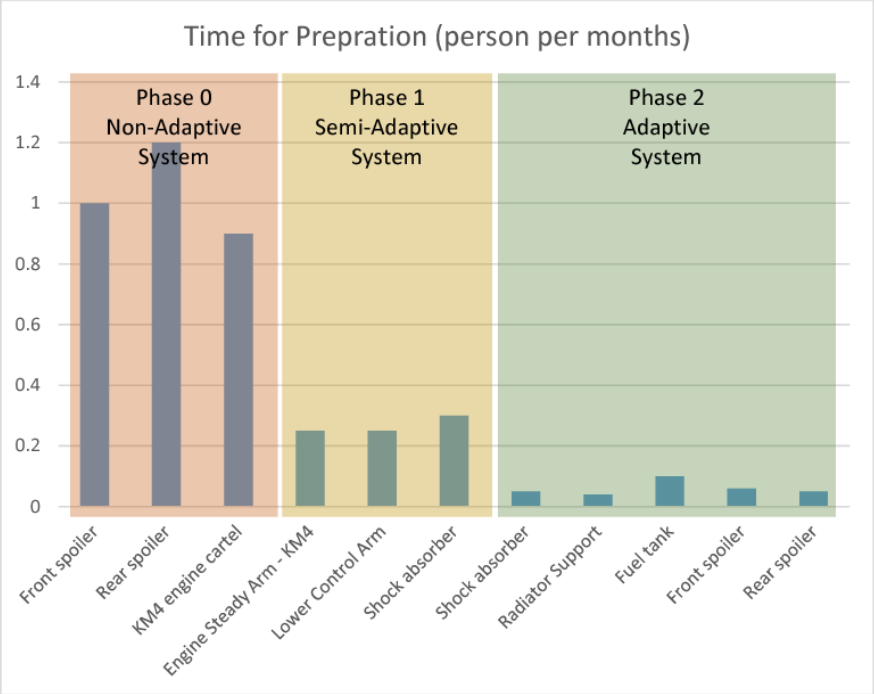


FIG. 5.1. *Preparation time for each automotive part*

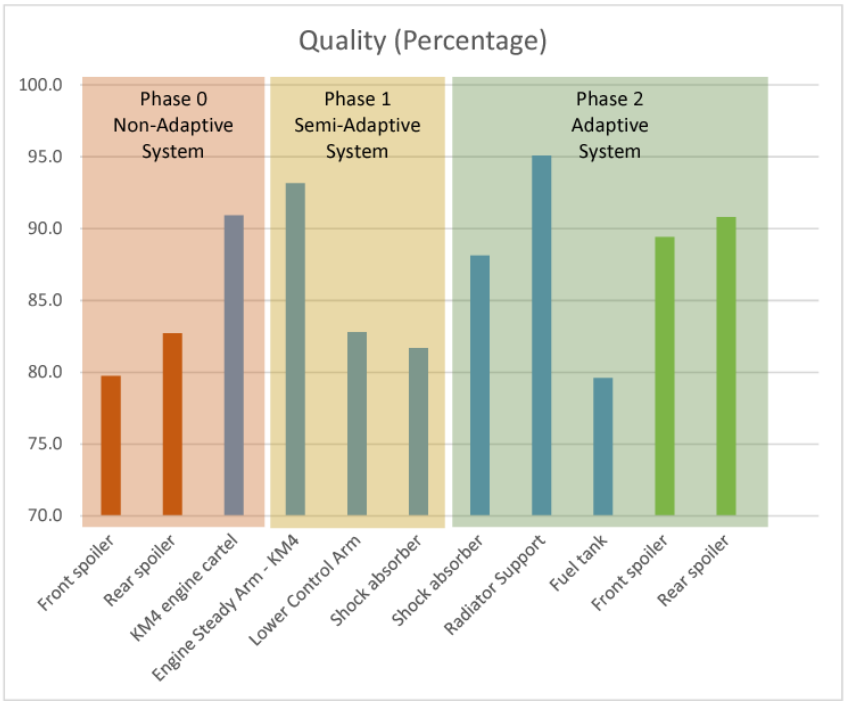


FIG. 5.2. *Quality of painting*

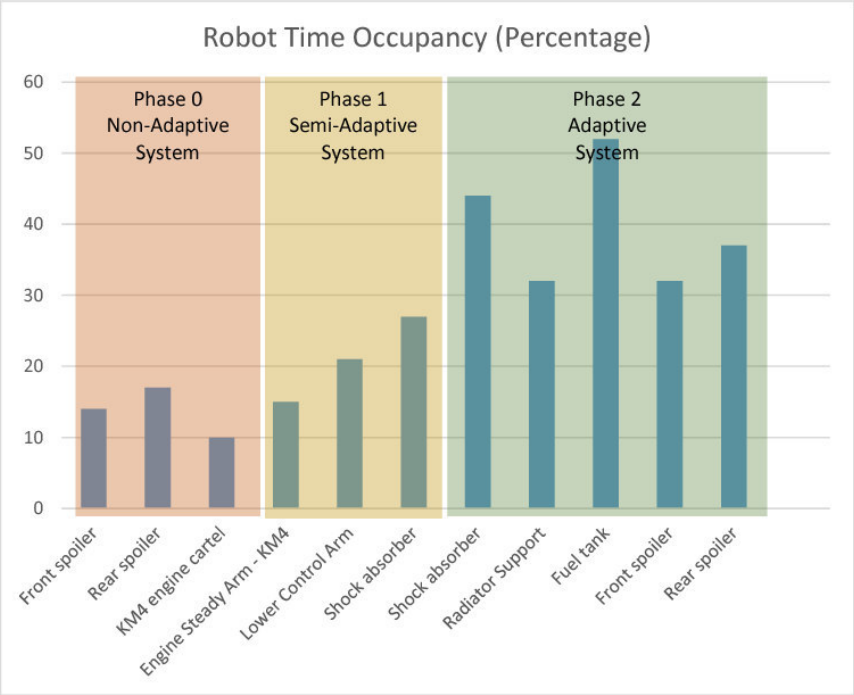


FIG. 5.3. *Robot's time occupancy (percentage of time)*

changes influence the time that is needed for preparing the system for new automotive parts, making it adaptable in a faster way. The time percentage of four automotive parts that had been painted in phase 2 can be seen in the figures. Times for these four parts painting had a significant reduction even compared to phase 1. For example, the first automotive parts (front and rear spoiler) took 1 and 1.2 person months in phase 0, while after the refactoring and implementation in phase 2 they took only 2 days. Also, as it is reported in Fig. 5.2 the quality of painting of these two parts were improved from around 80% to around 90%.

As mentioned before, these improvements imply some costs. The amount of occupied time slots of the robot increased from phase 0 to phase 2. In phase 0 it was around 10-15% and phase 1 has increased to around 15%, while in phase 2 it was increased to around 35%.

6. Conclusion. Computer systems are required to face new and often unpredicted situations during their execution. To react to the faults or to the changes in the environment they must exhibit the self-healing property, introduced in the context of AC, which enables the system to recognize incorrect results and to provide a remediation, by modifying its behaviour. This is true in particular in industrial environments, where the physical part of the systems are subject to faults and variation in the behaviour.

In this paper, we have reported an experience in the field of automotive, in particular, a system to paint parts in an automatic way. We have presented the previous version of the system, which was quite rigid and required a significant effort to repair faults and to be adapted to new parts. We have proposed a self-healing architecture to overcome these limitations.

A case study shows the applicability of the proposed architecture. This case study has been exploited to make some experiments in order to present the improvements in painting quality besides the reduction in time of preparation of the system for a new task. The cost of the improvements is more robot's time occupancy.

In the future works, this architecture could be checked for conflict and error by formal methods and also by software architecture evaluation methods such as ATAM [26] and CBAM [27]. In addition, the proposed architecture will be exploited in other case studies, to verify its generality.

REFERENCES

[1] D. Garlan and M. Shaw, *An Introduction to Software Architecture* ,Addison-Wesley, 1994.
[2] R. Isermann, *Fault-diagnosis systems: an introduction from fault detection to fault tolerance.*, Springer, 2005.
[3] S. Armando and G. Betancur, *Fault management in TELCO platforms using Autonomic Computing and Mobile Agents*, 2011.
[4] N. Technical, *Fault management handbook*, 2012.
[5] D. Garlan and B. Schmerl, *Model-based adaptation for self-healing systems*, Proceedings of the first workshop on Self-healing systems - WOSS 02, p. 27, 2002.
[6] S. Bouchenak, F. Boyer, D. Hagimont, S. Krakowiak, A. Mos, N. D. Palma, V. Quema, and J. Stefani,*Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters*, pp. 120.
[7] D. Garlan and B. Schmerl, *Model-based adaptation for self-healing systems*, Proceedings of the first workshop on Self-healing systems - WOSS 02, p. 27, 2002.
[8] P. Eeles and P. Cripps, *The process of software architecting*, Pearson Education, 2009.
[9] J. O. Kephart and D. M. Chess, *The vision of autonomic computing*, Computer, vol. 36, no. 1, pp. 4150, 2003.
[10] A. Computing, *An architectural blueprint for autonomic computing*, IBM White Paper, no. April, 2003.
[11] D. W. Cheun and S. D. Kim, *An Engineering Process for Autonomous Fault Management in Service-Oriented Systems*, 2010 IEEE/ACIS 9th International Conference on Computer and Information Science, pp. 901 906, Aug. 2010.
[12] R. Sterritt, *Autonomic computing*, Innovations in Systems and Software Engineering, vol. 1, no. 1, pp. 7988, Mar. 2005.
[13] Y. Qun, Y. X. XU, and X. Man-wu, *A Framework for Dynamic Software Architecture-based Selfhealing.*
[14] G. Antoniou, M. Baldoni, W. Nejdl, and D. Olmedilla, *Chapter 1 RULE-BASED POLICY SPECIFICATION*,
[15] D. Garlan, S. Cheng, and A. Huang, *Rainbow: Architecture-based selfadaptation with reusable infrastructure*, Computer, pp. 4654, 2004.
[16] D. W. Cheun and S. D. Kim, *An Engineering Process for Autonomous Fault Management in ServiceOriented Systems*, 2010 IEEE/ACIS 9th International Conference on Computer and Information Science, pp. 901 906, Aug. 2010.
[17] B. Schmerl, J. Cámara, J. Gennari, D. Garlan, P. Casanova,G. A. Moreno, J. M. Barnes, *Architecture-based self-protection: Composing and reasoning about denial-of-service mitigations*, ACM International Conference Proceeding Series, 2014
[18] J. Camara, P. Correia, D. Lemos, D. Garlan, P. Gomes, B. Schmerl, R. Ventura, *Evolving an Adaptive Industrial Software System to Use Architecture-based Self-Adaptation*, Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2013 ICSE Workshop on, 1322
[19] R. Mirandola, P. Potena, *A QOS-based Framework for the Adaptation of Service-based Systems*, SCPE, 2011
[20] E. Wegner,*WebNMS Framework: A Complete EMS Framework*,White Paper, 2012, www.webnms.com.
[21] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, *Towards architecture-based self-healing systems*, Proceedings of the first workshop on Selfhealing systems - WOSS 02, p. 21, 2002.
[22] B. Gaudin, M. H. Hinchey, E. Vassev, J. Garcia, W. Coelho Maalej, *FASTFIX: A Control Theoretic View of Self-healing for Automatic Corrective Software Maintenance*, SCPE, 2012
[23] D. B. Abeywickrama, N. Hoch, F. Zambonelli, *Engineering and Implementing Software Architectural Patterns Based on Feddback Loops*, SCPE, 2014
[24] S. Myat, M. Soe, M. Paing, P. Zaw, , *Design and Implementation of Rule-based Expert System for Fault Management*, World Academy of Science, Engineering and Technology, 3439, 2008
[25] D. Garlan, B. Schmerl, *Using Architectural Models at Runtime: Research Challenges*, EWSA Workshop, 2004
[26] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, K. Lipson, J. Carriere, *The architecture tradeoff analysis method*, Sei - Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Institute, 1998
[27] R. L. Nord, M. Barbacci, P. Clements, R. Kazman, M. Klein, L. O'Brien, J. E. Tomayko, *Integrating the Architecture Tradeoff Analysis Method (ATAM) with the cost benefit analysis method (CBAM)*, Sei - Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Institute, 2003

Edited by: Dana Petcu
Received: July 13, 2016
Accepted: August 18, 2016

DOI 10.12694/scpe.v17i4.1207
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 4, pp. 369–391. http://www.scpe.org

AQSORT: SCALABLE MULTI-ARRAY IN-PLACE SORTING WITH OPENMP

DANIEL LANGR, PAVEL TVRDÍK AND IVAN ŠIMEČEK *

Abstract. A new multi-threaded variant of the quicksort algorithm called AQsort and its C++/OpenMP implementation are presented. AQsort operates in place and was primarily designed for high-performance computing (HPC) runtime environments. It can work with multiple arrays at once; such a functionality is frequently required in HPC and cannot be accomplished with standard C pointer-based or C++ iterator-based approach. An extensive study is provided that evaluates AQsort experimentally and compares its performance with modern multi-threaded implementations of in-place and out-of-place sorting algorithms based on OpenMP, Cilk Plus, and Intel TBB. The measurements were conducted on several leading-edge HPC architectures, namely Cray XE6 nodes with AMD Bulldozer CPUs, Cray XC40 nodes with Intel Hasswell CPUs, IBM BlueGene/Q nodes, and Intel Xeon Phi coprocessors. The obtained results show that AQsort provides good scalability and sorting performance generally comparable to its competitors. In particular cases, the performance of AQsort may be slightly lower, which is the price for its universality and ability to work with substantially larger amounts of data.

