Co-funded by the European Commission within the Seventh Framework Programme

Project no. 318521

HARNESS

Specific Targeted Research Project
HARDWARE- AND NETWORK-ENHANCED SOFTWARE SYSTEMS FOR CLOUD COMPUTING

http://www.harness-project.eu/

Performance Modeling Report

D6.2

Due date: 30 September 2014
Submission date: 30 September 2014
Resubmission date: N/A

Start date of project: 1 October 2012

Document type: Deliverable
Activity: RTD
Work package: WP6

Editor: Guillaume Pierre (UR1)

Contributing partners: IMP, UR1

Reviewers: Carmelo Ragusa and Mark Stillwell

Dissemination Level

PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Description
0.1	2014/06/30	Guillaume Pierre	UR1	Outline
0.2	2014/08/28	Guillaume Pierre	UR1	Improved outline
0.3	2014/09/03	Guillaume Pierre	UR1	First version of Chapters 1 and 3
0.4	2014/09/09	Anca Iordache	UR1	Improved Chapter 4
0.5	2014/09/09	Alexandros Koliousis	IMP	Initial version of Chapter 5
0.6	2014/09/12	Guillaume Pierre	UR1	Initial version of state of the art
0.7	2014/09/14	Alexandros Koliousis	UR1	Chapter 5
0.8	2014/09/15	Alexandros Koliousis	IMP	Improved state of the art
0.9	2014/09/18	Guillaume Pierre	UR1	Applied internal review comments
0.10	2014/09/22	Anca Iordache	UR1	Applied internal review comments
0.11	2014/09/22	Alexandros Koliousis	IMP	Applied internal review comments
1.0	2014/09/23	Guillaume Pierre	UR1	Final Editor review
1.1	2014/09/29	Alexander Wolf	IMP	Final Coordinator review

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T6.2	Design/develop resource allocator and application performance predictor	UR1*, EPL, ZIB, MAX
T6.4	Devise software engineering methodology and architectures	IMP*, EPL, UR1, ZIB, MAX

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Executive summary

The goal of this deliverable is to present the performance modeling and resource allocation algorithms used by the *Hardware- and Network-Enhanced Software Systems for Cloud Computing* (HARNESS) platform, and developed in Task T6.2.

During Year 2 of the project we have developed two complementary performance modeling approaches: a "blackbox" modeling technique which applies to arbitrary cloud applications, and a specific "whitebox" modeling technique for MapReduce applications. The blackbox approach is very generic but potentially resource-intensive; the whitebox approach uses much less resources but it applies only to specific types of applications. We plan to integrate both techniques in a single performance modeling component during Year 3.

The main achievements this year are:

- The development of a "blackbox" performance modeling approach which automatically builds performance models for arbitrary batch applications in heterogeneous clouds.
- The development of a "whitebox" performance modeling approach which significantly speeds up the profiling process for MapReduce applications.
- The integration of the blackbox performance modeling approach in the HARNESS *Application Manager* (AM) implementation.
- The design of an integrated blackbox/whitebox performance modeling approach which aims to combine the benefits of both techniques.

In addition, Section 5.4 also presents our initial findings realized as part of Task T6.4 (*"Devise software engineering methodology and architectures,"* which started at M19). We propose to extend the scope of cloud applications targeted by the HARNESS project toward stream-processing applications. A full report on this task is scheduled for M36 in Deliverable D6.4.

Contents

Acronyms

AM *Application Manager*. i, 1, 7, 9–11, 14, 17, 21, 34

API *application programming interface*. 26, 28

CPU *central processing unit*. 4, 5, 12, 14, 17, 23–30, 32–36, 38–40, 43, 44

CRS *Cross-Resource Scheduler*. 10

CUDA *compute unified device architecture*. 5

DM *delta merge*. 17, 18

DRAM *dynamic random-access memory*. 29, 32, 35–37

EC2 *Amazon Elastic Compute Cloud*. 1, 4, 7, 18

FPGA *field-programmable gate array*. 5, 12, 23–25, 28, 29, 32–40

GPGPU *general-purpose graphics processing unit*. 5, 12, 24–30, 43, 44

HARNESS *Hardware- and Network-Enhanced Software Systems for Cloud Computing*. i, 1, 2, 7, 9, 17, 21, 23, 24, 28, 41, 45

HetMR *Heterogeneous MapReduce*. 4, 5, 24, 33, 34, 40

HPC *high-performance computing*. 3

IMP *Imperial College London*. 26

OCCI *open cloud computing interface*. 7, 17

OpenCL *Open Computing Language*. 26

PCI *peripheral component interconnect*. 29, 40

PCIe *peripheral component interconnect express*. 26, 29, 32, 33, 36, 38, 40

RTM *reverse time migration*. 13, 17–21

SLO *service-level objective*. 8–10, 14, 17, 19, 20

VM *virtual machine*. 3, 7, 17, 18

1 Introduction

Cloud computing offers unprecedented levels of flexibility to efficiently deploy demanding applications. Thanks to its support of heterogeneous hardware devices, *Hardware- and Network-Enhanced Software Systems for Cloud Computing* (HARNESS) provides access to a large variety of computing resources with various combinations of configurations and prices. In principle this should allow HARNESS users to make use of the exact types and numbers of resources their applications need. However, this flexibility also comes as a "curse" as choosing the best resource configuration for an application becomes extremely difficult: cloud users can easily under- or overestimate the requirements of complex applications.

Existing cloud platforms offer very little support to help users choose appropriate resources to address their particular needs. Autoscaling systems, such as *Amazon AutoScale*, merely allow users to define their own rules, but without guiding them in writing useful rules. Web autoscaling systems usually choose a single instance type and vary only the number of instances when the load changes. Such techniques are however not well suited for batch applications which often cannot adjust their choice of resources during execution.

Selecting the "right" set of resources for a batch application requires a fine-grained understanding of the relationship between a set of resources and the performance the application will have using these resources. This is challenging because the space of all possible resource configurations one may choose from is usually extremely large. For example, the *Amazon Elastic Compute Cloud* (EC2) currently proposes 23 different instance types. An application requiring just five nodes must therefore choose one out of $23^5 = 6,436,343$ possible configurations. In HARNESS the number of possible configuration would be even greater, as the system allows for fine-grained specification of the resources a user may request (as opposed to the fixed set of choices in Amazon EC2) and supports a large range of heterogeneous devices likely to offer different price/performance trade-offs.

It should be noted that, in a cloud computing environment, achieving the lowest possible execution time for a batch application is not always desirable, as the fastest execution often requires using expensive resources. Depending on the circumstances, a user may want to choose the fastest configuration, the cheapest, or any configuration implementing a trade-off between these two extremes.

An important goal of the HARNESS platform is to automate the choice of resources to be given to an application. With such a system, a user may merely need to indicate her *expectations* in terms of execution duration or financial cost, and let the platform automatically choose the set of resources that best satisfy these constraints.

From the platform's point of view, automating the resource selection on behalf of arbitrary applications submitted by the users is a difficult challenge. Deliverable D6.1 presented the way arbitrary applications can be described in an uniform manner in a way that a generic *Application Manager* (AM) can handle them seamlessly [30]. This deliverable, in turn, presents our approach for automatically modeling the performance of cloud applications in the presence of a very large variety of possible resource configurations.

Our performance modeling approach exploits the assumption that the same application is likely to be executed numerous times, albeit possibly with different input data each time. Upon the first few executions

of the application, the system tries various resource configurations and builds a custom performance model for this application. Once a model has been built, cloud users can simply specify the execution time or the financial cost they are ready to tolerate for each execution, and let the system automatically find the resource configuration which best satisfies their constraints.

We propose two complementary techniques for efficiently building accurate performance models. First, a "blackbox" method can build a performance model of arbitrary applications. This approach is very general, but on the other hand it may require a significant number of profiling executions before capturing all the relevant specificities of a new application. We therefore propose to extend it with one or more "whitebox" performance modeling components. Each such component may be specialized for a particular class of application, for example MapReduce applications. Whitebox performance modeling may be faster than blackbox modeling in identifying key information thanks to its domain-specific knowledge of a specific class of applications. When a whitebox component can be applied to a user's application, it can thus considerably speed up the blackbox performance modeling process.

In addition, Section 5.4 also presents our initial findings realized as part of Task T6.4 (*"Devise software engineering methodology and architectures,"* which started at M19). We propose to extend the scope of cloud applications targeted by the HARNESS project toward stream-processing applications. This type of applications is currently receiving lots of attention by the Cloud community as it allows users to get timely results from high-throughput input data streams. This work has started just a few months ago. A full report on this task is scheduled for M36 in Deliverable D6.4.

The remainder of this deliverable is organized as follows: Chapter 2 presents the relevant state of the art. Chapter 3 discusses the system model and our approach to combine blackbox and whitebox performance modeling. Then Chapter 4 presents our technique for blackbox performance modeling, Chapter 5 presents a whitebox performance modeling specialized for MapReduce applications, and Chapter 6 concludes.

2 State of the art

Modeling the performance of computer systems is the subject of a very abundant and varied literature. We discuss the differences between performance modeling in *high-performance computing* (HPC) environments vs. cloud computing environments, then survey the different types of techniques used and specific efforts related to performance modeling in heterogeneous environments.

2.1 HPC vs. cloud performance modeling

HPC aims at executing applications as fast as possible, in order to optimize a variety of metrics such as the makespan of a set of jobs,high throughput, and low average stretch. In consequence, performance modeling has always been a priority concern in this area [3]. HPC environments usually consist of large supercomputers where users have direct access to the bare-metal machine. This is useful for getting the best possible performance, and it also helps performance modeling because the computing resources often expose their detailed hardware configuration.

In Cloud environment, however, the use of virtualization means that users do not have access to the bare-metal machine. Instead, they only get access to *virtual machine* (VM) instances with little knowledge about the underlying hardware configuration. Virtualization also greatly reduces the extent to which a user may control the provisioned resources, and consolidation exposes users to performance interferences between multiple VM instances [14, 24, 42]. In consequence, precise performance modeling is more challenging in Cloud than in HPC environments.

A side effect of the flexibility and public availability of the cloud is that cloud applications are extremely diverse in their functionalities, complexity, and implementation maturity level. In HPC environments, users are usually able and willing to dedicate significant efforts for improving the performance of a single application, whereas in Cloud environments many users may prefer dedicating less efforts to fine-tuning and rather turn to automated performance modeling processes.

2.2 Performance modeling approaches

Modeling techniques can be classified into analytical predictive methods, code analysis and profiling [3, 46, 53]. Analytical methods require developers to provide a model of their application. They are potentially very accurate, but building analytical models is labor-intensive and difficult to automate. Moreover, user estimates of application runtimes are often highly inaccurate [62]. Code analysis automates this process, but it usually restricts itself to coarse-grained decisions such as the choice of the best acceleration device for optimizing performance [12].

Another approach for fully-automated performance modeling, similar to the blackbox modeling work presented in Chapter 4, requires training artificial neural networks using performance results from application executions on a target platform [39]. However, they use this technique to study the impact of

varying the input size while we are mostly interested in the consequences of different resource selection choices.

In order to understand the resource requirements of arbitrary applications, Amazon EC2 recommends to empirically try a variety of instance types and choose the one which works best [4]. CopperEgg automates this process by monitoring the resource usage of an arbitrary application over a 24-hour period before suggesting an appropriate instance type to support this workload [22]. However, as we shall see in Chapter 4, utilization-based methodologies do not necessarily lead to optimal results. Besides, CopperEgg does not allow the user to choose her prefered optimization criterion. Our work, in contrast, aims at finding Pareto-optimal configurations for arbitrary applications, and it supports the automatic selection of resources which match a given optimization criterion.

2.3 Performance modeling in heterogeneous clouds

Few efforts have been dedicated so far to blackbox modeling in heterogeneous environments. For example, one can observe the statistical distribution of task execution times in a bag-of-tasks, and automatically derive task scheduling strategies to execute the bag under certain time and cost constraints [51]. However, most research in this area has focused on whitebox approaches, as we discuss next.

Web Applications. Whitebox performance modeling in heterogeneous cloud environment has been studied in the context of online (Web) applications. Besides the numerous techniques which dynamically vary the number of identically-configured resources to follow the request workload, one can use machine learning techniques over historical traces in order to define horizontal and vertical scaling rules to handle various types of workloads [63]. Similarly, when scaling decisions are necessary, one may dynamically choose the best resource type based on short-term traffic predictions [27]. Some other works exploit the fact that identically-configured cloud resources often exhibit heterogeneous performance [24]. For instance, one can benchmark the performance of each individual virtual machine instance before deciding how this new resource can best be used in the application [25, 26].

Heterogeneous MapReduce. As *central processing unit* (CPU) heterogeneity emerged in today's computing environments, so did performance imbalances in data-parallel processing frameworks. In this context, one of the most widely studied frameworks in the literature is MapReduce [23] — not only in shared-nothing clusters, but also within the confines of a single shared-memory machine. In both these environments, the proposed solutions to tackle performance imbalances are similar: faster processors "steal" tasks from slower ones, albeit their associated limitations manifest at different architecture levels (e.g. data movement overheads shift from network and disk I/O to slow memory inter-connects and access conflicts in cache memory).

Tarazu [2] and its extension, Pikachu [33], are two recent MapReduce schedulers that aim to balance the load between "slow" and "fast" processors in a cluster. This loose distinction between processors is realized as a ratio of the rate at which each processor type consumes data chunks (termed *progress rate*; see LATE [67]). This ratio is estimated at run-time, after a job phase starts on both processor types. It is then used to partition (reshuffle) the input and intermediate data accordingly. These systems are agnostic of any differences in the micro-architecture between slow and fast processors and operate solely on two performance indicators: their progress rate and CPU utilization. *Heterogeneous MapReduce* (HetMR), our proposed MapReduce framework for heterogeneous hardware (see Section 5.3) is complementary

to this line of work in two ways: it pinpoints the loose terms "slow" and "fast" on specific hardware architectures (e.g., a CPU or an *field-programmable gate array* (FPGA)) on a per job basis; and, by doing so, it makes data movement overheads, those incurred during reshuffle, explicit.

Mate-CG [40] works under the same assumption as Tarazu and Pikachu — that faster nodes should process a larger fraction of the data than slower nodes — but it considers a cluster of *general-purpose graphics processing unit* (GPGPU)-enhanced nodes. The system further assumes that accelerators are throughput-oriented processors. As such, they should operate on large chunks of data to be effective. Therefore, their data partitioning policy is biased towards accelerators. Mate-CG deals only with iterative MapReduce jobs so that it can reshuffle data chunks at every iteration. Furthermore, it assumes that GPGPU implementations do not suffer from performance penalties inherent to that architecture (e.g. code branching or memory access conflicts).

A recent study suggests that many, if not all, MapReduce jobs fit well within the confines of a single shared-memory machine: there is enough thread parallelism and memory to support today's workloads [9]. Earlier on, a series of MapReduce systems (led by Phoenix [55, 61, 65]) studied how intermediate key/value pairs should be laid out in cache memory in symmetric multi-core systems. In parallel, Mars [36], MapCG [38], MapCG-shared [18, 19] have also attempted to wall in MapReduce within a GPGPU. The problem with those GPGPU-only solutions is that the higher they move in the memory hierarchy, the more severe the performance penalties become, mainly due to memory access conflicts and synchronization between shared and main memory. Our proposed alternative is a hybrid execution model.

Closely related to our notion of hybrid execution is MapCG-shared [19]. It proposes a map-dividing scheme (data partitioning) and a pipelining scheme when different phases run on different architectures. The latter is essentially the streaming model that *compute unified device architecture* (CUDA) proposes. However, their model is confined only to jobs with combiners — ignoring jobs without combiners. They also do not decide when to use either a hybrid or CPU solution. One of the main feature of our hybrid approach is that it allows for different hardware architecture designs to be combined within MapReduce.

FPMR [57] is a MapReduce framework specifically designed for FPGAs. It does not consider a hybrid solution but demonstrates that jobs can actually fit in the FPGA. One of the main features of FPMR is a common data path to share read-only job parameters between pipelines. This is indeed complementary to our work in some cases, because shared state has to be replicated and wired independently to each pipeline. There is also the case, however, that one of the streams can be dedicated to input parameters, as we do with the logistic regression application.

Beyond MapReduce models. GPGPU- or FPGA-based frameworks have attempted to maintain the simplicity of the MapReduce programming model, but they have not made explicit the performance penalty due to memory management. HetMR provides mechanisms to move data between the CPU and its accelerators and, for some cases, move data within an accelerator.

Dandelion [56], LINQits [21], Accelerator [58], Liquid Metal [11], and PetaBricks [6, 52] promise increased performance and reduced energy consumption in mainstream heterogeneous computing. Part of this promise can be attributed to code generation and compiler optimization techniques that deal with the intricacies of different hardware architectures internally. This is orthogonal to our line of work, where HetMR for FPGAs it imposes structure on how to handle key/value pairs both on- and off-chip.

SHEPARD and an the OmpSs framework. Deliverable D3.2 presents the SHEPARD and the OmpSs framework. While this work shares similar goals to ours — *when* and *how* to use an accelerator — there are two main differences. First, these frameworks propose scheduling mechanisms for batch jobs, while we are concerned with dataflow applications whose tasks may be scheduled or pipelined. The proposed OmpSs extension attempts to pinpoint relations between tasks, but they do not consider how these tasks are connected, and under what execution model. Second, the proposed scheduling strategies are based on execution time, while we are interested in non-clairvoyant scheduling strategies where the execution time is not known in advance. Furthermore, our work complements the data-driven scheduling models proposed for both SHEPARD and OmpSs.

3 Integrated system design

This chapter presents the general design of our performance models. We first detail our assumptions regarding the cloud infrastructure and the cloud applications, then we discuss a system model for automatically building performance models and the way to integrate blackbox and whitebox performance modeling.

3.1 Cloud model

Most current cloud offerings consist of a limited choice of predefined resource configurations. For example, Amazon EC2 currently proposes 23 standard configurations to choose from. In HARNESS we decided to design a cloud infrastructure capable of dynamically creating virtual resources based on a fine-grained description of its properties such as *number of cores*, *memory*, *core frequency* etc. Such fine-grained configuration specifications are already supported for example by the *open cloud computing interface* (OCCI) standard [50]. Note that this model creates considerably larger configuration search spaces than traditional clouds which offer a handful of fixed instance types. It is therefore trivial to adapt this work to EC2-like clouds, by simply disabling resource configurations which do not match one of the predefined instance types.

We also assume that cloud resources have a financial cost, and that the cloud can dynamically generate a cost value for any resource type that can be requested. Our system makes no assumption about the form that this cost model takes, and can therefore adjust to the specificities of any cloud's pricing model. In our experiments, however, we assume a cloud infrastructure which charges resources on a per-minute basis according to a linear model:

$$Cost(R) = R_1 \times Cost_{unit_{R1}} + R_2 \times Cost_{unit_{R2}} + \dots$$

As an example, for a VM with the configuration $\{N$ Cores, M GB of Memory $\}$, the cost per time unit may be calculated as:

$$Cost(VM) = N \times Cost_{1core} + M \times Cost_{1GB}$$

where $Cost_{1core}$ may be defined based on the frequency, amount of cache, etc.

3.2 Application model

Cloud applications may be extremely different from each other. They may require different sets of libraries, parameters, input/output files, etc. This is also true for the selection of resources they may execute on: some applications expect specific type of hardware, or constrain the number of machines they use (e.g., some applications may require the number of machines to be a power of two).

To allow a generic AM to handle arbitrary applications we rely on two specification file. The *Application Manifest* describes the application's structure and constraints about the resource types it

```
ApplicationName: HelloWorld

Parameters {
   Parameter_1 (
      Name:    Parameter_1
      Type:    Integer
      Values:  {v_1,v_2,...,v_n}
      Default: v_1
   ), ...
}

Resources {
  Resource_1 (
    Type:     Virtual Machine
    Number:   1
    Configuration: {
      Cores:  {1..16}
      Memory: {2,4,6,8,12,16,24,48,64,96,124}
    }
    Role:     Master
  ), ...
}

GlobalConstraints {...}
DeploymentActions {...}
ExecutionScript {...}
```

Figure 3.1: Application Manifest Example

needs. It is typically written by the application developer. The *service-level objective* (SLO) describes a user's expectations about acceptable execution times or costs, thus, typically written by the application user.

We already presented these specification files in detail in Deliverable D6.1 [30] so we discuss them very briefly here.

3.2.1 Application manifests

A manifest file is a specification of an application's structure and the type of resources it needs to execute correctly. As shown in Figure 3.1, it describes the types of resources supported by the application, the application input parameters, the constraints between resources and input parameters, and deployment actions on each resource type and execution script of the application.

```
ManifestUrl:          http://www.cloud.org/HelloWorldManifest
ExecutionParameters: {v1, ...}
Objective:            Cost <= 100
Optimization:         ExecutionTime
```

Figure 3.2: Service-Level Objective Example

Input parameters

The main step to manage applications is to identify performance-critical input parameters. This has to be done by an application expert (typically the application's developer). Performance-critical input parameters are parameters that influence execution time.

Each parameter is modeled as a *(type, values, default value)* tuple. The *ExecutionScript* part of the manifest describes how these parameters are passed to the application.

Resources

The manifest describes the types of resources an application needs, their number, configuration and role. An application manifest may specify either a fixed value, or a set of values to choose from. The *Configuration* describes the properties that resources may have. A computing resource may specify a number of cores and memory size, while a storage resource may describe properties such as the disk size. In addition, we can set a *Role* to a resource to describe applications with different components potentially having specific requirements. For example, a master/slave application may separately describe *Master* and *Slave* resources.

It is important to notice that, due to infrastructure limitations, not all combinations of resource parameters can be provisioned. The way to handle the issue with this asymmetric space is presented later.

3.2.2 Service-level objectives

A *service-level objective* (SLO) file describes a user's request for executing an application. As shown in Figure 3.2, it contains a reference to the application's Manifest, the list of parameters that should be passed to the application, and the user's expectations in terms of execution time and cost.

User's expectations can take two different forms. In Figure 3.2 the user imposes a maximum execution cost, with a secondary goal to execute the application as fast as possible. Alternatively one could rather impose a maximum execution time, while trying to spend as little as possible under this constraint.

3.3 System model

In the HARNESS system architecture, performance modeling is realized by the AM [32]: one AM instance is dynamically created to handle the execution of each new application. The internal architecture of the AM is depicted in Figure 3.3. A user triggers an execution of the application by submitting an SLO and a manifest file to the AM. The AM is a generic element which does not need any application-specific information except the application manifest and the SLO. It is in charge of choosing and provisioning

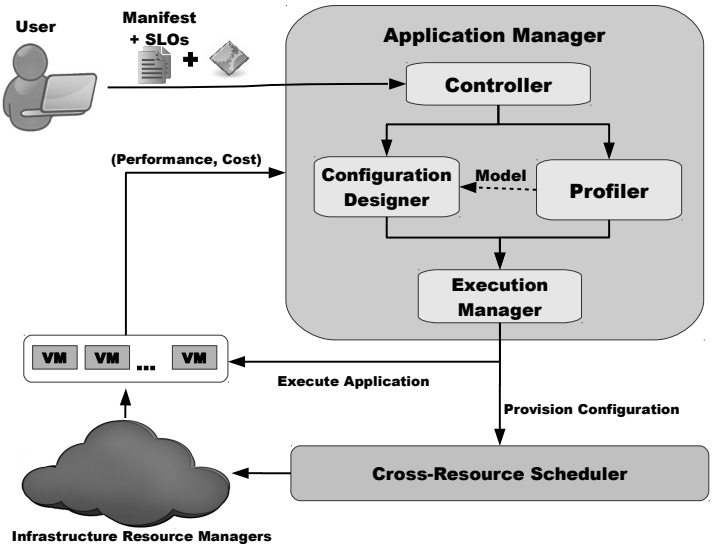


Figure 3.3: System overview

resources, deploying the application onto these resources, launching the execution of the application, and measuring the execution time and implied costs.

Initially, the AM has no knowledge about the types of resources it should choose for a newly submitted application. After loading the manifest and SLO files, in case no performance model is specified, the *Controller* forwards the application to the *Profiler* which repeadedly executes the application a number of times using various resource configurations. This profiling process runs until either a predefined number of iterations (executions) has been performed, or a profiling budget has been spent.

The result of these executions is used to build a performance model which is sent to the *Configuration Designer*. If a performance model is initially specified in the manifest, the *Controller* skips the *Profiler* and sends the application and its model directly to the *Configuration Designer*. Based on this model, the *Configuration Designer* then selects a configuration that satisfies the SLO.

The execution is handled by the *Execution Manager* which provisions the configuration through a *Cross-Resource Scheduler* (CRS) and finally executes the application.

Each time an application is run, the system monitors its total execution time and cost, and learns the relation between the choices made for this execution and the observed result:

$$(ExecTime, Cost) = Run_{app}(Parameters, Resources)$$

The results generated after several executions with various resource configurations can be plotted as shown in Figure 4.1 (p. 14). In this figure, each point represents the execution time and cost that are incurred by one particular resource configuration. For simplicity, the figure shows the result of an exhaustive exploration of a search space with 176 possible configurations. In a more challenging scenario the number of configurations would be much greater, and the profiling would have to stop long before completing the exhaustive exploration.

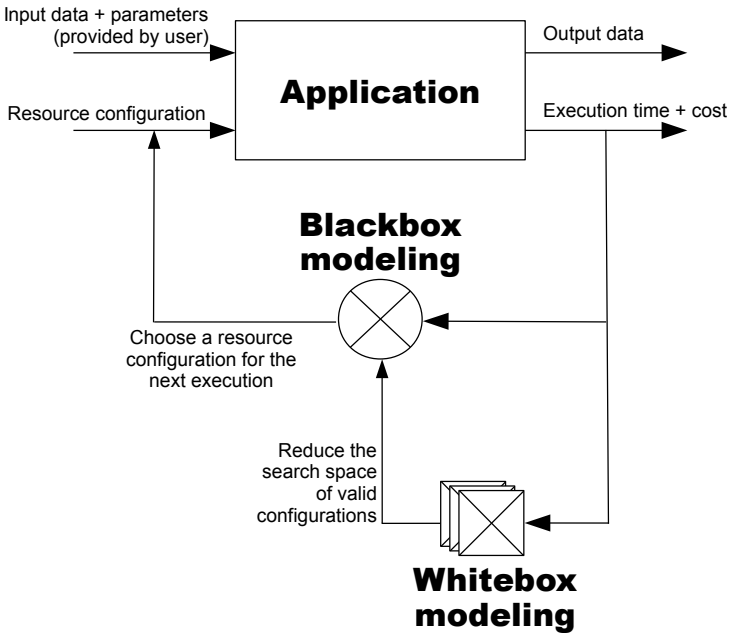


Figure 3.4: Integrated blackbox/whitebox performance model

3.4 Integrated blackbox and whitebox performance models

As previously discussed, WP6 investigates two complementary approaches to model the performance of cloud applications.

Blackbox modeling makes no assumption about the nature of the application, the way it is implemented, or the parallel frameworks it relies on. Instead it tries to discover the relationship between the resources that are given to an application and the execution performance. Blackbox modeling is very generic and can in principle be applied to any application. On the other hand it may require many executions with different resource configurations before generating a sufficiently complete performance model of the application.

Whitebox modeling makes use of specific knowledge about the application type. For example one can build a whitebox model specialized for MapReduce applications, or for Web applications. The scope of these models is limited to a certain range of applications, but on the other hand they may use domain-specific information to speed up the construction of a useful performance model.

We believe that combining blackbox and whitebox performance modeling may allow cloud users to benefit from the best of both worlds: a combined blackbox/whitebox model would be able to handle any type of application; but for a number of common application types it would make use of domain-specific information.

The idea for combining a generic blackbox model with one or more whitebox models is depicted in Figure 3.4. When an AM is created for a new application, it instantiates both types of models. Whenever an execution is triggered, the blackbox model chooses the resource configuration that should be assigned

to this execution. The resulting execution performance and cost are fed in the blackbox and whitebox models. Whitebox models can also collect additional domain-specific metrics such as MapReduce performance counters.

In the worst case, the whitebox performance models are unable to provide useful information about the application. The blackbox model therefore uses its own techniques (described in Chapter 4) to model application performance. However, if one of the whitebox models does generate useful information, it can inform the blackbox model about configurations that should be avoided. For example, the MapReduce performance model presented in Chapter 5 can efficiently decide if the use of GPGPUs or FPGAs accelerators is likely to provide a performance improvement compared to pure-CPU configurations. It can thus reduce the search space explored by the blackbox model and thereby steer its search towards useful configurations.

As of September 2014, the work on performance modeling has made good progress in both types of performance models taken in isolation, as reported in the next chapters. Integrating them in a single framework, as described here, remains to be realized during the last year of the project.

4 Blackbox performance modeling

Blackbox performance modeling relies on observing the performance and costs that can be obtained using specific resource configurations, and uses generic optimization algorithms to identify configurations that are likely to deliver good performance/cost trade-offs. This chapter describes how we represent the resource search space and the search methods implemented for blackbox profiling in order to select good resource configurations to extract a performance model.

4.1 Performance Profiling

The main issue when building the performance model of an application is that the space of all possible configurations is usually much too large to allow an exhaustive exploration. We therefore need to carefully choose which configurations should be tested, such that we identify the optimal configurations as quickly as possible.

4.1.1 Search Space

The search space of resource configurations to explore for an application is generated using the application manifest. Each resource parameter which should be chosen by the platform constitutes one dimension of the space. The number of possible configurations therefore increases exponentially as new dimensions are added, an issue often referred to as the curse of dimensionality.

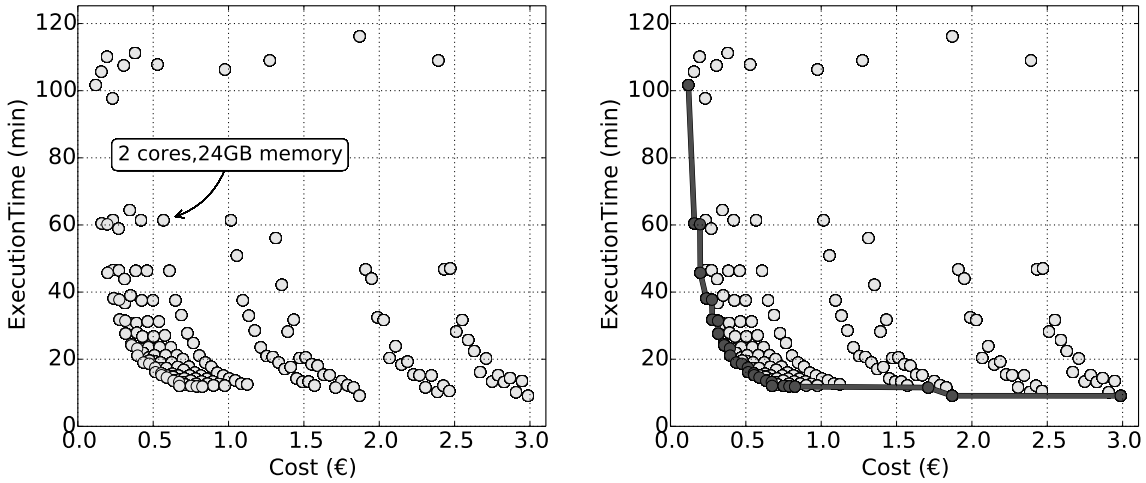
In the example from Figure 3.1, the search space of the application has 2 dimensions (corresponding to numbers of cores and memory). This creates a total of $16 \times 11 = 176$ possible configurations (due to 16 possible numbers of cores, and 11 possible memory sizes). Within these 176 configurations, only a subset of them may offer interesting trade-offs between performance and cost.