Key words: C++, high performance computing, in-place sorting, many-core, multi-array sorting, multi-core, multi-threaded algorithm, OpenMP, parallel partitioning, parallel sorting

AMS subject classifications. 68P10, 68W10

1. Introduction. The demand for sorting multiple arrays at once emerges not only in high-performance computing (HPC) codes. Such codes are mostly written in C/C++ and Fortran, however, common implementations of sorting algorithms in these languages do not support multi-array sorting. Pointer-based sorting routines—such as `qsort` from the C standard library [10, §7.20.5.2]—and their iterator-based generalizations—such as `std::sort` from the C++ standard library [11, §25.4.1.1]—can operate on a single array only. The Boost library [28] has introduced so-called zip iterators that can work with multiple arrays at once; however, Boost zip iterators are read-only iterators and therefore do not provide a solution for multi-array sorting. Generally, it is not feasible to create a portable standard-compliant implementation of zip iterators that can modify underlying arrays.[1]

The multi-array sorting problem has been frequently addressed by developers; numerous threads of C/C++ mailing lists and web forums have been devoted to this topic.[2] The suggested solution is practically always the same—to transform the *structure of arrays* (SoA) into a single *array of structures* (AoS), then perform sorting, and finally copy data back from AoS to SoA. Such a solution has the following drawbacks:

1. There must be enough free memory to perform the SoA-to-AoS transformation. Namely, the amount of free memory must be at least the same as the amount of memory occupied by the data that need to be sorted. Such a constraint might be inconvenient especially in HPC, where not the computational power but the amount of memory often limits the sizes of problems being solved. Users of HPC programs thus might in practice need to work with multi-array data structures that occupy more than the half of available memory, e.g., with representations of sparse matrices or meshes for spatial discretization of PDEs. Sorting/reordering of these data structures via SoA-to-AoS transformation would be not possible under such conditions.
2. The SoA-to-AoS and back AoS-to-SoA transformations imposes into programs a runtime overhead.

Sorting algorithms can be classified as being either in-place or out-of-place. We define an *out-of-place/not-in-place* sorting algorithm as an algorithm that requires $\Omega(n)$ extra space for auxiliary data structures, where n denotes the number of sorted elements; for a multi-array sorting problem, n denotes the number of sorted elements of each of the arrays. On the contrary, we define an *in-place* sorting algorithm as an algorithm that needs $o(n)$ extra space. Out-of-place sorting algorithms, such as mergesort, have typically the same amortized memory requirements as the SoA-to-AoS transformation. That is, the amount of free memory must be at

*Department of Computer Systems, Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, 160 00, Praha, Czech Republic (`langrd@fit.cvut.cz`).

[1]The problem stems from the fact that it is not possible to create a pointer or a C++ reference to multiple data entities. For example, the following code is perfectly valid for an implementation of a sorting algorithm in C++: `auto& next_elem = *(iter + 1); next_elem = std::move(...);`. But in case of multiple arrays, there does not exist any entity that `next_elem` could reference. A detailed discussion about this problem is beyond the scope of this text.

[2]Look, e.g., for posts regarding multiple arrays/vectors sorting in C/C++ on the Stack Overflow community developer site (http://stackoverflow.com).

least the same as is occupied by the sorted data. Within this article, we primarily address multi-array sorting problems where such an amount of free memory is not available. Therefore, for the solution of these problems, neither the SoA-to-AoS transformation nor the out-of-place sorting can be used.

Growing capacities of shared memories of computer hardware architectures and growing numbers of their computational cores have raised the demand for parallel/multi-threaded algorithms; we do not consider distributed-memory parallelism in this text, therefore, we use the terms *parallel* and *multi-threaded* as synonyms. In HPC, the utilization of shared memories via hybrid parallel paradigms, such as the combination of MPI and OpenMP, often outperforms distributed-memory implementations (pure MPI). However, the amount of data that fit shared memories might become so large that their sequential sorting would introduce into applications significant hotspots. For illustration, sorting of an array of 6.4 billions of integers, which fit 64 GB of memory, with sequential `std::sort` took in our measurements over 18 minutes on a contemporary CPU-based system. Using parallel in-place sorting algorithms, we were able to reduce the sorting time to less than 2.5 minutes (speedup 7.5) utilizing all 16 system cores. Similarly, we observed the reduction of runtime from 12 minutes to less than 24 seconds (speedup over 30) when sorting 1.6 billions of integers on an Intel Xeon Phi many-core coprocessor.

Available parallel C/C++ implementations of sorting algorithms usually adopt the pointer-/iterator-based approach of their sequential counterparts. Consequently, they are not able to work with multiple arrays at once. In this article, we present a new parallel variant of the in-place quicksort algorithm called AQsort that does not impose such a constraint. Instead of pointers/iterators, AQsort works with user-provided routines for comparing and swapping sorted data. The drawback of this approach is that it hinders some possibilities to optimize and tune resulting programs by developers and compilers (see Sect. 6 for details). One of the purpose of the presented experimental study is therefore to evaluate the effects of such a restriction.

The structure of this article is as follows. Section 2 introduces the state-of-the-art implementations of parallel sorting algorithms. Sections 3 and 4 describe the AQsort algorithm itself and its C++/OpenMP implementation, respectively. Section 5 presents a study that evaluates AQsort experimentally on modern HPC architectures and compares its performance with its not-multi-array competitors. Section 6 discusses the problem of choosing the most suitable sorting solution according to user's needs and constraints. Finally, Section 7 summarizes our work and concludes the article.

1.1. Motivation. Our research is tightly connected to sparse-matrix computations, mainly to sparse matrix storage formats. These formats determine the way how matrix nonzero elements are stored in computer memory. The simplest format is so-called *coordinate storage format* (COO) [1, 27] that consists of three arrays containing row indexes, column indexes, and values of matrix nonzero elements. COO does not prescribe any order of the elements in these arrays and it is the most suitable format for assembling sparse matrices—generated nonzero elements are simply to the arrays appended. However, COO has high memory requirements; it is therefore not a suitable format for sparse-matrix computations, which are generally bound in performance by memory bandwidth [14, 35].

In practice, likely the most commonly-used format for sparse matrix computations is the *compressed sparse row format* (CSR, CRS), together with its *compressed sparse column* (CSC, CCS) counterpart [1, 27]. The conversion of sparse matrices from COO to CSR consists of two steps: First, the nonzero elements in COO arrays are sorted lexicographically, which represents a multi-array sorting problem. Then, the array of row indexes is substituted by an (hopefully much smaller) array that indicates how many nonzero elements are in each row and where their column indexes and values can be found.

Numerous other formats have been developed in the past that were shown to provide high performance of sparse matrix computations on modern multi-core and many-core architectures, typically in comparison with CSR. Generally, these formats have two common features: (1) they store matrix nonzero elements in memory in some particular order, and (2) they are more or less parametrized. Finding (pseudo)optimal parameters for a given matrix and transforming this matrix into a given format typically involves multiple sorting of the COO arrays. For example, uniformly-blocking formats are parametrized by the block size [13]. To find an optimal block size, matrix nonzero elements need to be sorted repeatedly with respect to different tested block sizes [12, 29].

To amortize the usage of a given format in subsequent matrix-related computations, we thus need a fast scalable sorting algorithm. To allow users to work in their HPC programs with large matrices that occupy more than a half of available memory and thus effectively allow them to solve correspondingly large computational

problems, we need this algorithm to be in-place. Finally, to allow developers to integrate such an algorithm into their codes, we need its efficient portable OpenMP implementation, since the OpenMP threading paradigm [5] prevails in sparse matrix-related and HPC codes in general. All of these requirements AQsort fulfils.

2. Related Work. Many C/C++ implementations of parallel sorting algorithms have been developed in the past. To our best knowledge, we have not found any one capable of generic multi-array in-place sorting. To evaluate AQsort, we therefore compared its performance with the performance of iterator-based sorting functions from forefront libraries provided by GNU, Nvidia, and Intel.

GNU implements parallel sorting algorithms in the scope of its parallel version of the C++ Standard Library, namely the GNU Libstdc++ Parallel Mode [30]. It provides functions for both in-place and out-of-place sorting; the former implements the parallel quicksort algorithm proposed by Tsigas and Zhang [34], the latter implements a parallel multi-way mergesort. Moreover, the parallel quicksort exists in the library in a balanced and an unbalanced variant, which differ in the way of assigning threads to partitions of unequal sizes. The library uses the OpenMP threading paradigm and was originally developed as a standalone software project called the Multi-Core Standard Template Library (MCSTL) [31]. At the time of writing this article, the GNU Libstdc++ Parallel Mode was referred to as "an experimental parallel implementation of many C++ Standard Library algorithms".

Nvidia provides parallel sorting functions in the scope of its library called Thrust. The documentation does not discuss the implementation in detail, but according to the source code, the functions seemingly implement an out-of-place parallel mergesort. Thrust was primarily designed for GPGPU programming, but it also supports OpenMP as an underlying threading paradigm [3].

Intel provides parallel sorting functions for their own threading paradigms/frameworks Cilk Plus [26] and Thread Building Blocks (TBB) [24]. Cilkpub—a library of community-contributed Cilk Plus code—contains functions that implement both a parallel in-place quicksort and a parallel out-of-place samplesort. Intel TBB contains an implementation of an in-place parallel quicksort. Some details of these functions are provided by McCool et al. [18].

We have chosen the above mentioned (iterator-based) implementations since, thanks to their providers, we presumed highly efficient and optimized codes targeting modern hardware architectures. However, note that there are numerous other parallel implementations of sorting algorithms as well, available either in the form of standalone codes or within some more generic libraries/frameworks. These include, e.g., Intel PSS [25], OMPTL [2], Parallel STL [19], ParallelSort [21], psort [16], STAPL [33], and STXXL [7].

2.1. Parallel Quicksort. Current implementations of generic in-place sorting algorithms are mostly based on quicksort [8, 9]. Quicksort partitions sorted data according to a so-called pivot element and then recursively calls itself for both resulting parts. In multi-threaded environments, the recursive calls are natural candidates for task parallelism, which is available, e.g., in OpenMP since version 3.0, Intel TBB, and Intel Cilk Plus. However, partitioning needs to be parallelized as well; sequential partitioning at top levels of recursion would significantly hinder the scalability of parallel quicksort.