4.1.2 Pareto Frontier

It is interesting to notice that not all configurations provide interesting properties. Regardless of the application, a user is always interested in minimizing the execution time, the financial cost, or in finding a sweet spot between these two[1].

Figure 4.1(a) presents the search space of the single-node implementation of the *reverse time migration* (RTM) application. Configurations which appear at the top-right of the figure are both slow and expensive. Such configurations can be discarded as soon as we discover another configuration which is both faster and cheaper. The remaining configurations form the *Pareto frontier* of the explored search space. Figure 4.1(b) highlights the set of Pareto-optimal points of this application: they all implement interesting trade-offs between performance and cost: points on the top-left represent inexpensive-but-slow configurations, while points on the bottom-right represent fast-but-expensive configurations.

[1] An interesting extension of this work would be to consider additional evaluation metrics such as carbon footprint. This can be easily done as long as the relevant metrics are designed such that a lower value indicates a better evaluation.



(a) Exhaustive exploration of the resource configuration space.

(b) The set of Pareto-optimal configurations is shown in black.

Figure 4.1: Resource configuration space of the RTM application.

The Pareto frontier (and the set of configurations leading to these points) forms the performance model that the *Application Manager* (AM) uses to choose configurations satisfying the user's SLOs. If an SLO imposes a maximum execution time, the system discards the Pareto configurations which are too slow, and selects the cheapest remaining one. Conversely, if the SLO imposes a maximum cost, it discards the Pareto configurations which are too expensive, and selects the fastest remaining one.

4.1.3 Mapping Discrete Parameters

As shown in the manifest presented in the Figure 3.1 (p. 8), resource parameters take values from a discrete set. This ensures an efficient exploration of the search space. For some resource properties, assigning values that are very close from each other (e.g., by exploring available memory in steps of 1 MB) would provide no benefit as the performance would be largely the same.

While describing resource parameters using discrete values works well in many cases, exploring the search space defined by "truly discrete" dimensions is more difficult. For example, one dimension may represent the choice of a particular CPU architecture among a list (Intel-32, AMD-64, etc.). To tackle this problem and to be able to optimize the search process, we map the set of discrete values to a continuous interval $[0, 1]$. This allows applying agnostically different algorithms for function minimization, as long as the optimization algorithm does not depend on the search space being continuous.

The resource space may also be asymmetric. This means that not all the configurations resulting from the manifest can be provisioned. To solve this issue, we artifically assign a very large execution time and cost to these configurations in order to lower the probability of searching further in their neighborhood.

Algorithm 1 Uniform Search

Input: Application A, Resources $R = \{R_1, R_2, ..., R_n\}$
Output: Set of configurations, their execution time and cost $S_{r,t,c}$

```
1:  S_{r,t,c} ← ∅
2:  for r_1 = min_1 to max_1 by step_1 do
3:      for r_2 = min_2 to max_2 by step_2 do
4:          ...
5:          for r_n = min_n to max_n by step_n do
6:              r ← {r_1, r_2, ..., r_n}
7:              (t, c) ← execution time and cost of running A on r
8:              S_{r,t,c} ← S_{r,t,c} ∪ {(r, t, c)}
9:          end for
10:         ...
11:     end for
12: end for
```

4.1.4 Search Strategies

The goal of the profiling process is to search through the space of possible configurations and to quickly identify configurations that implement interesting performance/cost trade-offs, without having to explore every configuration from the search space. Notably, this search process does not only aim to find the fastest nor the cheapest configuration, but it also aims to identify as many configurations as possible which offer interesting trade-offs between these two extremes.

We define three strategies that can be used to explore a configuration space:

Uniform Search explores stepwise points in the resource search space to select a configuration for the profiling process. As shown in Algorithm 1, the application is executed for all combinations of stepwise resource values (lines 2-5). Although uniform search is extremely simple, it may waste time exploring large areas which are unlikely to deliver interesting performance/cost trade-offs. In addition, low exploration step values result in high complexity, while using high step values (to decrease the complexity) may skip relevant configurations.

Utilization-Driven is a simplified version of the CopperEgg strategy [22]. It iteratively refines an initial resource configuration by monitoring the resource utilization generated by the application. As shown in Algorithm 2, the algorithm starts with a random resource configuration (lines 1-2), and monitors the utilization of each resource type in configurations (lines 8-9). If a resource is highly used by the application, the algorithm then allocates a higher amount of this resource in the hope of delivering better performance (lines 10-11). On the other hand, if a resource utilization is low, the algorithm then reduces this resource amount in the hope of reducing resource costs (lines 12-13). Otherwise, it stops its exploration once there is no configuration that neither overuses nor underuses its resources. This strategy is simple and intuitive but, as we shall see later, it may stop prematurely whenever it reaches a local minimum in the search space.

Simulated Annealing is a generic algorithm for global optimization problems [64]. It initially tries a wide variety of configurations, then gradually focuses its search around configurations that were already found to be interesting. To control how many bad configurations are accepted as interesting,

Algorithm 2 Utilization-Driven

Input: Application A, Resources $R = \{R_1, R_2, ..., R_n\}$
Output: Set of configurations, their execution time and cost $S_{r,t,c}$

$r \leftarrow \{r_1, r_2, ..., r_n\}$ where r_i is random value of resource $R_i \in R$
$Q \leftarrow \{r\}$
$S_{r,t,c} \leftarrow \varnothing$
while $Q \neq \varnothing$ **do**
    $r \leftarrow$ dequeue(Q)
    $(t, c) \leftarrow$ execution time and cost of running A with resource configuration r
    $S_{r,t,c} \leftarrow S_{r,t,c} \cup \{(r, t, c)\}$
    for i = 1 to $|R|$ **do**
        if R_i is over- **or** underutilized **then**
            if R_i is overutilized **then**
                $r_i' \leftarrow$ next value of R_i (value after r_i)
            else if R_i is underutilized **then**
                $r_i' \leftarrow$ previous value of R_i (value before r_i)
            end if
            $enqueue(Q, \{r_1, r_2, ..., r_i', ..., r_n\})$
        end if
    end for
end while

Algorithm 3 Simulated Annealing

Input: Application A, Resources R, Temperatures $T_{cooling}$ and $T_{current}$
Output: Set of configurations, their execution time and cost $S_{r,t,c}$

$r \leftarrow \{r_1, r_2, ..., r_n\}$, r_i is random value of resource $R_i \in R$
$(t, c) \leftarrow$ execution time and cost of running A with resource configuration r
$S_{r,t,c} \leftarrow \{(r, t, c)\}$
while $T_{current} > T_{cooling}$ **do**
    $r_{new} \leftarrow neighbor(r, T_{current})$
    $(t_{new}, c_{new}) \leftarrow$ execution time and cost of running A with resource configuration r_{new}
    $S_{r,t,c} \leftarrow S_{r,t,c} \cup \{(r_{new}, t_{new}, c_{new})\}$
    if $Probability_{Acceptance}((t, c), (t_{new}, c_{new}), T_{current}) > random()$ **then**
        $r, t, c \leftarrow r_{new}, t_{new}, c_{new}$
    end if
    decrease $T_{current}$
end while

$neighbor(r, T_{current})$

$\Delta \leftarrow \varnothing$
for each r_i in r **do**
    $\Delta_i \leftarrow min(sqrt(T_{current}) * r_i,$
$(upper(r_i) - lower(r_i))/(3 * rate_{learn}))$
end for
return $r + random(0, 1) * \Delta * rate_{learn}$

it relies on a global time-varying parameter called the temperature. The algorithm starts with a random resource configuration (line 1), and explores new configuration in the neighborhood of the current configuration (line 5). The *neighbor* function determines a new configuration where $rate_{learn}$ is a scale constant for adjusting updates and $upper(r)$ and $lower(r)$ are the parameter r's interval bounds. While slow cooling (line 11), the algorithm accepts new configurations to explore with slowly decreasing probability (lines 8-10). Due to its convergence to optimal solution in a fixed amount of time, Simulated Annealing quickly explores the search space, focusing most of its efforts in the "interesting" parts of the search space.

The criterion used by the Simulated Annealing algorithm to decide on the utility of a configuration is the product between the cost and the execution time it generates:

$$utility = ExecTime \times Cost$$

Using this utility function the algorithm will explore the entire Pareto frontier, instead of focusing on optimizing only the execution time or the cost.

4.2 Evaluation

This section evaluates the search strategies along three relevant criteria: (i) the convergence speed of different search strategies towards identifying the full set of Pareto-optimal configurations; (ii) the quality of configurations we can derive from these results when facing various SLO requirements; and (iii) the costs and delays imposed by offline vs. online profiling.

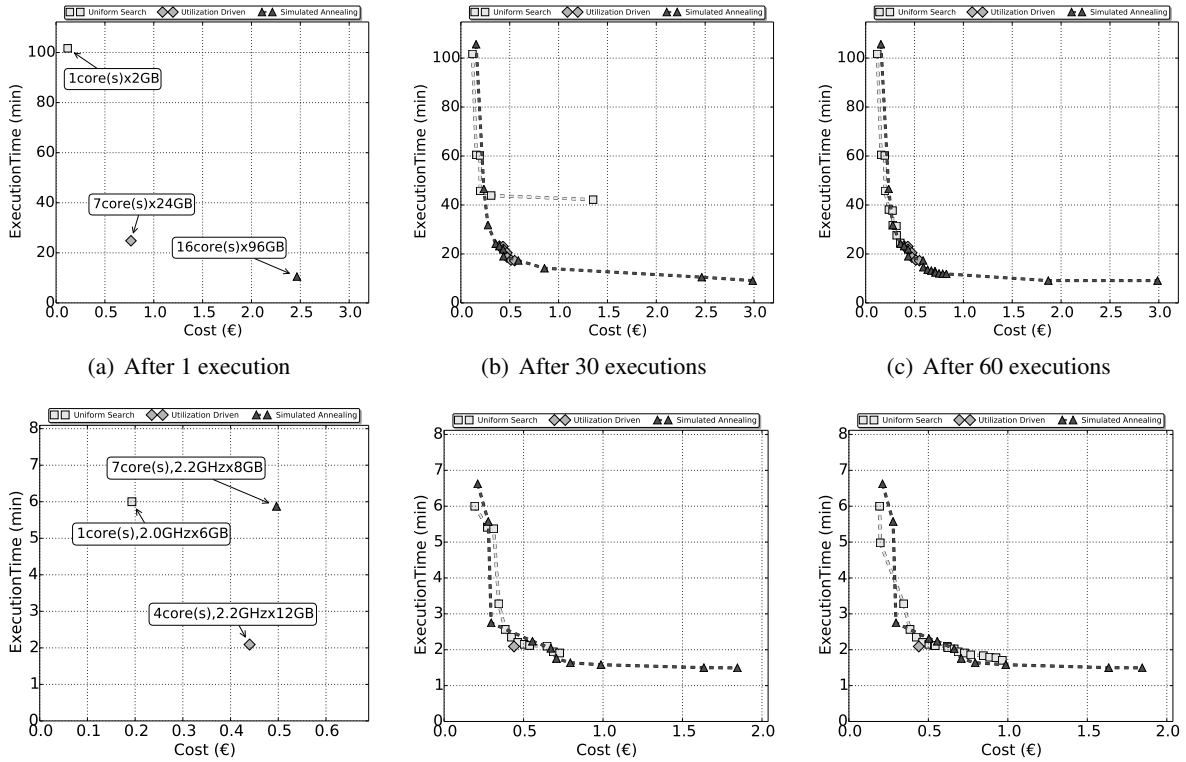
We base our evaluations on single-node, multithreaded implementations of two of the HARNESS use-case applications: RTM and *delta merge* (DM).

Both application manifests define resource configurations between 1 and 16 CPU cores and 11 discrete values between 2 and 124 GB of memory. We simplify the RTM case by setting a static CPU frequency of 2.2 GHz while for DM we assign 4 possible frequency values. This creates a relatively small search space with 172 configurations for RTM and a much larger one for DM. The main interest of the small size for RTM is to allow us to issue an exhaustive search through every possible configuration. Figure 4.1 shows the result of this exhaustive evaluation for RTM.

We perform all experiments using resources from the Grid'5000 experimentation testbed [34]. For RTM, we use machines from the "paranoia" cluster equipped with 2 Intel CPUs with 10 cores each running at 2.2 GHz, 128 GB of RAM and a 10 Gb Ethernet connectivity. Additional machines with different CPU frequency, number of cores and amount of memory are used for executing the DM application. This heterogeneity determines an asymmetric search space.

All machines run a 64-bit Debian Squeeze 6.0 operating system (OS) with the Linux-2.6.32-5-amd64 kernel. We use QEMU/KVM version 0.12.5 as the hypervisor. We deploy the OpenNebula cloud infrastructure in these machines so our AM can request any VM configuration via the OCCI interface. We repeated all experiments three times, and kept the average values for execution time.

Although we can use Grid'5000 at no cost, we defined a simple cost model to emulate the situation of a commercial cloud. Our applications typically run within tens of minutes so we based our pricing scheme on a one-minute pricing granularity. Longer-running applications would probably use a more classical one-hour granularity. Similarly, our model charges the user for each resource separately. The



(a) After 1 execution (b) After 30 executions (c) After 60 executions

(d) After 1 execution (e) After 30 executions (f) After 60 executions

Figure 4.2: Pareto frontiers for RTM (a,b,c) and DM (d,e,f). Utilization-driven stops after 19 executions of RTM and 1 execution of DM. Simulated Annealing stops after 52 executions for RTM and 42 executions for DM.

parameters of this model are derived from a linear regression over the price of cloud resources at Amazon EC2:

$$Cost_{min} = 0.0396 * N_{Cores} + 0.0186 * N_{Memory(GB)} + 0.0417$$

When using cores of different frequency, their cost is scaled accordingly.

Note that our system does not rely on this particular cost model. It is general enough to accept any other function capable of giving a cost for any VM configuration.

4.2.1 Convergence speed

To understand which search strategy identifies efficient configurations faster, we compute the Pareto frontiers after 30 and 60 executions. This helps observing which strategy tends to get closer to the real Pareto frontier of RTM while for DM we can only assume it as we did not explore it exhaustively in advance. The results are presented in Figure 4.2.

In the case of Uniform Search, we use a step equal to the unit for each dimension of the search space. It therefore actually completes an exhaustive search of the configuration space. We can observe that this

SLO / Strategy	C < 0.50 €	C < 1.50 €	C <2.50 €
Uniform Search	T = 43.86 min	T = 42.17 min	T = 42.17 min
Utilization-driven	T = 18.68 min	T = 17.25 min	T = 17.25 min
Simulated Annealing	T = 19.03 min	T = 14.16 min	T = 10.52 min

Table 4.1: Performance after 30 executions of RTM under cost (C) constraints.

SLO / Strategy	T < 20 min	T < 60 min	T< 100 min
Uniform Search	*Fail*	C = 0.20 €	C = 0.16 €
Utilization-driven	C = 0.47 €	C = 0.39 €	C = 0.39 €
Simulated Annealing	C = 0.43 €	C = 0.23 €	C = 0.23 €

Table 4.2: Performance after 30 executions of RTM under time (T) constraints.

strategy converges very slowly. It eventually finds the full Pareto frontier, but only after it completes its exhaustive space exploration.

The Utilization-Driven strategy starts from a randomly generated configuration in the search space. This randomly-chosen starting point creates a different search path for each run of this strategy. In the worst case, this strategy starts with a configuration which neither over- nor underutilizes its resources, so the search stops after a single run. In the best case, the algorithm starts from a configuration already very close to the Pareto frontier, in which case it actually identifies a number of good configurations. We show here an average case (neither the best nor the worst we have observed): it quickly identifies a few interesting configurations but then stops prematurely so it does not identify the entire frontier.

Finally, Simulated Annealing also starts from a randomly generated configuration. We can however observe that it converges faster than the others towards the actual Pareto frontier. For both applications, after just 30 iterations it has already identified most of the interesting configurations.

For both applications, Simulated Annealing seems to be the optimal search strategy in most of the cases. For redundancy reasons, we base further evaluations on RTM only.

4.2.2 SLO Satisfaction Ratio

Another important aspect of the search result is the range of SLO requirements it can fulfill, and the quality of the configurations that will be chosen by the platform under such SLOs. We froze the performance models generated by each strategy after 30 executions, and checked how these respective performance models would perform.

Table 4.1 presents the execution times that would be observed if the SLO imposed various values of maximum cost. Conversely, Table 4.2 shows the costs that would be obtained after defining a maximum execution time.

It is clear from both tables that Simulated Annealing provides better configurations. With its good approximation of the entire Pareto frontier, it can handle all SLOs from the table. The other strategies have only a partial frontier and cannot find configurations for demanding SLOs. At the same time, when

Strategy	Total cost	Duration
Uniform Search	20.80 €	2194.02 min
Utilization-driven	6.71 €	858.88 min
Simulated Annealing	38.42 €	812.07 min

Table 4.3: Total Cost and duration overhead for an offline profiling of RTM limited to 30 executions.

several strategies can propose solutions that match the SLO constraint, the solutions found by Simulated Annealing are almost always better.

4.2.3 Profiling Costs

Another important aspect is the time and cost incurred by the profiling process. As we can observe from the previous evaluation, to reach to the Pareto-optimal configurations, a number of experiments must be performed. These experiments incur an additional cost and duration which can be minimized based on user's choice on profiling approach:

1. The *offline approach* creates artificial executions of the application whose only purpose is to generate a performance model. In this case, the output of executions is simply discarded.

2. The *online approach* opportunistically uses the first real executions requested by the user to try various resource configurations and lazily build a performance model.

Table 4.3 presents the cost and duration overhead of offline profiling using 30 experiments. Offline profiling using the Utilization-driven strategy appears to be cheap and fast. However, this is only due to the fact that this strategy stops long before having identified the full Pareto frontier. Similarly, Uniform Search ends up being cheaper than Simulated Annealing because it starts its exploration from the cheapest available resource types.

Figure 4.3 shows the execution times and costs incurred by the user using the Simulated Annealing strategy in conjunction with online profiling. In this case, no artificial execution is generated. On the other hand, as we can see in the figure, many executions violate an arbitrary SLO of 1.5 €. However, it is interesting to notice that the overall group of execution remains within its aggregated budget (with a negative cost overhead of -9.11 €). Similarly, when applying an arbitrary SLO of 30 minutes of execution time, numerous individual executions violate the SLO but overall the execution time overhead is again negative (-242.66 minutes).

We conclude that the search based on Simulated Annealing shows the fastest convergence to optimal configurations and provides a better satisfaction for the SLOs. It generates good configurations to be used when creating an application profile in a smaller number of executions.

For users willing to tolerate SLO violations on individual executions, the online profiling strategy provides obvious benefits: it remains within the aggregate time or budget of the overall profiling phase, and therefore offers fast and cost-effective generation of a full performance model. On the other hand users unwilling or unable to tolerate individual SLO violations can revert to the offline strategy, at the expense of artificial executions which consume both time and money.

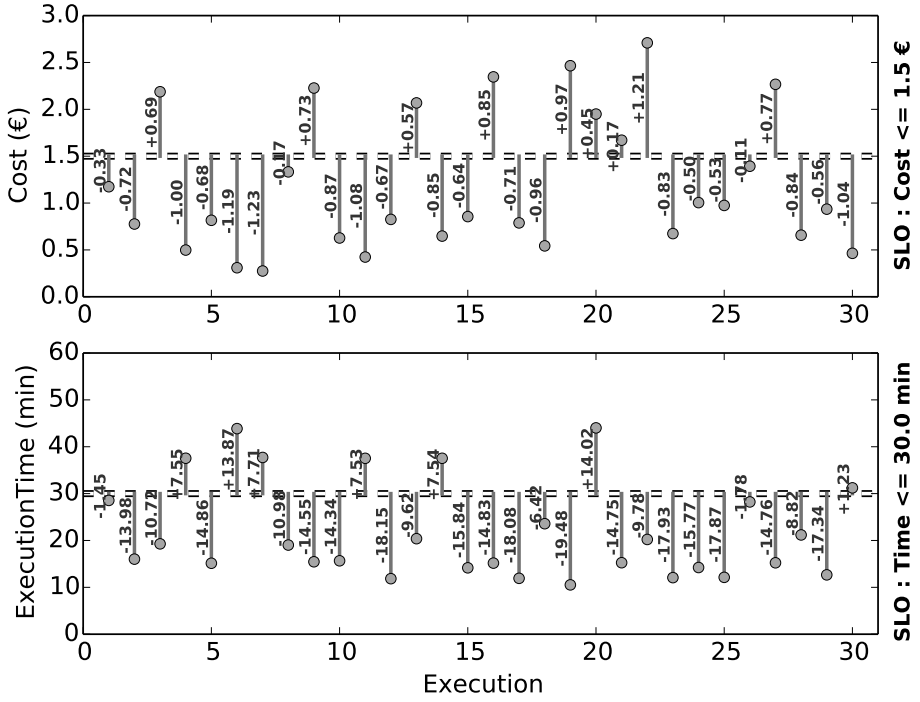


Figure 4.3: Cost and Execution Time fluctuation in an online profiling of RTM limited to 30 executions.

4.3 Current status

Following the experimental results, we integrated the Simulated Annealing-based profiling in the AM of the HARNESS platform. For simplicity, the AM currently implements the offline profiling policy. In the future, we plan to replace it by the online profiling or even to give the choice between the two policies so users get extra flexibility.

5 Whitebox performance modeling for heterogeneous data parallel processing systems

This chapter explores the design space of data parallel processing frameworks suitable for deployment on the HARNESS platform. Frameworks such as MapReduce [23], Storm [8], Spark [68] and Naiad [48] can scale out computation to a large number of nodes in a data center — and, to some degree, scale up in shared-memory multi-core machines. However, these frameworks do not only require developers to adopt a particular programming model (e.g., functional, declarative, or dataflow), they also employ custom execution models to parallelize modern applications. For example, such restricted models have recently gained wide-spread popularity among data scientists who want to use ever more stateful implementations of online machine learning algorithms (e.g., collaborative filtering, belief-propagation networks, and logistic regression) and execute them over large datasets while providing "fresh", low latency results (see Figure 5.1). HARNESS aims to explore the high-level programming abstractions and execution models of prominent data parallel processing frameworks and facilitate the mapping of their applications to heterogeneous resources. We often term this exploration as *whitebox performance modeling* because we study the effect of data representation, data movement, and phase transitions within a framework on the performance of a hybrid execution model for heterogeneous hardware (e.g., a model where an application executes on both a multi-core CPU **and** an FPGA accelerator). We argue that, by carefully crafting trade-offs between data transformation and movement and data parallelism, it is possible to increase the

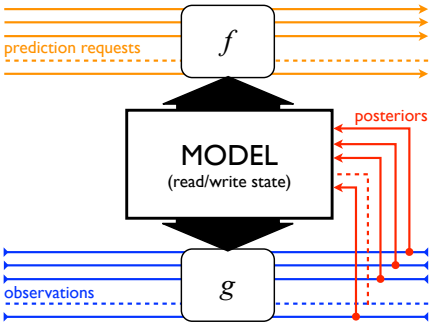


Figure 5.1: AdPredictor, one of our use cases, is a Bayesian belief-propagation system [28]. Its *model* holds prior beliefs for a set of parameters that are passed, along with new observations, to a training function g yielding a set of posterior beliefs. Training is an iterative process that requires a high data processing throughput. At the same time, the model is available to a prediction function f. Prediction requires low-latency responses, because ads are included in the search results. The more "fresh" is the model, the higher is the accuracy of predictions.

efficiency of mapping decisions of individual application components to heterogeneous computational resources.

This chapter makes the following contributions:

- We classify data-parallel processing frameworks according to their computational, programming, and execution models. We use this early design space exploration (Task T6.4) to motivate our *whitebox performance modeling* approach, asking what hints should HARNESS provide to a given framework when deciding to map part of an application's dataflow to a heterogeneous processor (e.g., a GPGPU, or an FPGA). We further demonstrate this challenge in practice with a simple application example (Section 5.1).

- Using MapReduce as an example framework, we argue that the performance improvement attained by accelerators, if any, is dependent on both the *data* and the *phase* (map or reduce) of a MapReduce job (Section 5.2). Based on a generalized model of shared-memory MapReduce, we describe **HetMR**, a MapReduce framework that can execute jobs in a hybrid fashion across CPUs and accelerators. HetMR first profiles the MapReduce job on the CPU, yielding a measure of potential speedup. Depending on the outcome, it then deploys the job in either a CPU-only or hybrid fashion by drawing on a library of designs. Our experimental evaluation shows that HetMR's hybrid execution model improves job completion by up to 6 times with an FPGA (Section 5.3).

- Finally, we provide a quick overview of our current work on multi-core stream processing systems. This work aims to apply lessons learned with HetMR to more complex dataflows [16, 17], with an emphasis on uniform programming and execution models for heterogeneous hardware (Section 5.4).

5.1 Design space

Table 5.1 classifies data parallel processing frameworks according to their computational, programming, and execution model [17]. For the latter, we further showcase support for low latency and iterative computations — such are the requirements of modern machine learning applications like AdPredictor.

5.1.1 Scheduling dataflows

Tasks in a dataflow graph can be *scheduled* for execution or materialized in a *pipeline*, each with different performance implications. In MapReduce, for example, map and reduce tasks are scheduled across a set of cluster nodes [23]; in Storm, tasks are pipelined. Some frameworks such as Spark follow a *hybrid* approach in which tasks on the same node are pipelined but not between nodes [68].

Since tasks in **stateless dataflows** and **incremental dataflows** are scheduled to process coarse-grained batches of data, such systems can exploit the full parallelism of a cluster but they cannot achieve low processing latency. For lower latency, **batched dataflows** divide data into small batches for processing and use efficient, yet complex, task schedulers to resolve data dependencies. These schedulers have a fundamental trade-off between the lower latency of smaller batches and the higher throughput of larger ones — typically they burden developers with making this trade-off [69]. **Continuous dataflows** adopt a streaming model with a pipeline of tasks. They do not materialize intermediate data between nodes and thus have lower latency without a scheduling overhead: batched dataflows cannot achieve the

Computational model	*Systems*	*Programming model*	*Execution model*	*Low latency*	*Iteration*
Stateless dataflow	MapReduce [23]	Map/reduce	Scheduled	×	×
	DryadLINQ [66]	Functional	Scheduled	×	✓
	Spark [68]	Functional	Hybrid	×	✓
	CIEL [49]	Imperative	Scheduled	×	✓
	Haloop [15]	Map/reduce	Scheduled	×	✓
Incremental dataflow	Incoop [13]	Map/reduce	Scheduled	×	×
	Nectar [35]	Functional	Scheduled	×	×
	CBP [45]	Dataflow	Scheduled	×	×
Batched dataflow	Comet [37]	Functional	Scheduled	✓	×
	D-streams [69]	Functional	Hybrid	✓	✓
	Naiad [48]	Dataflow	Hybrid	✓	✓
Continuous dataflow	Storm [8], S4 [7]	Dataflow	Pipelined	✓	×
	SEEP [16]	Dataflow	Pipelined	✓	×
Parallel, in-memory	Picollo [54]	Imperative	N/A	✓	✓
Stateful dataflow	SDG [17]	Imperative	Pipelined	✓	✓

Table 5.1: Design space of data parallel processing frameworks [17].

same low latencies. **Stateful dataflows**, like SDGs, are fully pipelined to support online processing with low latency [17].

To improve the performance of iterative computation in dataflows, early frameworks such as HaLoop cache the results of one iteration as input to the next [15]. Recent frameworks generalize this concept by permitting iteration over arbitrary parts of a dataflow graph, executing tasks repeatedly as part of loops [68, 49]. SDGs support iteration explicitly by permitting cycles in the dataflow graph.

5.1.2 Scheduling dataflows on heterogeneous hardware

Cloud providers offer access to hardware accelerators, such as FPGAs and GPGPUs, according to a pay-per-use model. Data-parallel processing frameworks make it easy for users to express data parallel applications, but an open challenge remains how such jobs can exploit accelerators in a cloud setting: users still lack decision support on when accelerators can deliver a benefit for their applications.

We describe an experiment in Storm: (i) to show that scheduling continuous dataflows on heterogeneous hardware can indeed increase the data processing throughput of CPU-intensive applications while achieving low latency; and also (ii) to motivate the challenge of helping the scheduler of one such framework to decide *when* and *how* to use an accelerator. As an example application, we use the Black-Scholes model, a financial model for estimating the future price of stock options.

Experimental design and setup. In Storm, a dataflow graph consists of *spouts* (stream generators) and *bolts* (stream processors) executed in a pipeline fashion by *executors* — one per spout or bolt — running on *worker* servers. Our experimental setup consists of two workers, $\mathtt{w}_1$ and $\mathtt{w}_2$, connected via a 1 Gbit network switch. Worker $\mathtt{w}_1$ hosts two spouts that send tuples to $\mathtt{w}_2$. Each tuple consisting of a timestamp and a 1 kB data buffer. One or more bolts on worker $\mathtt{w}_2$ process these tuples and emit a new tuple of

the same size, carrying the original timestamp forward. The new tuples are sent back to worker w_1 to compute the latency per tuple.

We run three kinds of bolts on w_2:

- *No-op* bolts that simply forward tuples without performing any computation;
- *Black-Scholes* bolts that compute future stock prices per input on the CPU in parallel; and
- a single *Black-Scholes acc'ed* bolt that compute future stock prices on the GPGPU.

Worker w_2 has an Intel i7 3.07GHz quad-core hyper-threaded CPU and an Nvidia Tesla C2070 GPGPU connected to the CPU via a *peripheral component interconnect express* (PCIe) gen.2 $\times 16$ adapter. For programming the GPGPU version of Black-Scholes we use AMD's Aparapi library [5]. This *application programming interface* (API) allows developers to write annotated, data-parallel Java functions. Aparapi then translates the resulting Java byte code to *Open Computing Language* (OpenCL), compiles it, and runs it on the GPGPU, handling automatically the data movement between the two processors based on the provided annotations.

I/O and CPU bounds. The two spouts of w_1, when sending tuples *as fast as they can*, are enough to saturate Storm's processing bandwidth on w_2. Storm can handle up to $\sim 60,000$ tuples/s due to the serialization and deserialization overhead involved when receiving data over the network. The No-op line in Figure 5.2(a) shows this peek throughput achieved with the No-op bolts. With the Black-Scholes bolts, however, the processing throughput drops. The Black-Scholes application is data parallel, so we expect it to scale linearly until it saturates the 8 hyper-threaded cores of w_2. It does so, but its peek throughput is still below Storm's line rate. Figure 5.2(c) shows the tuple end-to-end latency when running the No-op and Black-Scholes bolts. For the latter we observe that, although the average latency of Black-Scholes drops as we increase data parallelism, it has high variance. The 95th percentile results show this adverse effect due to CPU contention and contention on shared data structures (input and output message queues) among executors.