Tsigas and Zhang [34] proposed a parallel quicksort with efficient parallel partitioning that is widely used in practice and frequently mentioned in literature. Alternative solutions and their comparison has been presented by Pasetto and Akhriev [22, 23]. Within they work, they also proposed a new approach to parallel partitioning, however, they defined it only by words and did not provide a corresponding algorithm [23, Section 2.4].

Another parallel quicksort have been proposed by Mahafzah [15], however, he does not provide experimental comparison with other solutions. Moreover, he presents results only for small data up to 80 MB of memory footprint and small number of threads up to 8. Süs and Leopold [32] compared several parallel implementations of quicksort based on OpenMP and POSIX threads (Pthreads).

In practice, efficient implementations frequently combine quicksort with other sorting algorithms. One reason is the quicksort's worst-case complexity $O(n^2)$. To deal with the worst cases, such implementations allow the recursive process happen only to some maximum depth, commonly set to $\lfloor 2 \log_2 n \rfloor$. If it is exceeded, the quicksort is abandoned and the rest of the not-yet sorted data is processed by another sorting algorithm with the $O(n \log n)$ worst-case complexity, typically by heapsort [6]; such a combination of quicksort and heapsort is referred to as *introspective sort* or shortly *introsort* [20]. Another reason for combining quicksort with other algorithms is that applying quicksort to very small partitions might be inefficient; recursive calls are relatively expensive here. When partitions of sizes below some cutoff parameter are reached, they are thus usually sorted with some simple $O(n^2)$ algorithm, typically with insertion sort [6].

3. Algorithm Design. Available iterator-based C++ sorting functions—both sequential and parallel—typically adhere to the following declaration pattern:

```
template <class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

The `first` and `last` parameters represent random-access iterators that determine data to be sorted. The `comp` binary function decides whether its first argument should appear before the second in the sorted data. Unfortunately, there is no portable way how to construct writable random-access iterators for multiple arrays. To solve the in-place multi-array sorting problem, we thus need to give up the iterator-based approach.

Let us introduce some terminology used in the text below. Up to now, we have spoken about multi-array sorting, i.e., sorting of multiple arrays at once. However, by the term *array*, we generally mean any sequence of data, i.e., any data structure that contains multiple elements accessible via indexes; in C++, such data structures can be represented, e.g., by the `std::vector` and `std::array` STL containers, or ordinary C-style one-dimensional arrays. We assume that all the arrays have n elements indexed from 0 to $n-1$. We refer to the elements with the same index in all these arrays as to a *multielement*.

In a multi-array sorting problem, we thus want to sort n multielements according to some order given by *sorting keys* that can be derived from multielements data. We say that a multielement M_i is *less than* another multielement M_j if M_i should take place before M_j in the sorted arrays, according to their sorting keys. Otherwise, we say that M_i is *greater than or equal to* M_j.

Let us consider reordering of sparse matrix nonzero elements; a multielement is then represented by a single matrix nonzero element, i.e., by its row index, column index, and value. If we want to sort matrix nonzero elements, e.g., lexicographically (such as for conversion to CSR), we might define sorting keys for the ith multielement as

```
std::size_t key(std::size_t i) { return rows[i] * n_cols + cols[i]; }
```

where `n_cols` equals the number of matrix columns and the arrays `rows`/`cols` contain row/column indexes of matrix nonzero elements.

The AQsort implementation declares sorting functions as follows:

```
template <typename Comp, typename Swap>
void sort(std::size_t length, Comp* const comp, Swap* const swap);
```

The parameter `length` represents the number of multielements that need to be sorted, i.e., n. The parameter `comp` is a pointer to a binary function that takes two arguments and returns `true` if the multielement indexed by the first argument is less than the multielement indexed by the second argument; otherwise, it returns `false`. Finally, the parameter `swap` is a pointer to a function that takes two arguments and swaps multielements indexed by these arguments.

Thanks to this approach, AQsort has no information about data being sorted. The algorithm does not know how many arrays are sorted at once, it cannot recognize the types of the elements in these arrays, and there is no way how it could access these elements. It also does not work with sorting keys, only indirectly through the `comp` function. This function needs to derive sorting keys for both indexed multielements, compare them, and return the appropriate binary value. It is up to AQsort users to provide the expected functionality of `comp` and `swap` over the sorted arrays.

Let us now present the AQsort algorithm itself. Parallel quicksort is carried out by the `ParallelQuickSort` procedure presented by Algorithm 1. Its input consists of the following parameters:

1. n: total number of sorted multielements,
2. *numthreads*: total number of threads,
3. *start*: index of the first multielement to be sorted by this call,
4. *count*: number of multielements to be sorted by this call,
5. `Comp`: reference to a binary function that compares multielements with two different indexes and returns true if the first multielement is less than the second one; otherwise returns false,
6. `Swap`: reference to a procedure that swaps multielements with two different indexes,
7. *level*: auxiliary parameter for preventing quicksort worst case complexity $O(n^2)$.

Initially, `ParallelQuickSort` is called from a single thread with the following arguments:

$$start = 0, \quad count = n, \quad level = 2 \cdot \lfloor \log_2(n) \rfloor, \tag{3.1}$$

Algorithm 1 Main recursive procedure of AQsort

```
1: procedure ParallelQuickSort(n, numthreads, start, count, Comp, Swap, level)
2:     while true do
3:         if level = 0 then
4:             SequentialSort(start, count, Comp, Swap, level)
5:             return
6:         end if
7:         level ← level − 1
8:         T ← ⌊numthreads × count/n⌋
9:     end while
10:    if T < 2 then
11:        SequentialSort(start, count, Comp, Swap, level)
12:        return
13:    end if
14:    pivot ← SelectPivotMoM(start, count, Comp)
15:    Swap(pivot, start + count − 1)
16:    pivot ← start + count − 1
17:    lessthan ← ParallelPartition(start, count, pivot, Comp, Swap, T)
18:    Swap(start+ lessthan, pivot)
19:    greaterthan ← count − lessthan − 1
20:    while greaterthan > 0 and Comp(start + lessthan, start + count − greaterthan) = false and Comp(start +
       count − greaterthan, start + lessthan) = false do
21:        greaterthan ← greaterthan − 1
22:    end while
23:    if lessthan > greaterthan then
24:        run ParallelQuickSort(n, numthreads, start + count − greaterthan, greaterthan, Comp, Swap, level)
       in a new parallel task
25:        count ← lessthan
26:    else
27:        run ParallelQuickSort(n, numthreads, start, lessthan, Comp, Swap, level) in a new parallel task
28:        start ← start + count − greaterthan
29:        count ← greaterthan
30:    end if
31: end procedure
```

and *numthreads* set to the number of threads that a user wants to use for parallel sorting.

The `ParallelQuickSort` procedure works as follows:

1. If the maximum allowable depth of recursion is reached, `SequentialSort` is called, which further immediately proceeds to heapsort.
2. The number of threads that proportionally falls on the number of multielements processed by this call (*count*) is calculated and stored in T; the total number of threads as well as the total number of multielements therefore need to be passed to `ParallelQuickSort` as arguments. When T drops below 2, the processed multielements are sorted sequentially by calling the `SequentialSort` procedure.
3. Otherwise, a pivot multielement is selected and the processed multielements are partitioned with respect to this pivot using T threads by calling the `ParallelPartition` function.
4. In `ParallelQuickSort`, the pivot is chosen using the *median of medians* (MoM) strategy referred also to as *ninther*, which is usually recommended for large arrays; see, e.g., [4]. The MoM pivot selection is represented by the `SelectPivotMoM` function.
5. After parallel partitioning, multielements that lays behind the pivot and are equal to it are excluded from further processing, since they are already at their final positions. This might considerably improve the algorithm efficiency in cases when sorted arrays contain only few unique sorting keys.
6. So-called *tail call elimination* is applied to reduce the required call stack space (see, e.g., [6, Sect. 7-4]). `ParallelQuickSort` thus recursively calls itself only once instead of twice, and uses a while loop to

Algorithm 2 Parallel partitioning—Part 1

```
1: function ParallelPartition(start, count, pivot, Comp, Swap, T)
2:     m ← ⌊count/PBS⌋
3:     tleft[] ← integer array of size T
4:     tstart[] ← integer array of size T + 1
5:     for t = 0 to T − 1 do
6:         tstart[t] ← start + PBS × ⌊t × m/T⌋
7:     end for
8:     tstart[T] ← start + PBS × m
9:     for T threads do in parallel
10:        t ← actual thread number
11:        left ← tstart[t]
12:        right ← tstart[t + 1] − 1
13:        tleft[t] ← left+ SequentialPartition(left, right − left + 1, pivot, Comp, Swap)
14:        perform parallel barrier synchronization
15:    end for
16:    i ← 0
17:    j ← T − 1
18:    while i < j do
19:        imod ← (tleft[i] − start) mod PBS
20:        jmod ← (tleft[j] − start) mod PBS
21:        if imod = 0 then
22:            i ← i + 1
23:            continue
24:        end if
25:        if jmod = 0 then
26:            j ← j − 1
27:            continue
28:        end if
29:        ilast ← tleft[i] − imod + PBS − 1
30:        jfirst ← tleft[j] − jmod
31:        while tleft[i] ≤ ilast and tleft[j] − 1 ≥ jfirst do
32:            Swap(tleft[i], tleft[j] − 1)
33:            tleft[i] ← tleft[i] + 1
34:            tleft[j] ← tleft[j] − 1
35:        end while
36:    end while
```

process the second partition. The recursive call is always performed for the smaller partition.

The key for the good scalability of AQsort is efficient parallel partitioning at the high levels of the quicksort's recursive process accomplished by the `ParallelPartition` function. Its functionality stems from the solution described by Pasetto and Akhriev that was introduced in Section 2.1. We elaborated their concept into an fully-defined efficient blocking-based algorithm that is introduced by Algorithm 1.

The `ParallelPartition` function takes the following arguments:

1. *start*: index of the first multielement to be partitioned,
2. *count*: number of multielements to be partitioned,
3. *pivot*: index of a pivot multielement,
4. `Comp`, `Swap`: see Algorithm 1,
5. T: number of threads to be used for partitioning,

and returns the number of multielements that are less than the pivot. The functionality of `ParallelPartition` is as follows: It first splits the processed multielements into T parts of the same size, where T is the number of threads that are required to participate in parallel partitioning (lines 2–8). Each part is then independently partitioned by a single thread with respect to the pivot by calling the `SequentialPartition` function (lines 9–

Algorithm 3 Parallel partitioning—Part 2

```
37:   lessthan ← 0
38:   for k ← 0 to T − 1 do
39:       lessthan ← lessthan + tleft[k] − tstart[k]
40:   end for
41:   temp ← (tleft[i] − start) mod PBS
42:   if temp ≠ 0 and (start + lessthan < tleft[i] − temp or start + lessthan ≥ tleft[i] − temp + PBS) then
43:       if Comp(start + lessthan, pivot) then
44:           q ← PBS − temp
45:           while q > 0 do
46:               Swap(tleft[i], start + lessthan + q − 1)
47:               tleft[i] ← tleft[i] + 1
48:               q ← q − 1
49:           end while
50:       else
51:           q ← temp
52:           while q > 0 do
53:               Swap(tleft[i] − 1, start + lessthan − q)
54:               tleft[i] ← tleft[i] − 1
55:               q ← q − 1
56:           end while
57:       end if
58:   end if
59:   lthread ← 0
60:   rthread ← T − 1
61:   gleft ← tleft[0]
62:   gright ← tleft[T − 1]
```

15), which represents a standard sequential partitioning algorithm. For efficiency, multielements are processed in blocks, and the size of blocks represents a global AQsort parameter called *PBS*; such a blocking approach is inevitable for multi-core and many-core environments to avoid cache contention. The size of split parts is chosen to be an exact multiple of the block size.

We further distinguish 3 different types of blocks. A block is called *black* if all its multielements are less than the pivot. A block is called *white* if all its multielements are greater than or equal to the pivot. A block is called *grey* if it contains multielements of both types and the multielements less than the pivot are placed on its left side, thus have lower indexes, than the multielements greater than or equal to the pivot. After performing `SequentialPartition` by each thread, the corresponding part of multielements contains at most one grey block; the other blocks are either black or white. The next step is to "neutralize" these at most T grey blocks by swapping their multielements such that as a result, either only one or no grey block exists (lines 16–36). If it does, it is placed to its final position, which is already known (lines 37–58). The neutralization of grey blocks is performed sequentially; this step is very fast and there would be only little or no benefit from its parallelization.

After neutralization of the grey blocks—up to at most the single one—all black and white blocks are swapped in parallel so that all black blocks are finally placed left from white blocks (lines 59–105). Thought the most of the pseudocode of this step consists of a critical section, the most runtime is spent outside of it (lines 100–102).

In the end, `ParallelPartition` needs to process multielements that did not fit the blocking scheme. This consists of placing the remaining not-yet-processed multielements that are less than the pivot to the left side of the position where the pivot finally belongs (lines 106–111). This last step is performed sequentially (again, it is very fast and there would be only little or no benefit from its parallelization).

Now, it remains to show how sequential sorting is performed. AQsort uses the combination of quicksort, insertion sort, and heapsort. Heapsort is conditionally applied to prevent quicksort's worst case complexity $O(n^2)$. Insertion sort is applied to partitions smaller than a threshold given by an AQsort global parameter called *IST*. The `SequentialSort` procedure therefore first checks the number of multielements to be sorted. If it is greater than *IST*, it calls the `SequentialQuickSort` procedure, which performs a standard recursive

Algorithm 4 Parallel partitioning—Part 3