Batching tuples for the GPGPU. Figures 5.2(b) and 5.2(d) show the processing throughput and end-to-end latency achieved by the Black-Scholes acc'ed bolt, respectively. For this bolt, we batch n consecutive tuples before we send them to the GPGPU for processing. Note that when tuples arrives at a rate much lower than Storm's line rate (say, at 100 tuples/sec), batching will cause latency to increase linearly with n. With small batch sizes, however, the GPGPU processing throughput is dominated by the I/O overhead between the CPU and the GPGPU. The larger the batch size, the higher the Black-Scholes acc'ed processing throughput. Indeed, when sending tuples from w_1 to w_2 as fast as possible, our GPGPU-enhanced solution overcomes the CPU processing bottleneck, achieving a $\times 1.4$ speed-up; while at the same time it achieves low latency with very low variance.

Storm $vs.$ **SEEP.** The GPGPU-enhanced Black-Scholes application is eventually bound by the serialization and deserialization overhead of Storm. The GPGPU, however, can process data at a higher rate. To demonstrate this, we repeat the **same** experiment in SEEP [16], a Java-based continuous dataflow engine developed at *Imperial College London* (IMP). Figure 5.3 compares the throughput of the Black-Scholes acc'ed application achieved by both systems. SEEP has overcome much of the (de)serialization bottlenecks of Storm and, thus, the GPGPU can process data at ~100 MB/s. Like in the experiment with Storm,

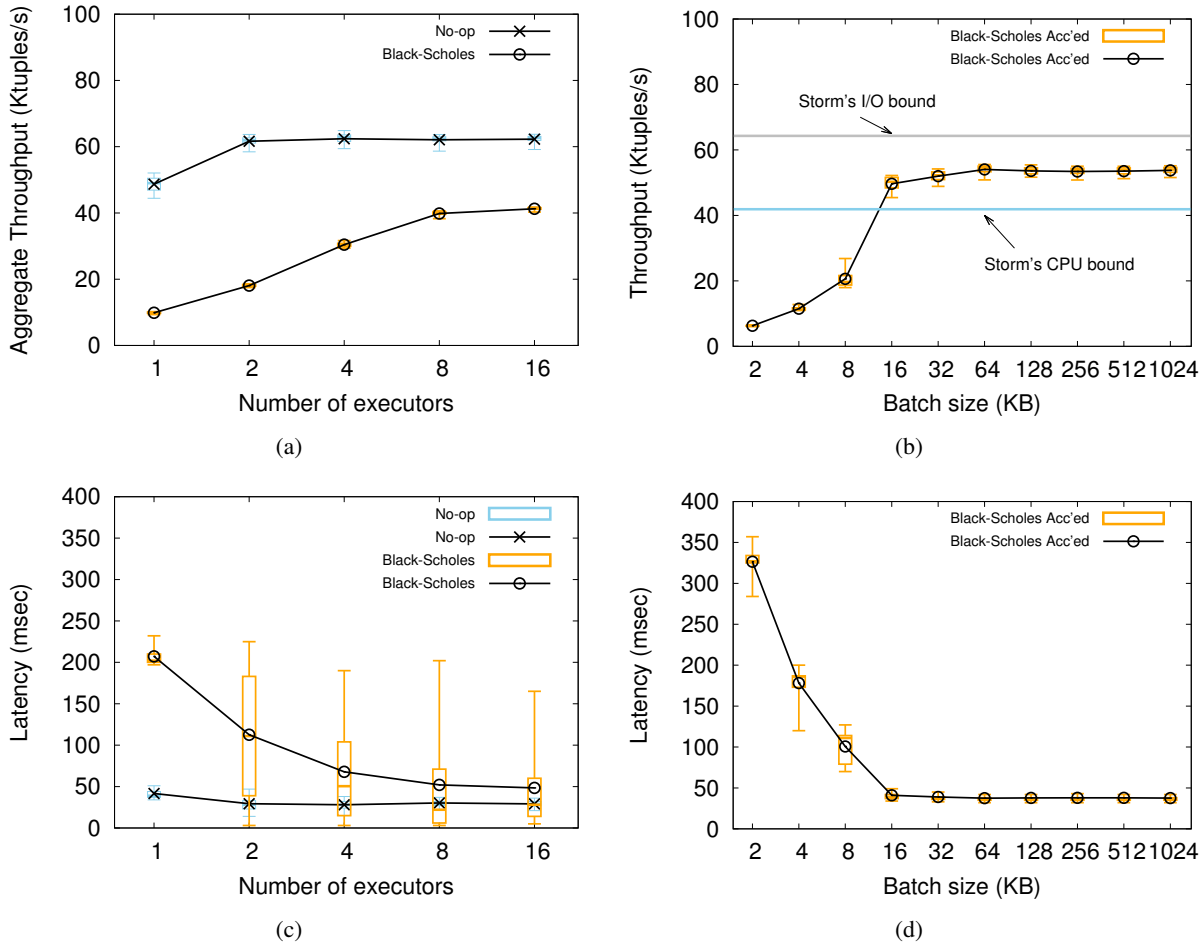


Figure 5.2: Storm performance with the *No-op*, *Black-Scholes*, and *Black-Scholes acc'ed* bolts. Bars represent the 5th, 25th, 50th, 75th, and 95th percentiles and points represent average values.

Black-Scholes acc'ed running on SEEP achieves low latency with low variance and overcomes the CPU bottleneck by a factor of up to 1.8×.

The challenge of *when* and *how* to use an accelerator. Storm's and SEEP's default schedulers assume homogeneous worker servers and assign the operators of a continuous dataflow to worker slots so that the load is balanced among their workers. For our experiments, we have pinned the Black-Scholes acc'ed operator on worker w_2 by extending these default schedulers to handle this particular placement constraint. However, such simple scheduling policies do not necessarily guarantee benefits[1]. Even in these simple experiments, the correct decision is not only dependent on just a hard placement constraint, but also on the input data rate, the data transformation (in our case, batching), as well as the data movement cost between the CPU and the GPGPU.

[1]For example, a scheduling policy of the form: "Given a dataflow graph $G = (V, E)$ and a set of GPGPU-enhanced workers Y, if a GPGPU version of bolt $v \in V$ exists, then schedule v on executor $y \in Y$."

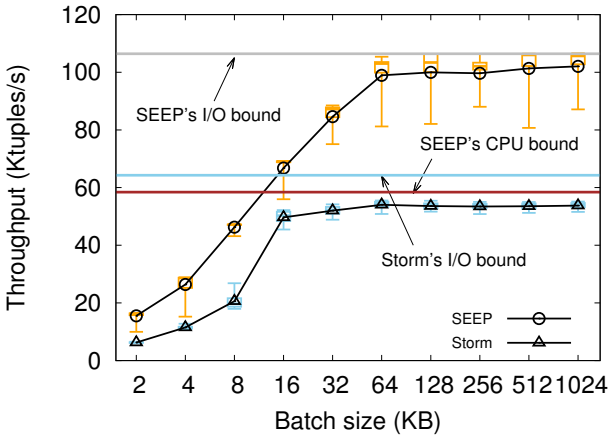


Figure 5.3: Performance of the Black-Scholes acc'ed operation in SEEP and Storm. Horizontal lines represent the peek throughput attained by a system with 16 workers running either the *No-op* (I/O bound) or the *Black-Scholes* operator (CPU bound).

All in all, whitebox performance modeling aims to hint developers and blackbox profilers (see Chapter 4) alike on the effect of such variables to the end-to-end performance of an application. We believe that by integrating whitebox and blackbox performance modeling we will be able to devise a richer API that (i) developers can use to provide hints to the HARNESS platform on *when* and *how* to use an accelerator for a given dataflow computational model (Task D6.4); and (ii) profiling tools can use to guide their exploration of the cost/performance search space.

5.2 Scaling up shared-memory MapReduce

We now focus on MapReduce applications. The goal here is to understand the impact of the different compute platforms on the performance of MapReduce jobs. We are not particularly concerned with how best to program a MapReduce application on a CPU, GPGPU or FPGA, which is itself an important area of study. Rather, we want to understand how best to make a *deployment decision* assuming good implementations are available for the alternative platforms.

5.2.1 Background and motivation

Classical MapReduce is structured as a *map phase* and a *reduce phase*. In the map phase, input data are divided into multiple pieces, called *shards*, that are processed by *mappers*. A mapper usually consumes more than one shard and emits one or more *key/value pairs*. In the reduce phase the data are also processed in multiple pieces, but on a per-key basis. Between the map and reduce phases, the *shuffle phase* groups and sorts the intermediate data emitted by the mappers and then presents those data to the reducers. As an optimization, *combiners* can in some cases be attached to mappers to perform localized data reduction, in particular when the reduction function is associative and commutative. This serves to reduce the volume of intermediate data that must be moved during the shuffle phase.

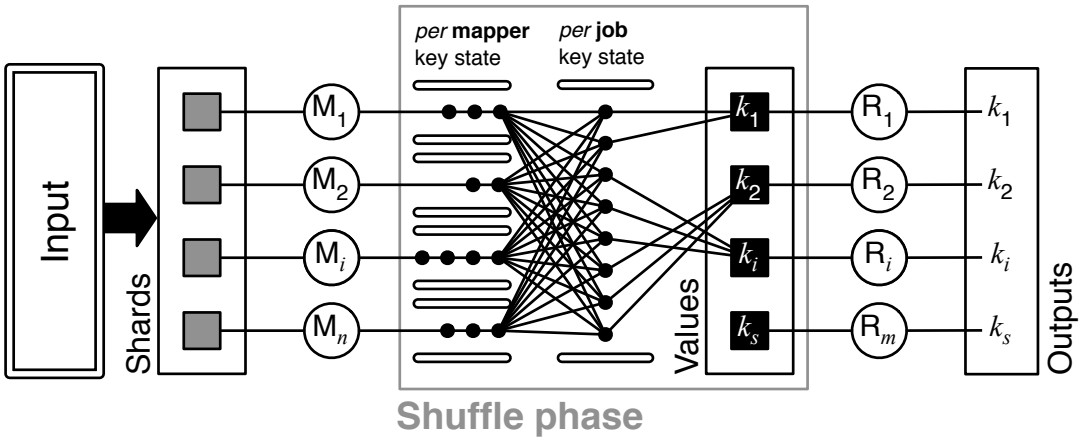


Figure 5.4: Shared-memory MapReduce model. *Local* state (*per* **mapper** key state) is replicated n times and populated in private by one of the $\mathsf{M}_1 \ldots \mathsf{M}_n$ mappers of the system to avoid lock contention. Each mapper maintains a key-indexed dictionary and either *a)* accumulates, by means of a combiner, or *b)* buffers, by means of chaining, intermediate values in isolation. When there are no more input shards to process, mappers copy their private state into *shared* state (*per* **job** key state). On the reduce side, reducers $\mathsf{R}_1 \ldots \mathsf{R}_m$ pull all values associated with a particular key, $k_1 \ldots k_s$, and copy them to a list. During this second data movement, it is also possible to accumulate values with combiners.

We make two critical observations. First, assuming the input data reside on the host CPU *dynamic random-access memory* (DRAM), the effective throughput of an accelerator tasked with running a MapReduce job is limited by the speed of the *peripheral component interconnect* (PCI) express (PCIe) bus that connects it to the host CPU. (The situation is similar for Infiniband or Ethernet connections to an accelerator.) For example, data are streamed to the NVIDIA GPGPU via a 6 GB/s PCIe bus and to the Virtex FPGA via a 2 GB/s PCIe bus. Therefore, the speed of the PCIe bus does not only dictate the computational throughput of the accelerator, but also the potential speedup attained over a CPU.

Second, the throughput of a MapReduce application can be bottlenecked by either the map or the reduce computation. Therefore, one phase might turn out to be a significantly better candidate for acceleration than the other. Both data movement and computational load turn out to be important factors in a deployment decision.

Shared-memory MapReduce. In classical MapReduce, input data are first moved to the mappers, then intermediate data moved between the mappers and the reducers, and finally the result data moved back toward the client. These data movements in a typical implementation of MapReduce involve expensive file system reads and writes. A significant performance improvement can be attained by instead taking advantage of large main memories to hold the intermediate data of the shuffle phase. In fact, the shuffle phase becomes nearly trivial in such a *shared-memory MapReduce*, where a hash table can be used to store intermediate results by key, thereby avoiding the need to sort the data (Figure 5.4). Of course, this assumes that the intermediate data fit within the available memory. Again, combiners can be used to help reduce those data.

Phoenix++ [61], which targets symmetric, multi-threaded CPUs, is the latest in a series of shared-memory MapReduce systems [55, 65]. Compared to its predecessors, Phoenix++ improves mainly on

cache memory locality during the shuffle phase. The CPU-based Phoenix++ system has been closely tracked by corresponding implementations on GPGPUs. MapCG-shared is one of the latest systems attempting to place a general MapReduce framework inside a GPGPU [18]. In line with Phoenix++, this system advocates the use of combiners and dedicates a large part of shared memory to their use. Although it yields better performance in comparison to its GPGPU predecessors [36, 38], it has poor performance in comparison to CPU-based Phoenix++. For example, the Histogram application is 150× slower on MapCG-shared than on Phoenix++. The reason is simple: although the GPGPU provides fast access to cache memory, the threads compete for that access, negatively impacting data parallelism.

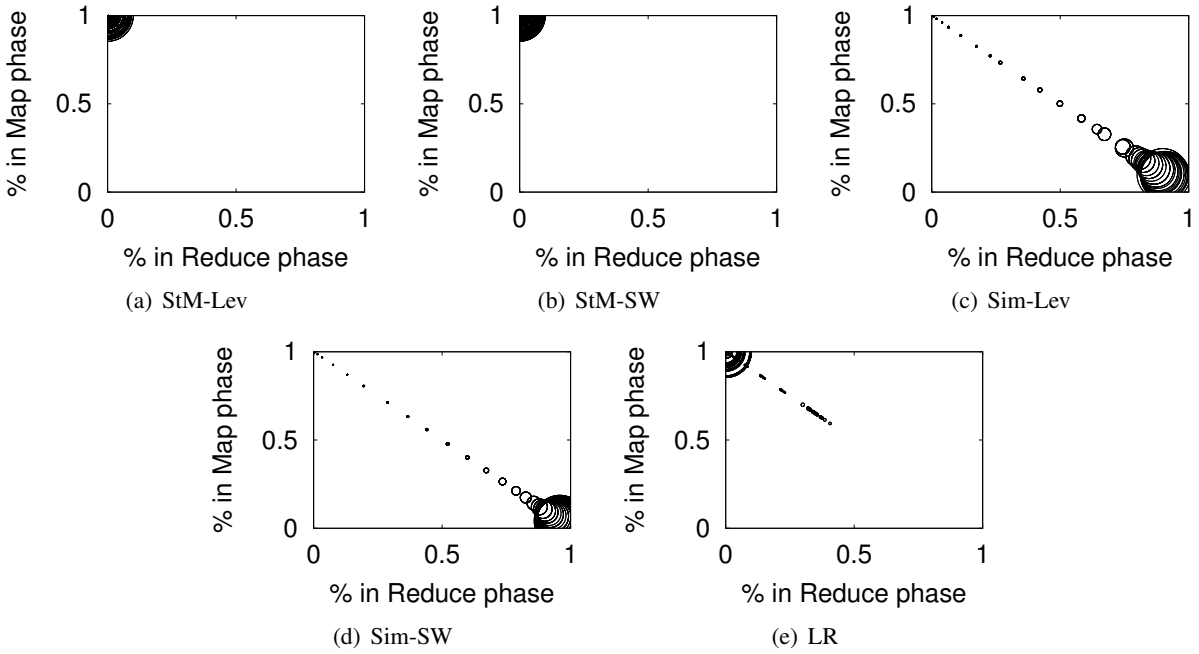
Example applications. Our work on MapReduce is informed and evaluated by looking at several example applications and a range of jobs (datasets) for those applications. The different datasets are created by varying some important parameter controlling the nature of the dataset. Abstractly, the applications fall into two classes: machine learning and correlation. For machine learning we use *logistic regression* (LR), while for correlation we use *string match* (StM) and *similarity* (Sim). The correlation applications are further specialized into whether they use the *Levenshtein* (-Lev) or the *Smith-Waterman* (-SW) distance algorithms. This yields five concrete applications.