```
63:  for T threads do in parallel
64:      done ← false
65:      while true do
66:          enter critical section
67:              if gleft ≥ gright or gleft ≥ start + lessthan or gleft − PBS < start + lessthan then
68:                  done ← true
69:                  break
70:              end if
71:              while gleft ≥ tstart[lthread + 1] do
72:                  lthread ← lthread + 1
73:                  if lthread ≥ T then
74:                      done ← true
75:                      break
76:                  end if
77:                  gleft ← tleft[lthread]
78:              end while
79:              myleft ← gleft
80:              gleft ← gleft + PBS
81:              while gright > PBS and gright − PBS < tstart[rthread] do
82:                  if rthread = 0 then
83:                      done ← true
84:                      break
85:                  end if
86:                  rthread ← rthread − 1
87:                  gright ← tleft[rthread]
88:              end while
89:              if gright ≤ PBS then
90:                  done ← true
91:                  break
92:              end if
93:              myright ← gright − PBS
94:              gright ← gright − PBS
95:          exit critical section
96:          if done = true then
97:              break
98:          end if
99:          if myleft < myright then
100:             for k ← 0 to PBS − 1 do
101:                 Swap(myleft + k, myright + k)
102:             end for
103:         end if
104:     end while
105: end for
```

quicksort. Then, insertion sort is executed to finally sort multielements not sorted by quicksort.

The `SequentialQuickSort` procedure works as follows:

1. If the maximum allowable depth of recursion is reached, the quicksort is abandoned and the processed multielements are sorted with the `HeapSort` procedure.
2. Otherwise, a pivot multielement is selected and the processed multielements are partitioned with respect to this pivot by calling the `SequentialPartition` function.
3. The pivot is chosen using so-called *median of three* (Mo3) strategy, which is faster than MoM.
4. As in `ParallelQuickSort`, after partitioning, multielements that are equal to the pivot are excluded

Algorithm 5 Parallel partitioning—Part 4

```
106:     for k ← tstart[T] to start + count do
107:         if Comp(k, pivot) then
108:             Swap(k, start + lessthan)
109:             lessthan ← lessthan + 1
110:         end if
111:     end for
112:     return lessthan
113: end function
```

from further processing.

5. As in `ParallelQuickSort`, the *tail call elimination* is applied to reduce the required call stack space.
6. Quicksort is performed only for partitions with sizes greater than IST. Smaller partitions are left to be sorted by insertion sort.

4. Implementation. We provide an AQsort implementation in the form of an open-source GitHub project[3]. It is written in C++ and uses OpenMP as a threading paradigm; OpenMP version 3.0 or higher is required because of task parallelism. Its usage consists of:

1. including the main AQsort header file `aqsort.h`,
2. defining functions for comparing and swapping multielements,
3. calling one of the provided sorting functions.

The contents of `aqsort.h` is shown in Figure 4.1.

There are three sorting functions defined, all within the `aqsort` namespace. The first function called `parallel_sort` performs sorting in parallel by using OpenMP. The second function called `sequential_sort` performs sequential sorting. The third function called `sort` is a simple "switch" that proceeds to `parallel_sort` if OpenMP is available and to `sequential_sort` otherwise. Note that `parallel_sort` is accessible only if the `_OPENMP` preprocessor macro is defined. This macro is typically provided automatically by compilers in case that OpenMP threading is activated (as, e.g., when the `-fopenmp` command line argument is passed to the GNU C/C++ compilers).

The `sequential_sort` function can be employed in codes where OpenMP threading is not available, such as sequential programs. In HPC, these may also be pure MPI-based programs; their MPI processes are typically mapped to all available system cores and, consequently, there is no room for shared-memory parallelism.

Both `parallel_sort` and `sequential_sort` functions are only simple wrappers. The implementation of sorting algorithms defined in Section 3 is hidden in the `aqsort::impl` namespace and the corresponding codes are stored in the `impl` subdirectory. There is no need for building and linking the AQsort library; its functionality is provided purely in the form of C++ header files.

In Section 3, we introduced two AQsort global algorithm parameters, namely PBS and IST. These parameters are represented within the implementation by preprocessor macros called `AQSORT_PARALLEL_PARTITION_BLOCK_SIZE` and `AQSORT_INSERTION_SORT_THRESHOLD`, respectively. If users want to change their default values for their codes, they need to define these macros before the `aqsort.h` header file is included.

Let us now show how to define comparison and swapping functions for AQsort. Primarily, we can define them either as *function objects*, or as *lambda functions*, which were introduced by the C++11 standard. Both options are illustrated by Figures 4.2 and 4.3, respectively; note that the latter is considerably clearer and more concise. These example codes show how to sort nonzero elements of a sparse matrix A, where:

$$A = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 5 & 0 & 0 & 6 \end{bmatrix} \quad \text{and} \quad n = 6. \tag{4.1}$$

Note that it would have no sense to perform such sorting in parallel; multi-threaded functionality of AQsort was designed for very large arrays. The reasonable condition for efficient parallel sorting with AQsort is

$$n \gg PBS \times numthreads, \tag{4.2}$$

[3] `https://github.com/DanielLangr/AQsort`

```
#ifndef AQSORT_INSERTION_SORT_THRESHOLD
   #define AQSORT_INSERTION_SORT_THRESHOLD 16
#endif

#include "impl/sequential_sort.h"

#ifdef _OPENMP
   #ifndef AQSORT_PARALLEL_PARTITION_BLOCK_SIZE
      #define AQSORT_PARALLEL_PARTITION_BLOCK_SIZE 1024
   #endif

   #include "impl/parallel_sort.h"
#endif

namespace aqsort
{
#ifdef _OPENMP
   template <typename Comp, typename Swap>
   inline void parallel_sort(std::size_t length, Comp* const comp, Swap* const swap)
   {
      impl::parallel_sort(length, comp, swap);
   }
#endif

   template <typename Comp, typename Swap>
   inline void sequential_sort(std::size_t length,
                               Comp* const comp, Swap* const swap)
   {
      impl::sequential_sort(length, comp, swap);
   }

   template <typename Comp, typename Swap>
   inline void sort(std::size_t length, Comp* const comp, Swap* const swap)
   {
#ifdef _OPENMP
      parallel_sort(length, comp, swap);
#else
      sequential_sort(length, comp, swap);
#endif
   }
}
```

Fig. 4.1: Contents of the AQsort main header file `aqsort.h`. Comments and some unimportant preprocessor directives are omitted.

where *numthreads* denotes the number of utilized OpenMP threads.

5. Experiments. We have conducted an extensive experimental study to evaluate the performance and scalability of AQsort and to compare it with existing forefront implementations of parallel sorting algorithms. Measurements were run on modern HPC hardware architectures, namely 3 types of multi-core computational nodes of large-scale HPC systems and a many-core Intel Xeon Phi coprocessor/accelerator; see Table 5.1 for details.

The utilized Intel Xeon processors of the Cray XC40 system supported hyper-threading, i.e., running two threads per a single physical core. However, within preliminary test measurements, hyper-threading brought

[4]Formerly known as Hornet (the system was renamed/upgraded during our work).
[5]High Performance Computing Center Stuttgart, University of Stuttgart, Stuttgart, Germany.
[6]National Center for Supercomputing Applications, University of Illinois, Urbana-Champaign, USA.
[7]Faculty of Information Technology, Czech Technical University in Prague, Prague, Czech Republic.
[8]Jülich Supercomputing Centre, Institute for Advanced Simulation, Jülich, Germany.
[9]Non-uniform memory access.
[10]Symmetric multiprocessing.

```cpp
#include <algorithm>
#include <vector>

#include <aqsort.h>

template <typename T> struct Comp {
   Comp(std::vector<T>& rows, std::vector<T>& cols) : rows_(rows), cols_(cols) { }

   inline bool operator()(std::size_t i, std::size_t j) const {
      // lexicographical ordering:
      if (rows_[i] < rows_[j]) return true;
      if ((rows_[i] == rows_[j]) && (cols_[i] < cols_[j])) return true;
      return false;
   }

   /* private: */ std::vector<T> &rows_, &cols_;
};

template <typename T, typename U> struct Swap {
   Swap(std::vector<T>& rows, std::vector<T>& cols, std::vector<U>& vals)
      : rows_(rows), cols_(cols), vals_(vals) { }

   inline void operator()(std::size_t i, std::size_t j) {
      std::swap(rows_[i], rows_[j]);
      std::swap(cols_[i], cols_[j]);
      std::swap(vals_[i], vals_[j]);
   }

   /* private: */ std::vector<T> &rows_, &cols_;
   /* private: */ std::vector<U> &vals_;
};

int main() {
   typedef unsigned int uint32_t;

   // sparse matrix in COO
   std::vector<uint32_t> rows, cols;
   std::vector<double> vals;

   // matrix assembly in reverese lexicographical order
   rows.push_back(0); cols.push_back(0); vals.push_back(1.0);
   rows.push_back(3); cols.push_back(0); vals.push_back(5.0);
   rows.push_back(1); cols.push_back(1); vals.push_back(3.0);
   rows.push_back(2); cols.push_back(2); vals.push_back(4.0);
   rows.push_back(0); cols.push_back(3); vals.push_back(2.0);
   rows.push_back(3); cols.push_back(3); vals.push_back(6.0);

   // sorting in lexicographical order:
   Comp<uint32_t> comp(rows, cols);
   Swap<uint32_t, double> swap(rows, cols, vals);

   aqsort::sort(rows.size(), &comp, &swap);
   // matrix elements are now sorted lexicographically
}
```

Fig. 4.2: Custom reordering of sparse matrix nonzero elements with AQsort using function objects.

no speedup in comparison with a default single-thread-per-core configuration. The AMD Opteron processors of the Cray XE6 system did not support hyper-threading. Therefore, within the presented study, we did not use more than a single thread per core on these architectures. On the contrary, on IBM BlueGene/Q (BG/Q) nodes and Intel Xeon Phi coprocessors, we usually achieved the highest sorting performance when running multiple threads per a single core. Both IBM BG/Q and Intel Xeon Phi support up to 4 simultaneously running threads

```cpp
#include <cstdint>
#include <utility>
#include <vector>

#include <aqsort.h>

int main() {
   // sparse matrix in COO, matrix elements in reverese lexicographical order
   std::vector<uint32_t> rows { 0, 3, 1, 2, 0, 3 };
   std::vector<uint32_t> cols { 0, 0, 1, 2, 3, 3 };
   std::vector<double> vals { 1.0, 5.0, 3.0, 4.0, 2.0, 6.0 };

   // sorting in lexicographical order:
   auto comp = [&rows, &cols] (std::size_t i, std::size_t j) /* -> bool */ {
      if (rows[i] < rows[j]) return true;
      if ((rows[i] == rows[j]) && (cols[i] < cols[j])) return true;
      return false;
   };

   auto swap = [&rows, &cols, &vals] (std::size_t i, std::size_t j) {
      std::swap(rows[i], rows[j]);
      std::swap(cols[i], cols[j]);
      std::swap(vals[i], vals[j]);
   };

   aqsort::sort(rows.size(), &comp, &swap);
   // matrix elements are now sorted lexicographically
}
```

Fig. 4.3: Custom reordering of sparse matrix nonzero elements with AQsort in C++11 using lambda functions.

Table 5.1: Configurations of HPC systems and their run-time environments used for experiments.

Architecture:	Cray XC40	Cray XE6	Intel Xeon Phi	IBM BlueGene/Q
System:	Hazel Hen[4]	Blue Waters	Star	Juqueen
Provider:	HLRS[5]	NCSA[6]	CTU[7]	JSC[8]
Processor:	Intel Xeon E5-2680 v3	AMD Opteron 6276	Xeon Phi 7120P	IBM PowerPC A2
Frequency:	2.5 GHz	2.3 GHz	1.238 GHz	1.6 GHz
Cores per node:	24	16	61	16
Node memory:	128 GB	64 GB	16 GB	16 GB
Memory access:	NUMA[9]	NUMA	SMP[10]	SMP
Used compilers:	GNU g++ 4.9.2 Intel icpc 15.0.2	GNU g++ 4.8.2 Intel icpc 15.0.3	Intel icpc 15.0.2	GNU g++ 4.8.1

per core, therefore, we present measurements for up to 64 and 244 threads for these architectures, respectively. All the measurements on Intel Xeon Phi were performed in the native mode, i.e., test programs were run directly on the coprocessor.

Please note that we do not provide results of all the measurements for IBM BG/Q, since the Juqueen's runtime environment was not intended for such types of experiments. There was neither a possibility to obtain an interactive access to computational nodes nor a possibility to use a single node only. The smallest allocation unit for a job consisted of 32 nodes and such an allocation had wall clock time limited to 30 minutes. We therefore performed only selected experiments on IBM BG/Q.

Within measurements, we used 3 types of data sets for sorting:

1. *integer numbers* (IN) represented with the 64-bit unsigned integer data type,
2. *binary numbers* (BN) represented with the 8-bit unsigned integer data type,
3. *sparse matrix nonzero elements* (SM) in the COO format represented with the 32-bit unsigned integer indexes and double-precision 64-bit values.

Characteristics of these data sets and experiments performed with them are shown in Table 5.2. Although

Table 5.2: Characteristics of data sets and experiments performed with them.

Data:	IN	BN	SM
Number of arrays:	1	1	3
(Multi)element memory footprint:	8 bytes	1 byte	$2 \cdot 4 + 8 = 16$ bytes
Data generation:	random	random	random
Random distribution:	uniform	uniform	uniform
Initial ordering:	none	none	reverse lexicographical
Final ordering:	natural	natural	lexicographical

AQsort is primarily intended for multi-array sorting, we chose the IN and BN single-array sorting problems for this study since it allowed us to directly compare AQsort with its iterator-based competitors. In case of the SM data sets, such a comparison had to be performed indirectly by running the SoA-to-AoS and back AoS-to-SoA transformations.

For sake of readability, we generally refer to the elements of IN and BN arrays as to multielements, even though they are in fact "single elements" only. For all types of data sets, multielements were generated randomly; we used the implementation of the Mersene Twister pseudorandom number generator [17] provided by C++11 and Boost. In the IN and BN cases, data were sorted directly, which corresponded to sorting of randomly shuffled integer/binary numbers. The matrix nonzero elements (SM) were first sorted in the reverse lexicographical order and the measurements were performed while sorting them in the lexicographical order.

Each particular measurement within this study can be characterized by the following *input parameters*:

1. the algorithm/its implementation used for sorting,
2. the utilized hardware architecture,
3. the type of sorted data (IN, BN, SM),
4. the number of sorted multielements (n),
5. the number of utilized threads.

Due to the randomness in the input data, sorting times generally differ for the same input parameters across different algorithm runs. Let algorithm's *performance stability* denote a degree of its ability to sort data with the same input parameters in the same runtime; we call it *stability* only if the context is clear. (We could quantify stability, e.g., as an inverse of the standard deviation of the sorting time from multiple algorithm runs. Note that the defined stability has nothing to do with sorting algorithms being either stable or unstable in the sense of preserving the order of elements with equal keys. Quicksort, and therefore AQsort as well, in inherently unstable in this sense.) The reported results represent average sorting times from multiple measurements. Typically, we used between 6 and 12 measurements for the same input parameters, depending mostly on the algorithm's stability and required computational resources.

The multielements of the IN, BN, and SM data sets have different memory footprints. Their different counts thus fit memories of the architectures presented by Table 5.1. For instance, $2^{30} \approx 1.07 \cdot 10^9$ SM elements can be stored in a memory of an Intel Xeon Phi coprocessor or an IBM BG/Q node. However, we could not work with such large data sets, since even in-place variants of quicksort require some call stack space to perform the recursion. Moreover, there has to be some amount of memory reserved for system processes. We therefore decided to work within our measurements with data sets of sizes $n = 2^k \cdot 10^8$, where $k \in \mathbb{Z}$. Such a choice allowed us to cover a wide range of data set sizes and also guaranteed enough memory for system processes and the call stack. On Intel Xeon Phi coprocessors and IBM BG/Q nodes, we thus sorted at most $8 \cdot 10^8$ SM multielements. However, AQsort was the only implementation that was able to sort such an amount. If the SoA-to-AoS transformation and/or out-of-place sorting was performed, the maximum number of sorted SM multielements was only $4 \cdot 10^8$.

All the presented results were obtained for default values of the AQsort parameters, i.e., $IST = 16$ and $PBS = 1024$ (see Figure 4.1), if not specified otherwise.