The datasets for the two classes of applications are quite different. For machine learning, the input is N d-dimensional points, where we vary d from 2^2 to 2^{26}. In order to keep the datasets of the same size we also vary N from 2^{26} to 2^2. Thus, the product of $N \times d$ is consistently 2^{28}. A point size of 4 bytes then results in a fixed input size of 1 GB. The dataset for correlation is drawn from a Wikipedia repository. In the case of StM, the specific application is to correlate the Wikipedia article titles with a given set of words, while in the case of Sim, it is to find a correlation in the article titles written by each author (i.e., a distance between the titles). The data-dependent variable for both StM and Sim is the distribution of article title lengths (binned into 10 percentiles). In StM we make the simplifying assumption that the application programmer "knows" the set of words that will need to be matched so that we can restrict the data-dependent variable to the Wikipedia articles themselves.

5.2.2 Bottlenecks in MapReduce

The execution time of any given MapReduce job is usually dominated by a particular phase, which thereby acts as a throughput bottleneck. A common assumption is that the bottleneck phase is consistently the map. However, this is not the case. In particular, the presence or absence of combiners — essentially a design decision made by the programmer of a MapReduce application — turns out to strongly determine the bottleneck phase. Consider that when a job uses combiners, all key values have already been reduced by the second stage of data movement, thus essentially rendering the reduce function to a noop. In contrast, the absence of combiners leaves intact lists of intermediate values per key and, therefore, requires the reducers to expend greater computational effort.

More specifically, the presence of combiners shifts the computational cost of jobs to mappers relative to reducers because the number of partitions (data parallelism cardinality) of jobs with combiners equals the number of input chunks. In other words, the parallelism strategy is established *before* the map phase: the input data set is partitioned into chunks, chunks are processed in parallel and then the results of each computation are aggregated. A large number of machine learning applications follow this pattern, including k-means clustering, linear and logistic regression, neural networks and principal component analysis [20]. The same property holds for arithmetic applications such as fast multiplication [60].



(a) StM-Lev (b) StM-SW (c) Sim-Lev

(d) Sim-SW (e) LR

Figure 5.5: Bifurcation of Map/Reduce applications: map-intensive or reduce-intensive.

On the other hand, the absence of combiners shifts the computational cost of jobs to reducers relative to mappers, because the data parallelism cardinality of jobs without combiners is determined by partitioning intermediate key/value pairs based on some relation of the input data. In other words, the parallelism strategy is established *after* the map phase. The role of mappers is simply to ensure that intermediate key/value pairs are grouped accordingly. For example, graph algorithms are parallelized, in the map phase, by splitting the input graph into subgraphs to be processed in the reduce phase [41, 59]. Similarly, in database joins, mappers group together pairs of values for reducers [1]. Such reduce functions typically have computations that are exceedingly difficult to parallelize.

To summarize, we recognize two broad classes: *map-intensive* applications, consisting of a computationally expensive map function, an associative and commutative combiner function, and a trivial (noop) reducer; and *reduce-intensive* applications, consisting of a computationally inexpensive map function whose main purpose is to balance the load among reducers (usually by means of some universal hash function), a buffer combiner, and a computational expensive reducer (typically polynomial in time complexity).

Figure 5.5 illustrates the bottleneck effect by showing the relative amount of time spent in the map and reduce phases for each of the five example MapReduce applications. Each data point represents a job. The size of the point is related to the total amount of time. We can see that StM in both its variants is highly map-intensive, since their mappers were designed to include combiners that render the reducer irrelevant. Both Sim jobs, on the other hand, do not have combiners, which means more computation takes place in the reducers. Furthermore, we can see that as the jobs become more complex (i.e., their run times increase), they also become more reduce-intensive. Finally, LR is shown to be a map-intensive application.

5.2.3 Data movement cost

When deploying MapReduce jobs in a hybrid fashion across a CPU and an FPGA, it is important to account for the data movement costs between the CPU and the accelerator. Typically, this is limited by the maximum throughput supported by the PCIe bus, which interconnects the CPU memory with the FPGA on-board DRAM memory. The PCIe bus bottleneck means that a MapReduce job deployed in a hybrid fashion theoretically cannot achieve a throughput that is higher than 1.3 GB/s.

In practice, as we show as part of our experimental evaluation in Section 5.3.4, the maximum achievable data movement rate between the CPU and FPGA is lower than the PCIe throughput limit due to *data transformation cost*: when data is exchanged after the map phase ran on the CPU, the data sent over the PCIe bus must be transformed and laid out in a way that is compatible with the data ingestion pipelines of the FPGA reduce kernel (see Section 5.3.3). This transformation incurs a computational cost, and it may also increase the amount of intermediate data to be transferred over the PCIe bus.

5.2.4 Deployment procedure

Next we describe a deployment procedure that, by profiling the data movement requirements of a MapReduce job, can make a decision about the potential reduction in job completion time that a given MapReduce job may experience as part of a hybrid deployment.

The essence of the procedure involves two steps: *profiling* a sample and *selecting* an implementation based on the profile. We assume a scenario in which CPU-only and hybrid implementations of an application are placed into a library managed by a cloud data center platform. A customer comes to the data center with a particular job to be run, which amounts to presenting the platform with a dataset. The customer wishes to reach some Pareto-optimal point (discussed in Section 4.1) implementing a trade-off that maximizes end-to-end throughput and minimizes execution cost, so the hybrid implementation and its associated higher costs should be used only if there is an acceleration benefit to be gained.

Step 1: Profiling on a sample. We use an instrumented version of the CPU-only shared-memory implementation of the application to *predict* which implementation will provide the given job with a higher end-to-end throughput. The instrumentation measures the throughput attained by the CPU-only implementation at each data movement point: insertion into the map phase, extraction from the map phase, insertion into the reduce phase and extraction from the reduce phase. Critical to the utility of this prediction is that we can obtain sufficient information by running the profile on only a relatively *small sample* of the input dataset. The sample must, of course, be representative of the dataset. We make the assumption that the customer is able to provide such a sample. In our evaluation in Section 5.3.4 we use the trivial sampling technique of taking a small prefix of the input dataset.

Step 2: Selecting an implementation. The metrics collected in Step 1 will tell us whether and where the job would cause the PCIe throughput cap to be exceeded. The hybrid implementation will induce a particular need for the PCIe to move data into and out of the bottleneck phase. Taking the metrics together with the implementation requirements, we can determine if the hybrid implementation would respect the PCIe cap for that job. If it does not, then we choose the CPU-only implementation. Notice that if both map-intensive and reduce-intensive hybrid implementations are available, we are given more freedom in making a selection.

5.3 Heterogeneous MapReduce

In this section we describe the design of the HetMR system, which executes a MapReduce job across a CPU and an FPGA based on the outcome of the decision procedure (Section 5.3.1). We first give an overview of the architecture of the HetMR system (Section 5.3.2). After that, we describe some of the implementation details of map and reduce phases on an FPGA for different classes of jobs, which permit HetMR to achieve superior performance across a range of applications (Section 5.3.3).

5.3.1 Overview

The HetMR system implements a hybrid shared-memory MapReduce model that can execute (i) both the map and reduce phases on a multi-core CPU; (ii) execute the map phase on the CPU and the reduce phase on an FPGA; or (iii) execute the map phase on the FGPA and the reduce phase on the CPU.

As part of its architecture, HetMR must manage the execution of map and reduce phases and the data movement between them:

Execution environment. HetMR must offer an execution environment for map and reduce functions both on the CPU and the FPGA. While this is trivial to achieve for a CPU, which can execute functions implemented in a general-purpose programming language according to a shared-memory MapReduce model, it is challenging for an FPGA, which is not programmable in a general-purpose fashion. Instead, designs for FPGA kernel implementations for the map and reduce phases of different applications must be prepared ahead of job deployment time. Therefore, we adopt an approach in which HetMR has access to a *library* of FPGA designs for different MapReduce applications, which can be executed on demand. The FPGA designs focus on either the map or reduce phase for a given application based on the bottlenecks specific to that application (Section 5.2.2).

Data movement. Under hybrid execution of a MapReduce job, the HetMR system must manage the movement of data between the map and reduce phases. It therefore includes run-time functionality that moves data from the CPU memory to the FPGA memory, and back. It also manages the data movement from the FPGA memory to the FPGA on-chip memory, and back.

The challenge is to intercept the MapReduce data flow at the right stage of the computation in order to avoid stalling either the CPU or FPGA resources. In addition, the moved data must be laid out within a contiguous memory region and aligned in a way that enables data parallelism across multiple FPGA pipelines. When data is moved from the FPGA back to CPU memory, it should be laid out in such a way that the CPU can resume processing using its multiple threads.

5.3.2 Architecture

As shown in Figure 5.6, the HetMR design includes five main components: (i) the *deployment manager* makes a decision about how to execute a given job; (ii) the *CPU executor* runs map and reduce tasks on a multi-core CPU; (iii) the *FPGA executor* can initiate the execution of job-specific map or reduce kernels, which are selected from (iv) the *FPGA job library*; and (v) the *CPU-FPGA data mover* manages the movement of data between the CPU memory and the FPGA memory over a PCIe bus.

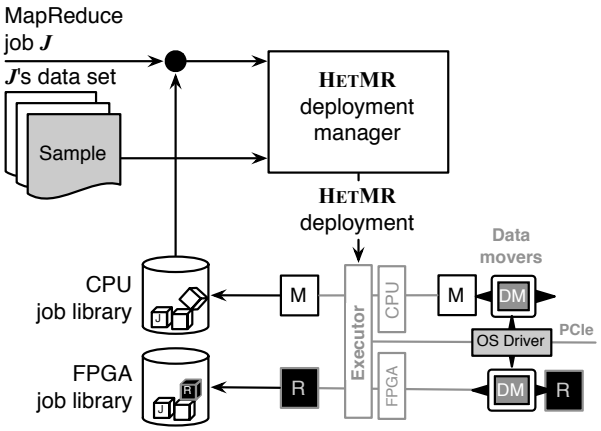


Figure 5.6: Overview of the HetMR architecture.

Deployment manager

The deployment manager implements the decision procedure (Section 5.2.4) to determine how a given job can benefit from FPGA acceleration by deciding to deploy a given phase on the FPGA. The particular phase deployed to the FPGA is determined by: (i) which phase exhibits the job's throughput bottleneck and (ii) whether the required data movement can be achieved within the throughput limit of the CPU/FPGA bus. The deployment manager receives a MapReduce job specification and, based on its decision, passes the individual map and reduce tasks to the CPU and FPGA executors for execution.

Note that the deployment manager is one of the focal integration points with the *Application Manager* (AM) for the joint blackbox/whitebox performance modelling approach (discussed in Chapter 3).

CPU executor

The CPU executor provides an execution environment for map and reduce tasks according to a shared-memory MapReduce model:

Threads. It manages a set of *execution threads*, which are permanently assigned to individual CPU cores. Each thread processes a queue of map or reduce tasks that is populated after the input data have been partitioned. Threads process either map or reduce tasks until their task queue becomes empty, and there are no tasks to steal from other threads.

Dataflow. Similar to previous shared-memory MapReduce frameworks such as Phoenix++, the CPU executor uses a two-stage *data movement* that groups intermediate key/value pairs in memory. There are two main data structures that hold intermediate key/value pairs: *local state* and *shared state*. Local state is replicated n times, each populated by one of the n threads that execute map tasks. Shared state is accessible by all threads. In the first stage of data movement, map tasks accumulate intermediate key/value pairs as part of their local state. After they have finished, they copy the data into a shared memory space. The executor maintains pre-allocated memory slots per key and per thread so that synchronization overheads between threads are minimized. In the second stage of data movement, threads execute reduce tasks that merge together the results of each map task according to the key.

Map. During the map phase, each thread executing a map task maintains a key-indexed dictionary and either (a) *accumulates* key/value pairs by means of a *combiner function* or (b) *buffers* key/value pairs by means of concatenating intermediate values. The first data movement above occurs when there are no more input partitions to process, at which point map tasks copy their private state into shared state.

Reduce. After all map tasks have finished, the reduce phase begins. Each reduce task collects all values associated with a given key, and copies them to a list — the input to the reduce function. During this data movement, it is also possible to accumulate values with combiner functions. The shared state is large enough to hold the output of all map tasks. Reduce tasks do not compete to access shared state because it is accessed per key.

FPGA executor

The FPGA executor is responsible to executing the map or reduce phase of an application on the FPGA. It obtains appropriate designs for job-specific map and reduce task implementations from the FPGA job library. Once it has received a given task design, it loads the design into the FPGA. It then invokes the CPU-FPGA data mover to initiate the transfer of data into the deployed FPGA kernel.

After data movement, the input data reside in FPGA memory. The FPGA executor sets the number of cycles execute on the FPGA based on the input size and the degree of parallelism on-chip. It also sets any static job-specific parameters, such as the number of dimensions in the input data, in registers or block RAM (BRAM). The executor interfaces with the FPGA kernel through two memory controllers for input and output, respectively. It initializes the memory controllers to the start address so that the deployed FPGA kernel can read or write data. Finally it signals the kernel to start synchronous data transfer, and it awaits an interrupt that signals that the last output byte has been written to memory by the FPGA kernel.

FPGA job library

The FGPA job library contains a set of application-specific FPGA kernel designs for map and reduce tasks that can be accelerated. Internally, each *computation kernel* is complemented by two *memory kernels* that handle the data movement from DRAM to the kernel and back. These memory kernels are the same across designs; different designs simply configure them with different parameters. A kernel can have more than one input and output stream. With typical FPGA hardware, there is a limited number of streams available between the memory and the computation kernel (e.g. up to 16 streams in total).

CPU-FPGA data mover

The CPU-FPGA data mover manages the data transfer from the CPU memory to the FPGA memory and vice versa. It handles the transcoding of data into correct data formats, the memory layout of the data, and initiates data transfers to ensure that FPGA kernels that execute map or reduce tasks have required input data.

The data mover supports data represented in a range of data structures, including lists, arrays and vectors, that can be used to represent input or intermediate key/value pairs as used by the CPU executor. In addition, it also manages the contiguous CPU memory region in which data can reside. When it is passed a pointer by the CPU or FPGA executors, it can transfer a specific number of bytes between

memory types. It assumes that the data in the FPGA memory have already been aligned and laid out appropriately by the FPGA executor.

5.3.3 FPGA implementation

Next we provide details on some of the FPGA implementation challenges when implementing map or reduce phases on the FPGA in order to interface them with CPU execution without a performance penalty. First, we describe our solution to provide fast read memory access through an optimized data layout, which leverages the knowledge of the application semantics to deal with potential memory conflicts arising during write operations. After that, we discuss our implementation of data-parallel combiner functions on the FGPA.

Memory access

The key challenge related to the FPGA kernel designs of map and reduce tasks is to ensure that the hardware pipelines are not stalled on memory accesses (both for reads and writes). Due to the low clock frequency of the FPGA, pipeline stalls drastically reduce performance, possibly outweighing the benefits of running a map or reduce phase on the FPGA in the first place.

Optimized data layout for memory reads. Depending on the nature of the phase being executed on the FPGA, data can be read either from the main CPU memory, through the PCIe bus, or directly from the on-board FPGA memory through the DRAM bus. Efficient memory access crucially determines performance. For example, many machine-learning jobs require multiple iterations during the map phase and so intermediate data are stored in the on-board memory; some applications, such as word count, only require a single pass and, hence, data are only read through the PCIe bus.

The PCIe bus offers a much lower throughput compared to the DRAM bus. On our hardware set up (Section 5.3.4), it exhibits a peak performance of 16 bytes per cycle, corresponding to 1.49 GB/s; in contrast, the DRAM bus has a peak throughput of 38.4 Gbps (i.e. 384 bytes/cycle). This difference in memory access throughput introduces a challenge: an unoptimized data layout can severely reduce the rate at which data can be read. For example, consider a reduce phase implementing an edit distance function (e.g. the Levenshtein or Smith-Waterman distance), in which two text strings are compared character-by-character. A natural way to implement this comparison would be to use a *systolic architecture* [44] such as the one depicted in Figure 5.7. However, this constrains the read throughput to only 2 bytes/cycle (i.e. 0.19 GB/s) because of the loop dependencies: the second character of a string is valuable to the computation only after the first has been processed.

Based on our experience implementing many MapReduce applications, we identify two main recurring patterns to optimize the data layout and overcome the above problem:

Circular memory access. Machine-learning applications in MapReduce are typically written in an iterative fashion where data are read from the on-board memory at the beginning of each iteration. To efficiently support this, we have built a memory controller optimized for circular memory access. This memory controller has an extra counter that resets the read address to the start address of the section of the data that has to be iterated.

Systolic array. To solve the issue of loop dependencies when dealing with a systolic array implementation, we utilize the following layout: rather than storing the strings sequentially, we rearrange them

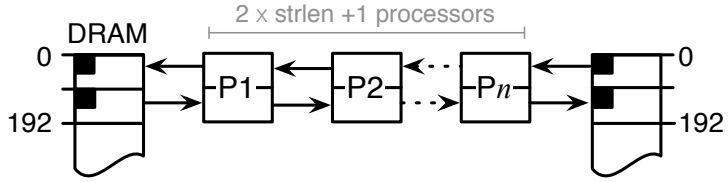


Figure 5.7: String distance calculator as a systolic array of processors P1 . . . Pn where n is a function of the string length (strlen). Each processor in the pipeline reads and writes 2 bytes per cycle, one from left and one from right. The pipeline can be replicated 96 times.

in memory so that the first 192 bytes correspond to the first character of 96 string pairs, the next 192 bytes to the second character, and so on. This allows the FPGA executor to feed 96 arrays in parallel with 2 bytes/cycle each, thus achieving the maximum throughput available.

Avoiding memory write conflicts. Another potential cause of low FPGA processing throughput is the occurrence of memory access conflicts between concurrent pipelines on the FPGA. This event typically predominates during the map phase when intermediate key/value pairs are written into an array structure, realized through registers or with on-chip BRAM. If data cannot be written immediately, the FPGA pipeline stalls and throughput decreases.

We implement two different approaches to address this issue, thus reducing the impact of memory conflicts. The first is *application-specific* and exploits the knowledge of the application semantics: if all FPGA pipelines generate the same key set — which would generate a large number of conflicts in a naive design — we extend the map kernel to also include a data-parallel combiner function that aggregates all keys. This removes the need to store each individual key/value pair in memory, thus avoiding the occurrence of write conflicts. In general, however, it is hard to predict memory access usage. In this case, we adopt an *application-agnostic* approach that dedicates a separate DRAM write stream to each pipeline. This approach, however, is limited by the hardware to 16 streams and excess use of resources for buffering between them.

Data-parallel combiners

When a given MapReduce application supports combiner functions, the job performance can be improved using a data-parallel design of combiner functions on the FPGA. In addition, such a class of applications is also more likely to result in a design that fits in its entirety on the FPGA.

In its simplest form, an FPGA pipeline for a combiner lags behind a map pipeline by only a few cycles, assuming that all keys fit in cache memory. Given a newly emitted key/value pair $(k2, v2)$, a combiner requires at least one cycle to read the previous value associated with key $k2$ and increment it with $v2$, and at least one cycle to write the accumulated value back to the same physical memory location. Figure 5.8 shows one such combiner implemented with a dual-port BRAM.

We assume that all keys can be read in parallel in the same cycle because they fit in cache memory. Similarly, in the next cycle, all values can also be updated in parallel. However, when multiple FPGA pipelines request access to the same physical memory address in the same cycle, we are required to multiplex both the address bus (Address A, Figure 5.8) and the data bus (Data B). If this multiplexing is

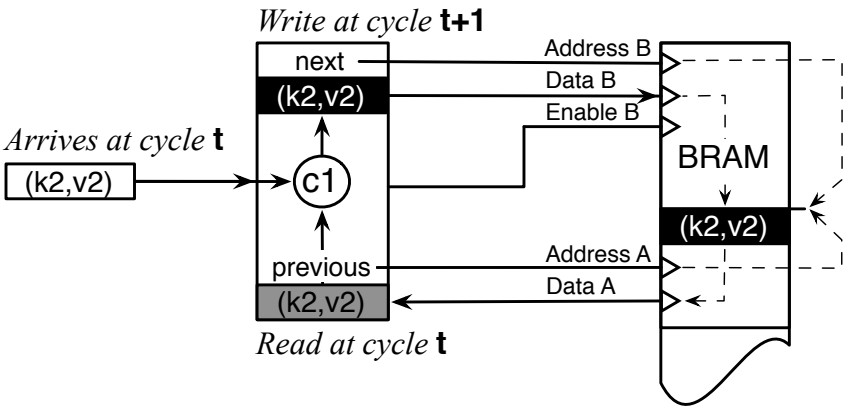


Figure 5.8: A $k2$ combiner object using a dual-port BRAM. Port A is `READ_ONLY` and port B is `WRITE_ONLY`.

known in advance (e.g. if all FPGA pipelines generate the same key set), it is straightforward to attach one such data-parallel combiner to the map kernel, thus fitting the entire job on-chip. Otherwise, we must deal with memory access conflicts by partitioning the key space of intermediate data.

5.3.4 Evaluation

Our evaluation consist of two parts. First, we want to understand the impact of the sample when profiling a job. Next, we want to verify that our deployment manager correctly selects the fastest deployment option. Our result show that when the sample is representative of the full data set, our deployment manager *always* makes the right decision. Further, they indicate that if the job execution time is dominated by the map, the sample size is largely irrelevant, while it becomes critical for reduce-intensive job. These result confirm the feasibility and effectiveness of our approach and, at the same time, it stresses the importance for the user to properly select the sample.

Experimental setup

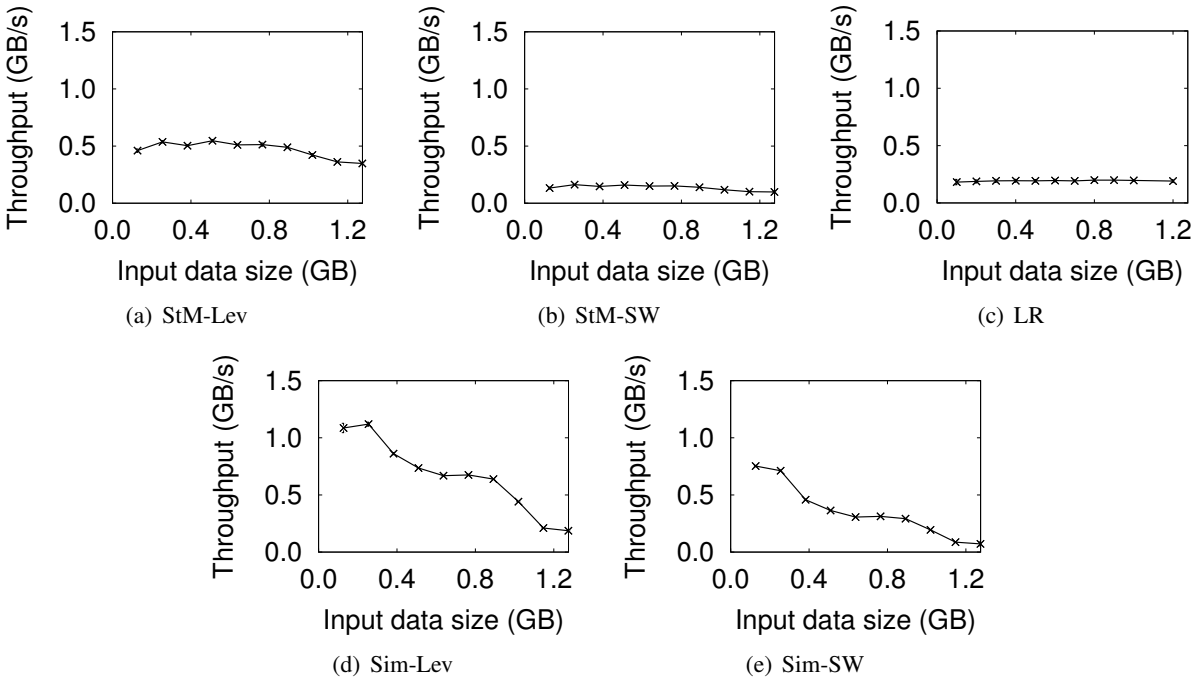
We performed all the CPU runs (including profiling) of our experiments using Phoenix++ on a Maxeler's 1U MPC-C series node [47]. This comprises two Intel Xeon 5650 chips, with six hyper-threaded cores and 12 MB L3 cache per chip; totaling 24 execution threads. The CPU is connected to a Xilinx Virtex-6 FPGA board via the PCIe bus.

In our experiments, we consider a set of representative MapReduce jobs, detailed in Table 5.2. Logistic Regression and String match represent map-intensive jobs; the Similarity application represents reduce-intensive jobs. Logistic regression is an ML job which is strongly structured in terms of both inputs and outputs, while String match and Similarity have only structured outputs. As input data for the latter two, we use a snapshot of 1.274 GB of Wikipedia articles. For logistic regression, instead, we randomly generate the numbers using a uniform distribution.

We run each experiment 10 times and we plot the average of the results. We do not show error bars in the chart as the standard deviation across runs was always below 5%.

Application	Abbrv.	Size (GB)
Logistic regression	LR	1
Similarity (Levenstein)	Sim-Lev	1.3
Similarity (Smith-Waterman)	Sim-SW	1.3
String match (Levenstein)	StM-Lev	1.3
String match (Smith-Waterman)	StM-SW	1.3

Table 5.2: Datasets



(a) StM-Lev (b) StM-SW (c) LR

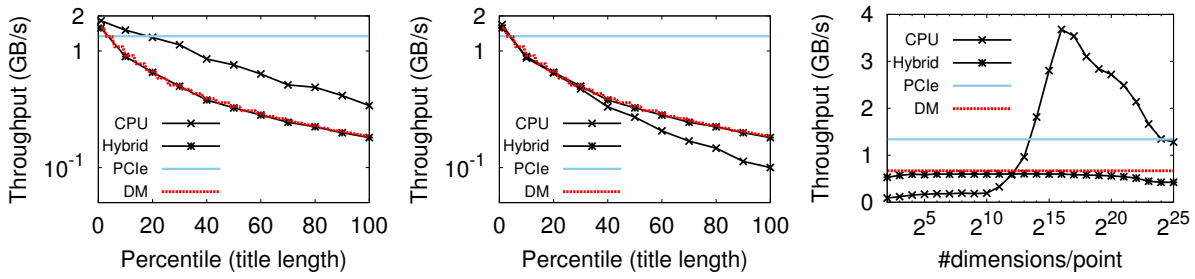
(d) Sim-Lev (e) Sim-SW

Figure 5.9: CPU throughput against sample data size for different MapReduce jobs

On stability and sample size

Figure 5.9 shows the impact of the sample data size on the CPU throughput. For jobs in Figure 5.9 (a)-5.9 (c), the throughput obtained is largely independent from the sample size. This is important because it means that small samples can be used, which drastically reduces the profiling time. Based on our results, a small sample size of 100 MB seems sufficient to obtain a good estimation of the expected CPU throughput when running the entire job. Even assuming a conservative throughput of 100 Mbps, this would lead to a profiling time shorter than 10 s. We consider this overhead negligible in practice.

Sim-Lev (Figure 5.9 (d)) and Sim-SW (Figure 5.9 (e)), instead, show a different behavior as throughput sharply decreases with larger sample size. The reason is that the first three jobs are *map-intensive* while these other two are *reduce-intensive*. For the first category, the size of the data to move to the FPGA is



(a) StM-Lev (b) StM-SW (c) LR

Figure 5.10: CPU throughput against FPGA throughput. The PCI line identifies the PCIe throughput while DM is the data movement upper bound (see Section 5.2.3).

identical to the input data of the job. Conversely, for the reduce phase, the size of the data moved to the FPGA depends on the output of the map phase. This explains why we observe such variability.

HetMR decisions

Next, we want to assess the correctness of the decision taken by the deployment manager. In Figure 5.10, we show the throughput achieved by our map-intensive jobs when running with the sample data set size fixed; but, to model different input data, on the x-axis we vary a job-specific (data-dependent) parameter. For example, for Logistic Regression we vary both the number of dimensions per point and number of points simultaneously, keeping the input size fixed at 1 GB. For the Wikipedia data set, we sample from the distribution of title lengths (based on percentiles).

The goal of these charts is twofold. First, they confirm that while size is not a critical parameter of the input size, the *type* of data has a direct impact on the throughput achieved by the job. This is the reason why HetMR cannot use static information but it has to rely on the profiling stage to derive the expected throughput. The second important result shown in the charts is that our deployment manager always makes the correct decision. Interestingly, just looking at the PCIe throughput would have not given the correct results in some cases, e.g., in the StM-Lev.

The reason is that, as explained in Section 5.3.3, in order for the job to efficiently run on the accelerator, the input data must be pre-processed to change the data layout in memory. In this specific case, it turns out that this pre-processing stage was actually dominating the execution time, thus outweighing the expected benefits. This is clearly visible in Figure 5.11 in which we show the breakdown of the execution time. When the input data size increases, the cost of changing the data layout becomes more prominent and for large input this becomes the main bottleneck.

5.4 Engineering a heterogeneous stream processing system

This section discusses our design and implementation of a heterogeneous stream processing system built on top of SEEP [16, 17]. This is part of our ongoing work to devise software engineering methodology and architectures for heterogeneous dataflow processing systems, namely Task T6.4. This task started at

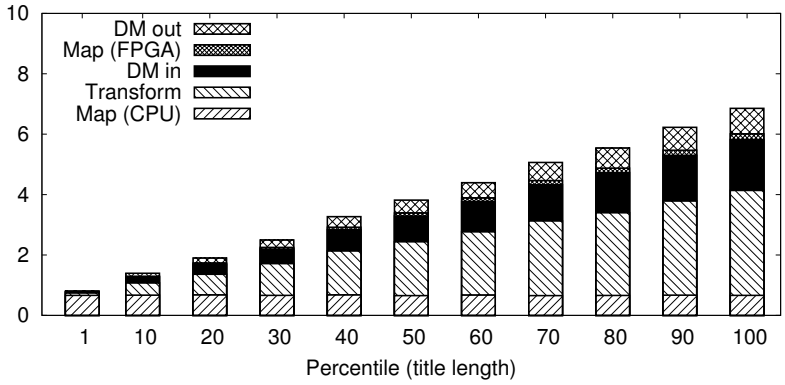


Figure 5.11: Map phase breakdown for StM-Lev.

Month 19. We have already made enough progress to report our initial findings here. A full report on this work is scheduled for M36 in Deliverable D6.4.

The goal of Task T6.4 is to develop software engineering methodology and a set of design patterns for heterogeneous cloud applications. Earlier in this chapter we presented a preliminary classification of prominent cloud application frameworks according to their computational, programming, and execution models (Table 5.1). These frameworks provide high-level programming abstractions (e.g., dataflow or map/reduce) that facilitate the mapping of modern applications (e.g., graph processing or machine learning) to homogeneous resources, while the underlying system automatically handles issues of state management, data parallelism, and failure recovery: some systems offer high processing throughput [23], other systems offer low latency [8, 16], while others offer both [48, 17]. In HARNESS, we seek to increase the efficiency of mapping decisions of individual application components (e.g., tasks in a dataflow) to heterogeneous resources.