5.1. Evaluation of AQsort. First, we measured the *strong scalability* of AQsort, i.e., the response of the sorting time to a different number of OpenMP threads with constant n. Results of these experiments are shown in Figure 5.1. Due to different memory footprints of multielements, we set different n for different types of sorted data (IN, BN, SM). However, for each particular type, n was fixed for all the considered architectures to allow their mutual comparison with respect to AQsort performance.

Table 5.3: Maximum measured speedup of parallel AQsort with respect to `std::sort` (*left*) and sequential AQsort (*right*).

	Speedup with respect to `std::sort`			Speedup with respect to sequential AQsort		
Architecture	IN	BN	SM	IN	BN	SM
Cray XC40	8.8	20.4	8.8	9.9	6.5	8.8
Cray XE6	5.2	17.6	4.6	6.9	3.8	5.2
Intel Xeon Phi	30.7	16.5	28.1	41.8	4.1	41.3
IBM BG/Q	11.8	36.9	14.3	15.6	5.6	16.2

To illustrate the benefits of parallel sorting, Figure 5.1 also shows sorting times for the sequential `std::sort` function from the C++ Standard Library. Clearly, AQsort reduced these sorting times in all cases considerably. The maximum obtained speedup with respect to `std::sort`, and also with respect to the sequential version of AQsort, are shown in Table 5.3.

The results for the IN and SM data sets are similar—the maximum obtained speedup with respect to `std::sort` falls between about 1/4 and 1/3 of the number of threads. The BN case is specific—the total memory footprint of the data being sorted was the same as in the IN case, but the number of sorted multielements was 8 times higher, which caused much longer sorting times of `std::sort`. On the contrary, sorting times of the sequential version of AQsort were actually lower in the BN case than in the IN case. The speedup of parallel AQsort with respect to sequential AQsort was relatively low in the BN case, especially on Intel Xeon Phi and IBM BG/Q. We attribute this effect to the saturation of the memory subsystems; on all utilized architectures, the number of memory controllers/channels was much lower than the number of their computational cores.

Note that for AMD and Intel CPU-based nodes, the lowest sorting times were always obtained when the number of OpenMP threads equaled the number of node cores (one-thread-per-core configuration). For Intel Xeon Phi and IBM BlueGene/Q, the lowest sorting times mostly occurred for the two-threads-per-core configuration, and, even if not, the differences were minimal. We can thus generally recommend these configurations for AQsort maximum speedup on given architectures.

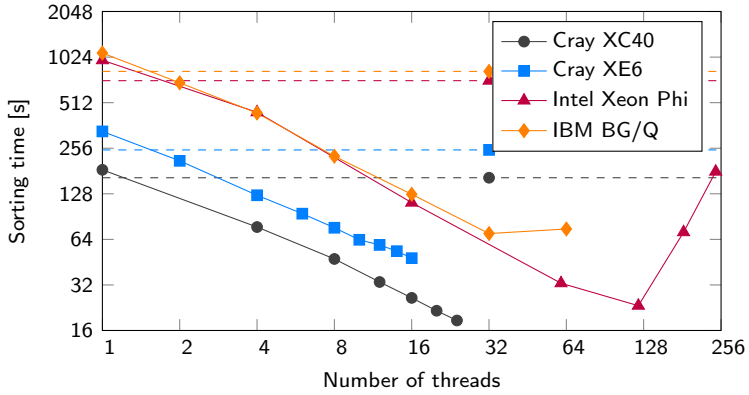
Second, we measured the relation between AQsort sorting times and the number of sorted multielements n; results are shown in Figure 5.2. On all architectures, we performed experiments up to the maximum number of multielements that fitted the available memory. The number of threads was always set to a value that provided highest speedup within the strong scalability results. The results show that the sorting time grew linearly with n and that the rate of this growth was close to 1 in all cases. Such a growth clearly does not correspond to a quicksort complexity $O(n \cdot \log n)$. In auxiliary measurements, we have observed similar linear growth with all the considered implementations of sorting algorithms, including `std::sort`.

The third experiment evaluated the dependence of AQsort sorting time on the algorithm global parameter *PBS*; results are presented in Figure 5.3. They show that, as might have been expected, small block sizes provided low algorithm performance. As the block size grew, the sorting time decreased accordingly. At some point sorting time established and no longer improved with higher block sizes; small variances in sorting times were caused by the instability of the algorithm. Generally, setting a higher block size is seemingly preferable. However, users need to be sure that there is enough data to be sorted efficiently according to condition (4.2).

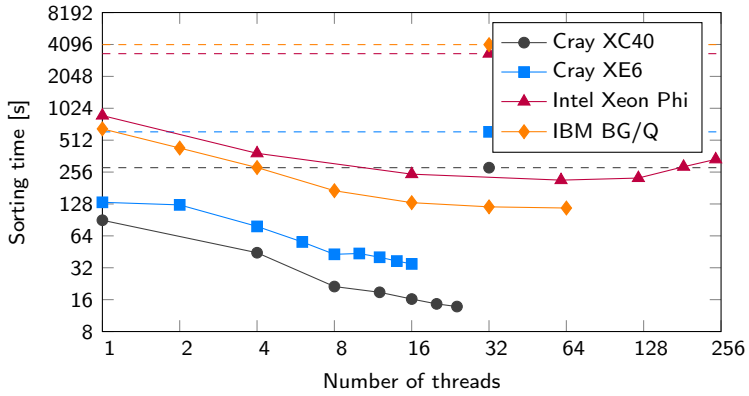
The fourth experiment evaluated the dependence of AQsort sorting time on the second algorithm global parameter *IST*; results are presented in Figure 5.4 (for space reasons, we show the measurements for the IN data sets only). According to these results, users should not set too high values of the threshold for insertion sort; again, small variances in sorting times were caused by the instability of the algorithm.

The last experiment evaluated the stability of AQsort. We performed 200 runs of the algorithm with each particular set of input parameters and processed the results statistically, see Table 5.4. For easier comparison, we show *relative sorting times* that are sorting times in percents normalized by the their average values. Obviously, the sorting times can vary significantly in practice, especially for the BN data sets. However, standard deviations reveal that typical sorting times might be expected much closer to the average than the measured extrema.

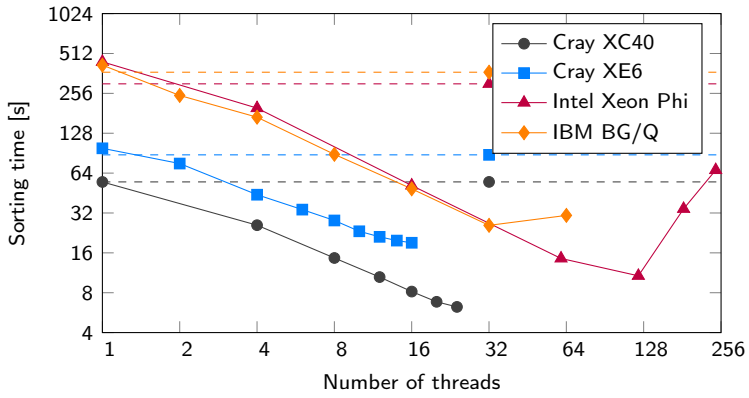
For the above described experiments, we build test programs with the GNU C++ compiler for both AQsort and `std::sort`, with the exception of Intel Xeon Phi, where we used the Intel C++ compiler.



(a) integer numbers (IN), $n = 1.6 \cdot 10^9$

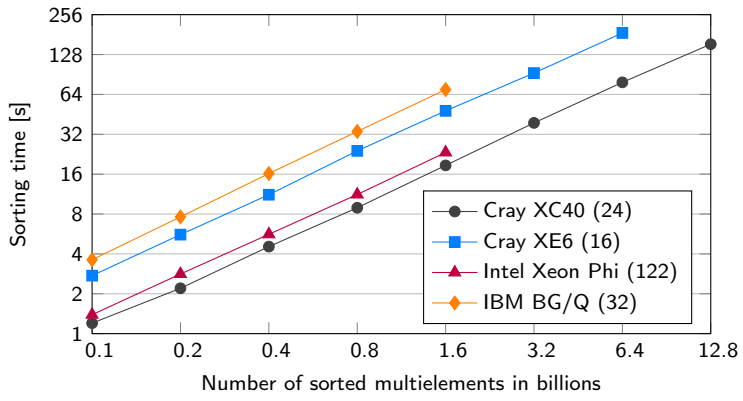


(b) binary numbers (BN), $n = 12.8 \cdot 10^9$

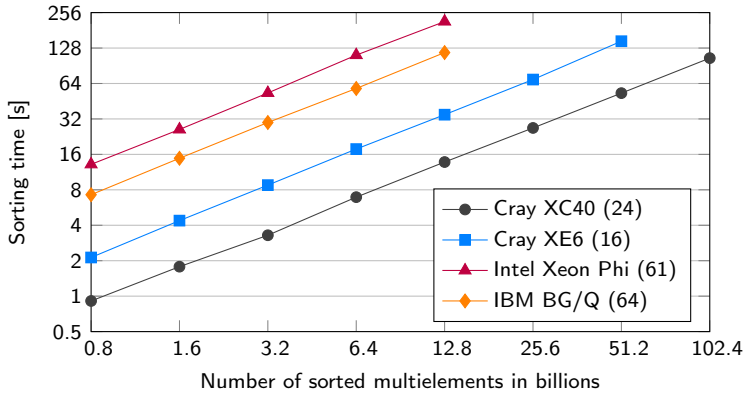


(c) sparse matrix nonzero elements (SM), $n = 0.4 \cdot 10^9$

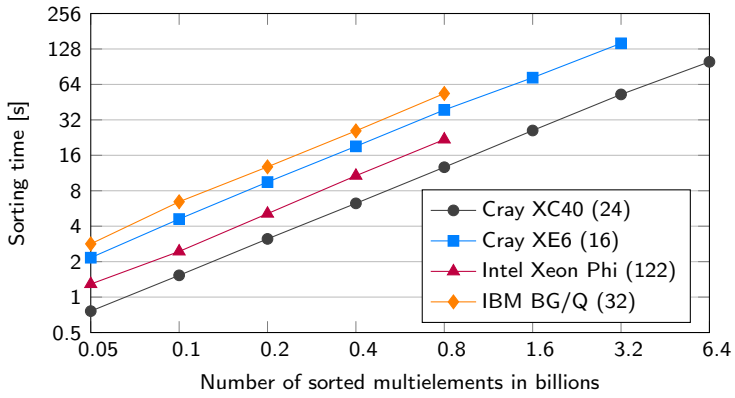
Fig. 5.1: Strong scalability of AQsort measured on different architectures. Dashed lines show the sorting times of the sequential `std::sort` function.



(a) integer numbers (IN)

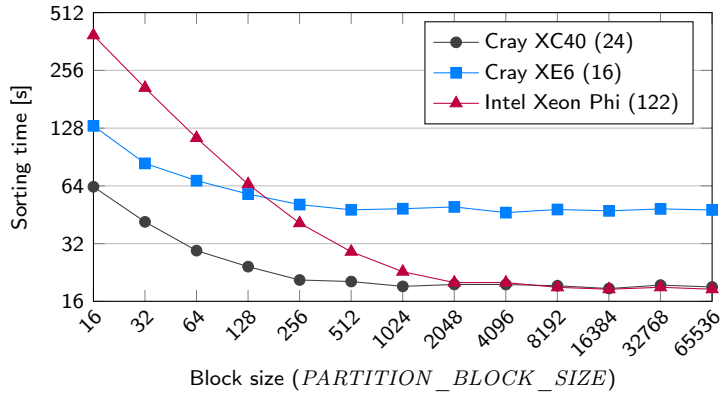


(b) binary numbers (BN)

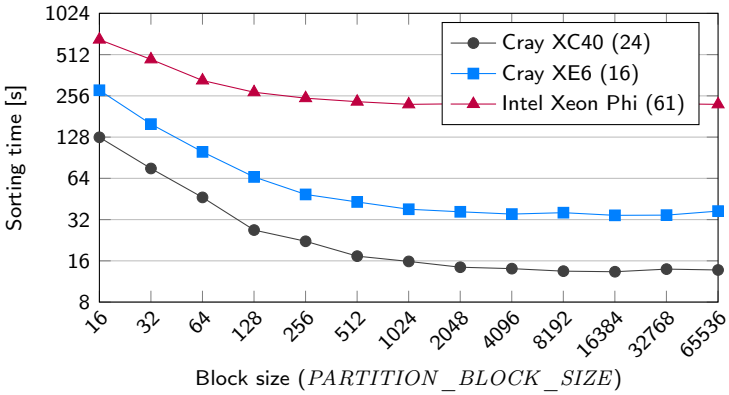


(c) sparse matrix nonzero elements (SM)

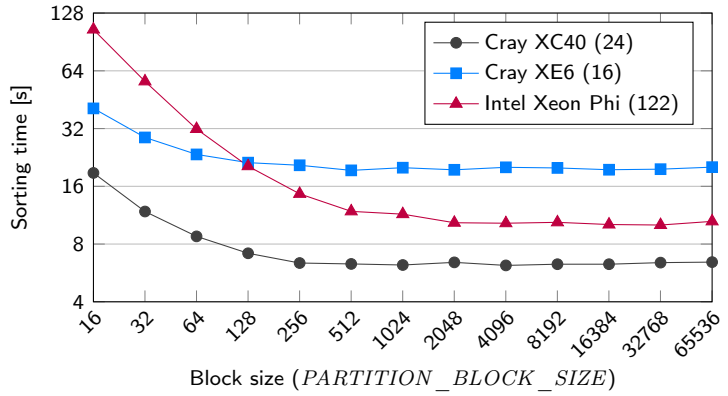
Fig. 5.2: Relation between AQsort sorting times and the number of multilements n, measured on different architectures. Numbers in parentheses in the legend denote the number of utilized OpenMP threads.



(a) integer numbers (IN), $n = 1.6 \cdot 10^9$



(b) binary numbers (BN), $n = 12.8 \cdot 10^9$



(c) sparse matrix nonzero elements (SM), $n = 0.4 \cdot 10^9$

Fig. 5.3: AQsort sorting times for different partitioning block sizes (parameter *PBS*) measured on different architectures. Numbers in parentheses in the legend denote the number of utilized OpenMP threads.

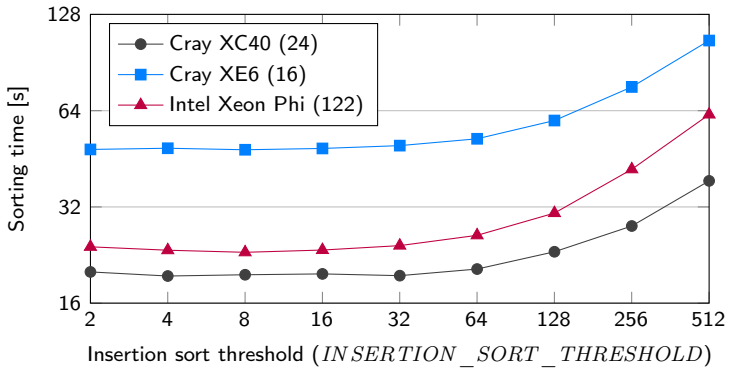


Fig. 5.4: AQsort runtimes for different values of the insertion sort threshold (parameter *IST*) measured with the IN data sets on different architectures. Numbers in parentheses in the legend denote the number of utilized OpenMP threads.

Table 5.4: Statistical values for 200 sorting times of parallel AQsort in percents normalized by their average values. The number in parentheses denote the number of utilized OpenMP threads.

	Cray XC40 (24)			Cray XE6 (16)			Intel Xeon Phi (122)		
Statistics	IN	BN	SM	IN	BN	SM	IN	BN	SM
Minimum	87.0	77.6	90.9	87.8	80.9	88.2	94.0	93.1	92.1
Average	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Maximum	108.4	135.7	108.9	110.6	129.9	112.6	109.6	110.0	116.7
Stddev	4.7	6.6	3.8	4.7	7.9	5.2	2.6	4.8	3.5

5.2. Comparison of AQsort with Modern Implementations of Sorting Algorithms. In this section, we show the results of experiments that were designed to compare the performance of AQsort and modern implementations of sorting algorithms introduced in Section 2. Namely, we considered the following parallel solutions:

1. in-place `std::__parallel::sort(..., quicksort_tag())` function from GNU Libstdc++ Parallel Mode (further referred to as GNU-QS),
2. in-place `std::__parallel::sort(..., balanced_quicksort_tag())` function from GNU Libstdc++ Parallel Mode (GNU-BQS),
3. out-of-place `std::__parallel::sort(..., multiway_mergesort_tag())` function from GNU Libstdc++ Parallel Mode (GNU-MWMS),
4. in-place `tbb::parallel_sort` function from Intel TBB (TBB),
5. in-place `cilkpub::cilk_sort_in_place` function from Cilkpub (CP-IP),
6. out-of-place `cilkpub::cilk_sort` function from Cilkpub (CP-OoP),
7. out-of-place `thrust::sort` function from Nvidia Thrust (Thrust).

The `cilkpub::cilk_sort` function calls `cilkpub::cilk_sort_in_place` if there is not enough memory to perform out-of-place sorting; we avoided this approach within our study. Cp-OoP thus always refer to the out-of-place sorting.

To evaluate the benefits of parallel sorting, we show the sorting times for the sequential `std::sort` function as well. The version of GNU Libstdc++ was given by the version of the GNU C++ compiler, see Table 5.1. As for other libraries, we compiled test programs against Intel TBB version 4.3 Update 4, Cilkpub version 1.05, and Nvidia Thrust version 1.8.0.

AQsort was the only implementation that could sort multiple arrays. To compare its performance with other implementations, we thus either measured runtime of sorting single arrays (the IN and BN cases), or we had to encapsulate sorting with the SoA-to-AoS and back AoS-to-SoA transformations (the SM case). Presented measurements were on all architectures made over the largest possible data sets, i.e., data sets that fitted the half

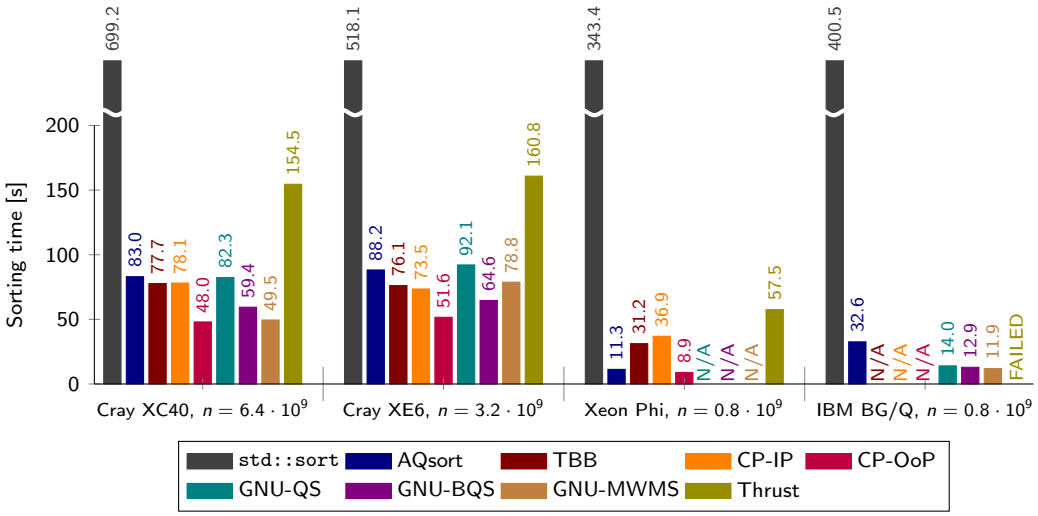


Fig. 5.5: Comparison of sorting times for different implementations of sorting algorithms for IN data sets.

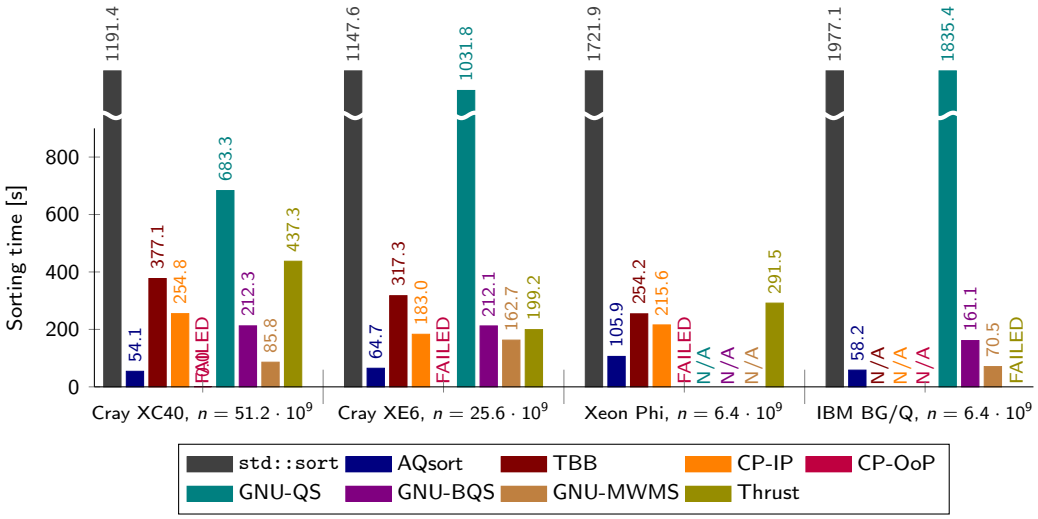


Fig. 5.6: Comparison of sorting times for different implementations of sorting algorithms for BN data sets.

of the available memory because of either out-of-place sorting and/or the SoA-to-AoS transformation. When both the SoA-to-AoS transformation and out-of-place sorting took place, the memory of the original arrays had to be released before sorting and reallocated afterwards.

All the mentioned implementations were available in the runtime environments of Cray XC40 and Cray XE6 nodes. The runtime environment of available Intel Xeon Phi accelerator did not contain the GNU Libstdc++ Parallel Mode functionality (GNU-QS, GNU-BQS, GNU-MWMS). The runtime environment of IBM BG/Q did not contain solutions provided by Intel (TBB, CP-IP, CP-OoP).

For experiments run on Intel Xeon Phi, we used exclusively the Intel C++ compiler. Otherwise, we used the GNU C++ compiler for `std::sort`, AQsort, GNU-QS, GNU-BQS, GNU-MWMS, and Thrust; and the Intel C++ compiler for TBB, CP-IP, and CP-OoP.

The obtained results are shown in Figures 5.5–5.7. The "N/A" labels denote an absence of a given implementation on a given architecture. The "FAILED" labels denote measurements that were repeatedly terminated due to runtime errors, typically segmentation faults; we have not performed analyses of these errors. Based on these results, we can make the following observations:

1. All the parallel solutions considerably reduced sorting times in comparison with sequential `std::sort`.

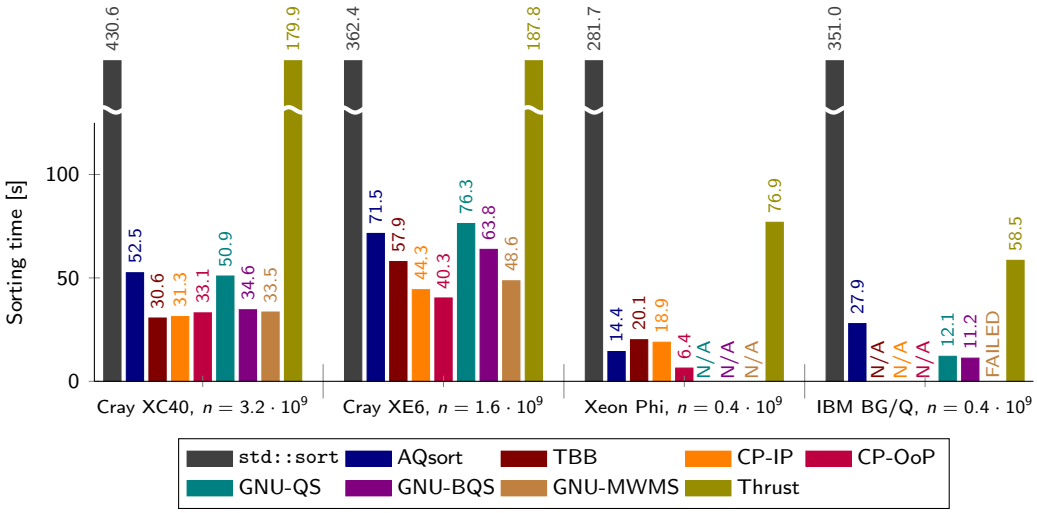


Fig. 5.7: Comparison of sorting times for different implementations of sorting algorithms for SM data sets.

An exception was only GNU-QS in combination with the BN data sets.

2. Thrust performed poorly in comparison with other parallel solutions. Recall that though Thrust supports OpenMP, it was primarily designed as a library for Nvidia GPU accelerators. It thus seems that the optimization level for OpenMP is in Thrust relatively low, at least for its sorting functionality.
3. The other out-of-place sorting implementations, i.e., CP-OoP and GNU-MWMS, mostly, but not always, provided the lowest sorting times.
4. GNU-BQS always outperformed GNU-QS.
5. AQsort outperformed for BN data sets all other implementations—even the out-of-place ones—on all architectures.
6. For IN and SM data sets, AQsort run slowly on IBM BG/Q in comparison with GNU-QS, GNU-BQS, and GNU-MWMS. This might have been caused by not-so-well optimized OpenMP environment for task-based parallelism on this architecture; AQsort was the only implementation build upon OpenMP tasks.
7. AQsort provided lowest sorting times of all in-place sorting implementations on Intel Xeon Phi.
8. For IN data sets and Cray architectures, AQsort performed slightly worse than its in-place competitors (TBB, CP-IP, GNU-QS, and GNU-BQS).
9. For SM data sets, AQsort provided higher sorting times than most of other implementations. However, these sorting times do not include the overhead given by the SoA-to-AoS and back AoS-to-SoA transformations; see Section 5.3 for details.

5.3. SoA-to-AoS and AoS-to-SoA Transformations. The reason why we did not include the SoA-to-AoS and AoS-to-SoA transformations in measurements presented by Figure 5.7 was that the performance and efficiency of theses transformations are highly implementation-dependent. For illustration, consider the following C++ solution for the SoA-to-AoS transformation that we used within our codes:

```
// sparse matrix in COO
std::vector<uint32_t> rows, cols;
std::vector<double> vals;
...
struct element_t {
   uint32_t row, col;
   double val;
};
const std::size_t n = rows.size();
std::vector<element_t> elements(n);
#pragma omp parallel for
for (std::size_t k = 0; k < n; k++) { elements[k].row = rows[k]; ... }
```

Profiling of this code on a Cray XC40 node for $n = 3.2 \cdot 10^9$ and 24 threads revealed that the parallel `for` loop took only 1.58 seconds, while the initialization of the `elements` vector took 23.15 seconds (subsequent sorting with GNU-MWMS took 32.75 seconds). A similar problem arose in the case of out-of-place sorting, where memory occupied by the `rows`, `cols`, and `vals` arrays had to be released prior to sorting and reallocated afterwards. This consisted of 3 calls of the `std::vector::resize` function, which took in this experiment 24.61 seconds, while the final parallel AoS-to-SoA transformation took only 2.13 seconds. A detailed analysis of this problem is beyond the scope of this article. Let us just note that it is caused by so-called *zero-initialization* of elements during resizing of C++ `std::vector`s, which is inherently carried out sequentially.

6. Discussion. A straightforward conclusion of our work is that though speedups are far from being linear, parallel/multi-threaded sorting considerably reduces sorting times on modern multi-core and many-core hardware architectures. The question that remains to be answered is which implementation to choose for particular sorting problems.

If we need to sort multiple arrays at once and suppose that there might not be enough available memory for the SoA-to-AoS transformation, then AQsort is the only option of all the considered solutions. This might happen, e.g., when developing scientific or engineering applications where multi-array sorting is required for possibly very large data, such as large sparse matrices or discretization meshes. In such cases, we do not want to limit the sizes of users' problems being solved just because of sorting. AQsort is the only implementation that does not need to run the SoA-to-AoS transformation. Therefore, it is the only implementation that can be straightforwardly applied to multiple arrays large enough to fill more than the half of the available memory.

On the contrary, if we certainly know that there is enough memory available, there will be no simple answer. Generic sorting of a single array with a lot of distinct sorting keys would be likely most efficient with some parallel out-of-place implementation. However, one might need to implement the SoA-to-AoS and back AoS-to-SoA transformations carefully to prevent its large runtime overhead.

Another aspects that play a crucial role in the selection of sorting implementation are threading paradigms and portability. Generally, OpenMP prevails in the domains of HPC, mathematical, scientific, and engineering software. Of all the considered implementations, only AQsort and Nvidia Thrust provide portable OpenMP solutions. GNU Libstdc++ Parallel Mode implementations are tightly bound to the GNU C++ compiler; except of this compiler, we succeeded in compiling programs that called the GNU Libstdc++ Parallel Mode sorting functions only with the newest versions of the Intel C++ compiler, namely versions 14 and 15. PGI, Cray, IBM, and older versions of Intel C++ compilers failed in such a compilation. The portability of sorting implementations based on Intel TBB and Intel Cilk Plus is relatively low in comparison with OpenMP. Moreover, though it is technically possible to integrate TBB or Cilk Plus routines into OpenMP code, such a combination of threading paradigms might result in malformed parallelism; we encountered such a behavior in the Cray runtime environments. Generally, it is not recommended to combine multiple threading paradigms within a single code.

Moreover, the interface of AQsort allows its integration with C and Fortran codes through explicit instantiation of its sorting functions with fixed footprints of comparison and swapping functions. Such an option is not feasible with other iterator-based sorting solutions.

AQsort generally seems to provide somewhat longer sorting times when compared with the sorting implementation that is most suitable for each particular case. We attribute this fact to the AQsort's universality represented by the usage of user-provided compare and swap functions. Recall that within AQsort implementation, we cannot directly touch sorted data. We therefore cannot optimize and tune the code with respect to cache subsystems, which is a common practice, e.g., in the form of software prefetching, vectorization, or accessing data with respect to their alignment. Or, for instance, we cannot store a pivot during partitioning in a temporary variable, which might cause its storage in registers during runtime for suitable types of data. Moreover, calling compare and swap functions via pointers might generally hinder many optimizations carried out by compilers on a regular basis. Users/developers therefore need to decide themselves whether they require maximum sorting performance with all that restrictions given by current implementations or whether they are willing to abandon a bit of performance for the possibility to work with larger data sets in a fully portable way.

7. Conclusions. The contribution of this article is a new parallel/multi-threaded version of the quicksort sorting algorithm and its implementation called AQsort. This implementation, written in C++ and using OpenMP, is available in the form of an open-source software project. It is primarily intended to be integrated into scientific and engineering HPC codes that operate in modern multi-core and many-core runtime environments

where OpenMP parallel paradigm is dominant.

AQsort fills some gaps in modern implementations of parallel sorting algorithms. Moreover, it provides also sequential sorting functionality that can be exploited either in sequential codes or in HPC codes based on pure MPI parallelism.

The main features of AQsort are:

1. *Generality*—it works with user-provided functions for comparing and swapping sorted data. Consequently, in contrast to pointer-based/iterator-based sorting functions, it can directly sort multiple arrays at once (see Section 3).
2. *Space-efficiency*—it operates in place and for multiple arrays it does not require the SoA-to-AoS and back AoS-to-SoA transformations. Therefore, it is the only implementation that can be straighforwardly applied to multiple arrays large enough to fill more than the half of the available memory (see Section 1).
3. *Scalability*—it considerably reduces sorting time with respect to optimized implementations of sequential sorting algorithms and it efficiently utilizes all the cores of modern multi-core and many-core hardware architectures (see Section 5.1).
4. *Portability*—it is implemented with standard OpenMP pragmas and functions. It is build upon the combination of nested parallelism and tasking, which are supported by majority of modern C++ compilers (see Section 4). Moreover, AQsort allows to create wrappers for C and Fortran programming languages. Of all the considered parallel implementations, only AQsort and Thrust were available on all the tested architectures. In comparison with AQsort, Thrust provided longer runtimes in all measurements and required more memory due to the implementation of an out-of-place sorting algorithm (see Section 5.2).
5. *Efficiency*—its performance is generally comparable with modern implementations of sorting algorithms when running on forefront HPC hardware architectures (see Section 5.2). It seems to be especially suitable for Xeon Phi coprocessors and for data that contain only few distinct sorting keys. In other cases, AQsort might be outperformed by its competitors, which is a price for its universality (see Section 6).

In addition to the presentation of AQsort, this article also:

1. serves as a brief updated survey of existing implementations of parallel sorting algorithms and their experimental comparison on leading-edge shared-memory hardware architectures (see Sections 2 and 5.2),
2. presents a parallel partitioning algorithm briefly proposed by Pasetto and Akhriev [23] in terms of detailed pseudocode (see Section 3),
3. generally evaluates the universal approach to parallel sorting that uses custom compare and swap functions with traditional solutions (see Section 6).

Acknowledgements. The authors would like to thank T. Dytrych of the Louisiana State University for providing an access to the Blue Waters system. The authors would like to thank M. Václavík of the Czech Technical University in Prague for providing an access to the Star university cluster. The authors acknowledges support from M. Pajr of CQK Holding and International HPC Initiative.

We acknowledge PRACE for awarding us access to resource JUQUEEN based in Germany at the Gauss Center for Supercomputing. We acknowledge PRACE for awarding us access to resource Hornet based in Germany at the High Performance Computing Center Stuttgart. This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

REFERENCES

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, PA, 2nd ed., 1994.

[2] F. Beekhof, *OMPTL: OpenMP multi-threaded template library*, 2012. Accessed July 17, 2015 at `http://tech.unige.ch/omptl/`.

[3] N. Bell and J. Hoberock, *Thrust: A productivity-oriented library for CUDA*, in GPU Computing GEMs: Jade Edition, W.-M. Hwu, ed., Morgan Kaufmann, 2011.

[4] J. L. Bentley and M. D. McIlroy, *Engineering a sort function*, Software: Practice and Experience, 23 (1993), pp. 1249–1265.

[5] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*, The MIT Press, 2007.
[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, The MIT Press, Cambridge, Massachusetts, USA, 3rd ed., 2009.
[7] R. Dementiev, L. Kettner, and P. Sanders, *STXXL: standard template library for XXL data sets*, Software: Practice and Experience, 38 (2008), pp. 589–637.
[8] C. Hoare, *Algorithm 64: Quicksort*, Communications of the ACM, 4 (1961), pp. 321–322.
[9] ———, *Quicksort*, The Computer Journal, 5 (1962), pp. 10–16.
[10] ISO/IEC, *ISO/IEC 9899:1999: Information Technology — Programming languages — C*, 1999.
[11] ———, *ISO/IEC 14882:2011: Information Technology — Programming languages — C++*, 2011.
[12] D. Langr, I. Šimeček, and T. Dytrych, *Block iterators for sparse matrices*, in Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2016), IEEE Xplore Digital Library, 2016. Accepted for publication.
[13] D. Langr, I. Šimeček, P. Tvrdík, T. Dytrych, and J. P. Draayer, *Adaptive-blocking hierarchical storage format for sparse matrices*, in Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2012), IEEE Xplore Digital Library, 2012, pp. 545–551.
[14] D. Langr and P. Tvrdík, *Evaluation criteria for sparse matrix storage formats*, IEEE Transactions on Parallel and Distributed Systems, 27 (2016), pp. 428–440.
[15] B. A. Mahafzah, *Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture*, The Journal of Supercomputing, 66 (2013), pp. 339–363.
[16] D. Man, Y. Ito, and K. Nakano, *An efficient parallel sorting compatible with the standard qsort*, International Journal of Foundations of Computer Science, 22 (2011), pp. 1057–1071.
[17] M. Matsumoto and T. Nishimura, *Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator*, ACM Transactions on Modeling and Computer Simultaion, 8 (1998), pp. 3–30.
[18] M. McCool, R. A. D., and R. James, *Structured Parallel Programming: Patterns for Efficient Computation*, Morgan Kaufmann, 2012.
[19] Microsoft, *Parallel STL*, 2014. Accessed July 17, 2015 at `https://parallelstl.codeplex.com/`.
[20] D. R. Musser, *Introspective sorting and selection algorithms*, Software: Practice and Experience, 27 (1997), pp. 983–993.
[21] R. Nair, *ParallelSort: Parallel sorting algorithm using OpenMP*, 2014. Accessed July 17, 2015 at `https://github.com/rsnair2/ParallelSort`.
[22] D. Pasetto and A. Akhriev, *A comparative study of parallel sort algorithms*, in Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, OOPSLA '11, New York, NY, USA, 2011, ACM, pp. 203–204.
[23] ———, *A comparative study of parallel sort algorithms*. Accessed July 17, 2015 at http://researcher.watson.ibm.com/files/ie-albert_akhriev/sort2011-full.pdf, 2011.
[24] J. Reinders, *Intel Threading Building Blocks*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.
[25] A. D. Robinson, *A parallel stable sort using C++11 for TBB, Cilk Plus, and OpenMP*, 2014. Accessed July 17, 2015 at `https://software.intel.com/en-us/articles/a-parallel-stable-sort-using-c11-for-tbb-cilk-plus-and-openmp`.
[26] A. Robison, *Composable parallel patterns with Intel Cilk Plus*, Computing in Science Engineering, 15 (2013), pp. 66–71.
[27] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd ed., 2003.
[28] B. Schäling, *The Boost C++ Libraries*, XML Press, 2nd ed., 2014.
[29] I. Šimeček and D. Langr, *Efficient parallel evaluation of block properties of sparse matrices*, in Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2016), IEEE Xplore Digital Library, 2016. Accepted for publication.
[30] J. Singler and B. Konsik, *The gnu libstdc++ parallel mode: Software engineering considerations*, in Proceedings of the 1st International Workshop on Multicore Software Engineering, IWMSE '08, New York, NY, USA, 2008, ACM, pp. 15–22.
[31] J. Singler, P. Sanders, and F. Putze, *MCSTL: The multi-core standard template library*, in Proceedings of the Euro-Par 2007 Parallel Processing, vol. 4641 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 682–694.
[32] M. Süs and C. Leopold, *A user's experience with parallel sorting and OpenMP*, in Proceedings of the Sixth European Workshop on OpenMP, EWOMP'04, 2004, pp. 23–28.
[33] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger, *A framework for adaptive algorithm selection in STAPL*, in Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '05, New York, NY, USA, 2005, ACM, pp. 277–288.
[34] P. Tsigas and Y. Zhang, *A simple, fast parallel implementation of Quicksort and its performance evaluation on SUN Enterprise 10000*, in Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2003), 2003, pp. 372–381.
[35] S. Williams, A. Waterman, and D. Patterson, *Roofline: An insightful visual performance model for multicore architectures*, Commun. ACM, 52 (2009), pp. 65–76.

Edited by: Dana Petcu
Received: July 22, 2016
Accepted: August 31, 2016

AIMS AND SCOPE

The area of scalable computing has matured and reached a point where new issues and trends require a professional forum. SCPE will provide this avenue by publishing original refereed papers that address the present as well as the future of parallel and distributed computing. The journal will focus on algorithm development, implementation and execution on real-world parallel architectures, and application of parallel and distributed computing to the solution of real-life problems. Of particular interest are:

Expressiveness:

- high level languages,
- object oriented techniques,
- compiler technology for parallel computing,
- implementation techniques and their efficiency.

System engineering:

- programming environments,
- debugging tools,
- software libraries.

Performance:

- performance measurement: metrics, evaluation, visualization,
- performance improvement: resource allocation and scheduling, I/O, network throughput.

Applications:

- database,
- control systems,
- embedded systems,
- fault tolerance,
- industrial and business,
- real-time,
- scientific computing,
- visualization.

Future:

- limitations of current approaches,
- engineering trends and their consequences,
- novel parallel architectures.

Taking into account the extremely rapid pace of changes in the field SCPE is committed to fast turnaround of papers and a short publication time of accepted papers.

INSTRUCTIONS FOR CONTRIBUTORS

Proposals of Special Issues should be submitted to the editor-in-chief.

The language of the journal is English. SCPE publishes three categories of papers: overview papers, research papers and short communications. Electronic submissions are preferred. Overview papers and short communications should be submitted to the editor-in-chief. Research papers should be submitted to the editor whose research interests match the subject of the paper most closely. The list of editors' research interests can be found at the journal WWW site (`http://www.scpe.org`). Each paper appropriate to the journal will be refereed by a minimum of two referees.

There is no a priori limit on the length of overview papers. Research papers should be limited to approximately 20 pages, while short communications should not exceed 5 pages. A 50–100 word abstract should be included.

Upon acceptance the authors will be asked to transfer copyright of the article to the publisher. The authors will be required to prepare the text in LaTeX 2ε using the journal document class file (based on the SIAM's `siamltex.clo` document class, available at the journal WWW site). Figures must be prepared in encapsulated PostScript and appropriately incorporated into the text. The bibliography should be formatted using the SIAM convention. Detailed instructions for the Authors are available on the SCPE WWW site at `http://www.scpe.org`.

Contributions are accepted for review on the understanding that the same work has not been published and that it is not being considered for publication elsewhere. Technical reports can be submitted. Substantially revised versions of papers published in not easily accessible conference proceedings can also be submitted. The editor-in-chief should be notified at the time of submission and the author is responsible for obtaining the necessary copyright releases for all copyrighted material.