5.4.1 Motivation

HARNESS, thus far, has been focusing only on batch applications. Batch processing did well out of the "pay-as-you-go" model of cloud computing, as users could scale out their applications to a large number of homogeneous, cloud-hosted machines to increase their data processing throughput.

However, users nowadays also expect "fresh", low-latency results. Typical applications with such a requirement include online advertising, financial data analysis and Web log mining. These applications are best served by stream processing frameworks, to which users can issue queries and get timely results from high-throughput input data streams. With this work, we aim to **broaden the scope of applications** supported by the HARNESS platform by adding data stream processing ones to its portfolio.

We have already integrated SEEP with the HARNESS platform, described in Deliverable D6.3.2 [31]. In this section, we detail extensions to SEEP's software stack that enable users to submit and execute **sliding-window queries** over data streams on a heterogeneous hardware infrastructure.

5.4.2 Programming model

SEEP exposes a low-level operator assembly interface, upon which high-level query languages and applications can be built. Using this interface, we have implemented a compiler for the Continuous Query Language (CQL) [10], a prominent SQL-based declarative language for registering continuous queries against streams and updatable relations. The semantics of CQL are tightly-coupled with the window semantics of a data stream.

Window semantics. A data stream is an infinite series of tuples $t = \langle \tau, a_1, \ldots, a_k \rangle$, each tuple consisting of a timestamp τ and a set of k attributes $\{a_i\}_{i=1}^{k}$. A *sliding window* over a stream specifies a moving view that decomposes the stream into overlapping bags of tuples. The overlap is determined by the *slide* of a window. A window can be either *time-based*, in which case the range of computable tuples is defined over timestamps, or *count-based*, in which case range is defined over the last N tuples appended to the stream. If the slide equals the window range, then there is no overlap and the window is *tumbling*.

Examples. Using the aforementioned window semantics, a user can write sliding-window queries of the form:

select $\star$ from S **[** range x slide y **]**
where $P(S.a_1, \ldots, S.a_k)$,

where S is a data stream; P is a selection predicate over its attributes; and, x and y are variables that define a window over S.[2] For example, x can be "10 seconds" (a time-based window) or "5 rows" (a count-based window). Sliding windows are usually used to produce aggregate results:

select count($\star$) from S **[** range x slide y **]**
group by $S.a_1, \ldots, S.a_{i<k}$;

or, joined with other sliding windows to produce correlated results:

select $S.a, R.b$ from S **[** range x_1 slide y_1 **]**, R **[** range x_2 slide y_2 **]**
where $P(S.a, R.b)$.

5.4.3 System model

A query application consists of one or more *sub-queries*, expressed in CQL. A sub-query takes a number of input streams, processes their tuples and produces an output stream. When compiled, a sub-query translates into a pipeline of *micro-operators* (Figure 5.12). Each micro-operator implements typical CQL operators, such as selection, projection, aggregation, n-ary join, and so on; users also have the ability to write custom micro-operator code that implements some application-specific function.

Multiple sub-queries are translated to a logical dataflow graph, linked by typed connectors — types are determined by the schemas of input and output streams. We bundle sub-queries and their connectors together into a *multi-operator* (Figure 5.13). The role of a multi-operator is to manage the timely execution of micro-operators and handle data exchanges in-between data processing stages based on the window semantics of each sub-query.

[2] If $x > y$, the window is sliding; if $x = y$, the window is tumbling.

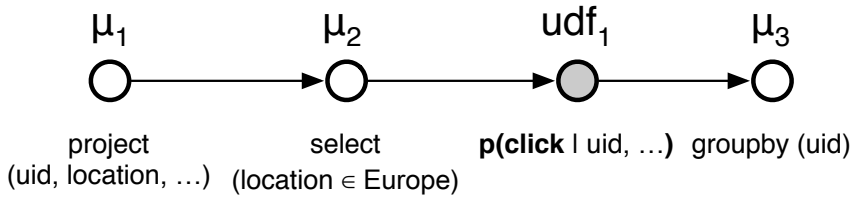


Figure 5.12: An example micro-operator pipeline in SEEP, inspired by the AdPredictor use case [28]. This particular sub-query first filters certain features of an ad click log stream (project and select), updates their probability distribution with a user-defined function (udf_1) and aggregates results by key, building a model per user (uid).

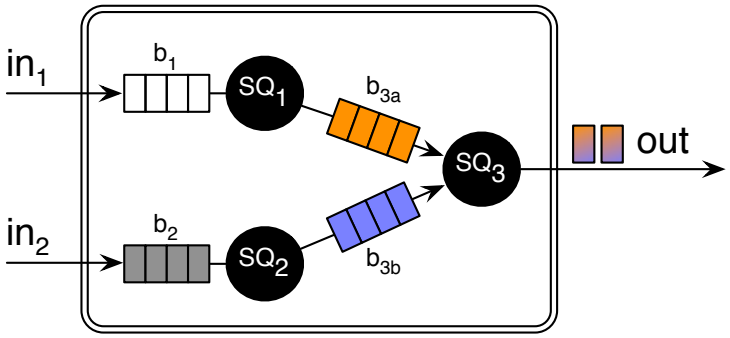


Figure 5.13: A multi-operator in SEEP. Sub-queries $SQ_1 : b_1 \longrightarrow b_{3a}$ and $SQ_2 : b_2 \longrightarrow b_{3b}$ consume input streams in_1 and in_2 and forward their results to sub-query $SQ_3 : b_{3a}, b_{3b} \longrightarrow out$ that joins the two intermediate streams and produces the final output stream.

Query execution. Connectors between sub-queries are realized in our system as in-memory circular buffers, each associated with a particular sliding-window definition (*viz.* its type, range, and slide). For example, sub-queries SQ_1 and SQ_3 in Figure 5.13 are connected via buffer b_{3a}. Our execution model is a hybrid of both scheduled and pipelined tasks. When a tuple is inserted into a buffer, the multi-operator checks whether a window has slid, at which point it may launch a sub-query processing task. These tasks are executed in parallel by a thread pool using cooperative scheduling. During a task execution, the micro-operators of a sub-query are executed in a pipeline fashion. Scheduled tasks ensure that all cores in a machine are utilized enough to saturate the input data throughput of the system; while pipelined tasks ensure cache locality when processing intermediate results.

5.4.4 Data parallelism

Currently, we are considering a single-machine execution environment that consists of a multi-core CPU and a GPGPU co-processor. We choose a GPGPU as an accelerator because a lot of stream processing algorithms (e.g., set operations, streaming joins, and sliding-window aggregates) can saturate the ample memory bandwidth this device offers.

SEEP relies on data parallelism to increase the aggregate computation and memory bandwidth of the heterogeneous resources available to a query application. **Data parallelism is achieved by batching**

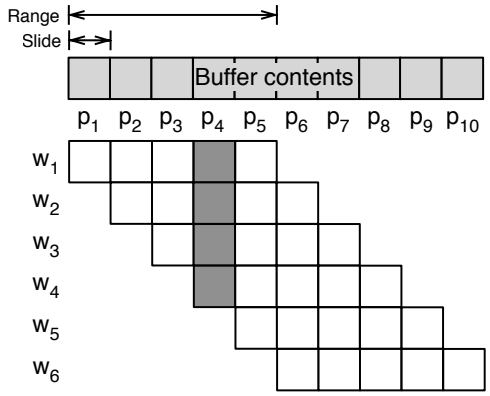


Figure 5.14: A batch of sliding windows, each composed of 5 panes. A pane (e.g., p_4) can belong to more than one window.

multiple windows together into a single task. With batching, we can also use the concept of *panes* [43] to improve the throughput of the system. The pane-based approach is illustrated in Figure 5.14. Panes are non-overlapping sub-windows whose range is the slide of the original window definition. The number of panes per window is:

$$\frac{\text{range}}{\gcd(\text{range}, \text{slide})}$$

By default, sliding windows have an overlap that results in redundant data processing. For example, p_4 in Figure 5.14 belongs to windows w_1, w_2, w_3, and w_4. Using panes for evaluating sliding-window queries, we can reduce the computation cost by sharing the results of, say, p_4 when computing windows w_1–w_4. Our approach is to first divide the stream into disjoint panes, process them in parallel, and then aggregate the pane-level results to get the final window output. This way we avoid redundant computations, especially when panes contain a lot of tuples which can be reduced to a much smaller set. For example, in the case of sliding-window aggregate, a pane reduces to a set of unique key/value pairs. With batching, we can use the multiple threads of a CPU/GPGPU machine to perform pane-level and window-level computations.

The data parallel execution model is the same for both the GPGPU and CPU. By using the same interface, we can seamlessly interchange sub-query tasks across the two processors based on sliding window query semantics. The GPGPU is managed by a single-threaded task queue service, handling issues of data movement between the two processors automatically.

Outlook

A goal of Task T6.4 is to develop a software engineering methodology and a set of design patterns for heterogeneous cloud applications with which developers can hint a cloud platform with their non-functional concerns (e.g., low latency and/or high data throughput). For query applications, we have currently extended SEEP to support sliding-window queries that can run on heterogeneous hardware. In Year 3, we plan to leverage experiences with the LARA aspect-oriented language, discussed in Deliverable D3.1 [29], to *weave* such non-functional requirements with query operators.

6 Conclusion

Automatic performance modeling is an essential component of the HARNESS platform: it allows the system to automate the choice of heterogeneous cloud resources which should be assigned to an application in order to achieve the user's requirements. We have proposed two complementary techniques to automatically profile cloud applications: (i) a blackbox approach applies to arbitrary batch applications, but it may require a significant number of profiling executions before capturing all the relevant specificities of a new application; (ii) a whitebox approach applies domain-specific knowledge about MapReduce applications to considerably speed up the blackbox performance modeling process. We have shown that both approaches provide complementary benefits, and can respectively profile all the use cases of the project.

We also reported our initial findings with respect to the task T6.4 (*"Devise software engineering methodology and architectures"*), and presented our plans for extending the application domain targeted by HARNESS in the direction of stream processing applications.

Our plans for Y3 include merging the blackbox and whitebox modeling approaches in a single performance profiling component, improving the algorithms used in both parts of the system, and finalizing the integration of these models in the HARNESS platform prototype.

Bibliography

[1] F. N. Afrati, A. D. Sarma, D. Menestrina, A. Parameswaran, and J. D. Ullman. Fuzzy joins using MapReduce. In *Proceedings of the IEEE ICDE Conference*, 2012.

[2] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: optimizing MapReduce on heterogeneous clusters. In *Proceedings of ASPLOS Conference*, 2012.

[3] R. Allan. Survey of HPC performance modelling and prediction tools. Technical Report DL-TR-2010-006, Science and Technology Facilities Council, July 2009.

[4] Amazon Web Services. Choosing the right EC2 instance type for your application. `http://aws.amazon.com/blogs/aws/choosing-the-right-ec2-instance-type-for-your-application/`.

[5] AMD. Aparapi. `http://code.google.com/p/aparapi/`.

[6] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: a language and compiler for algorithmic choice. In *Proceedings of the ACM SIGPLAN Conference*, pages 38–49, 2009.

[7] Apache Software Foundation. S4: distributed stream computing platform. `http://incubator.apache.org/s4/`.

[8] Apache Software Foundation. Storm: distributed and fault-tolerant realtime computation. `https://storm.incubator.apache.org`.

[9] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for Hadoop: Time to rethink? In *Proceedings of the ACM SoCC Conference*, 2013.

[10] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[11] D. Bacon, R. Rabbah, and S. Shukla. FPGA programming for the masses. *Queue*, 11(2):40–52, Feb. 2013.

[12] T. H. Beach, O. F. Rana, and N. J. Avis. Integrating acceleration devices using CometCloud. In *Proceedings of the ORMaCloud workshop*, June 2013.

[13] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: MapReduce for incremental computations. In *Proceedings of the ACM SoCC Conference*, 2011.

[14] P. Brebner and A. Liu. Modeling cloud cost and performance. In *Proceedings of the Conference on Cloud Computing and Virtualization*, May 2010.

[15] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, Sept. 2010.

[16] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the ACM SIGMOD Conference*, 2013.

[17] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Making state explicit for imperative big data processing. In *Proceedings of the USENIX ATC Conference*, 2014.

[18] L. Chen and G. Agrawal. Optimizing MapReduce for GPUs with effective shared memory usage. In *Proceedings of the ACM HPDC Conference*, 2012.

[19] L. Chen, X. Huo, and G. Agrawal. Accelerating MapReduce on a coupled CPU-GPU architecture. In *Proceedings of the SC Conference*, 2012.

[20] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-Reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems 19*, pages 281–288. 2007.

[21] E. S. Chung, J. D. Davis, and J. Lee. LINQits: Big data on little clients. *ACM SIGARCH Computer Architecture News*, 41(3):261–272, June 2013.

[22] CopperEgg. AWS sizing tool. `http://copperegg.com/aws-sizing-tool/`.

[23] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the USENIX OSDI Conference*, 2004.

[24] J. Dejun, G. Pierre, and C.-H. Chi. EC2 performance analysis for resource provisioning of service-oriented applications. In *Proceedings of the Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing*, Nov. 2009.

[25] J. Dejun, G. Pierre, and C.-H. Chi. Resource provisioning of Web applications in heterogeneous clouds. In *Proceedings of the USENIX WebApps Conference*, June 2011.

[26] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift. More for your money: exploiting performance heterogeneity in public clouds. In *Proceedings of the ACM SOCC Conference*, Oct. 2012.

[27] H. Fernandez, G. Pierre, and T. Kielmann. Autoscaling web applications in heterogeneous cloud infrastructures. In *Proceedings of the IEEE IC2E Conference*, Mar. 2014.

[28] FP7 HARNESS Consortium. Industrial requirements. Project Deliverable D2.2, 2013.

[29] FP7 HARNESS Consortium. Characterisation report. Project Deliverable D3.1, 2013.

[30] FP7 HARNESS Consortium. Application characterisation report. Project Deliverable D6.1, 2013.

[31] FP7 HARNESS Consortium. Heterogeneous platform implementation (updated). Project Deliverable D6.3.2, 2014.

[32] FP7 HARNESS Consortium. Integrated HARNESS platform and validation use cases (initial). Project Deliverable D7.2.1, 2014.

[33] R. Gandhi, D. Xie, and Y. C. Hu. PIKACHU: How to rebalance load in optimizing MapReduce on heterogeneous clusters. In *Proceedings of the USENIX ATC Conference*, 2013.

[34] Grid'5000. http://www.grid5000.fr/.

[35] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the USENIX OSDI Conference*, 2010.

[36] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the PACT Conference*, 2008.

[37] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: Batched stream processing for data intensive distributed computing. In *Proceedings of the ACM SoCC Conference*, 2010.

[38] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. MapCG: writing parallel program portable between CPU and GPU. In *Proceedings of the PACT Conference*, 2010.

[39] E. Ipek, B. de Supinski, M. Schulz, and S. McKee. An approach to performance prediction for parallel applications. In *Proceedings of the Euro-Par Conference*, 2005.

[40] W. Jiang and G. Agrawal. MATE-CG: A map reduce-like framework for accelerating data-intensive computations on heterogeneous clusters. In *Proceedings of the IEEE IPDPS Conference*, 2012.

[41] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 2010.

[42] H. Khazaei. *Performance Modeling of Cloud Computing Centers*. PhD thesis, University of Manitoba, Oct. 2012.

[43] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *ACM SIGMOD Record*, 34(1):39–44, Mar. 2005.

[44] R. J. Lipton and D. Lopresti. A systolic array for rapid string comparison. In *Proceedings of the Chapel Hill Conference on VLSI*, 1985.

[45] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *Proceedings of the ACM SoCC Conference*, 2010.

[46] U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso. A survey of performance modeling and simulation techniques for accelerator-based computing. *IEEE Transactions on Parallel and Distributed Systems*, To appear.

[47] Maxeler Technologies. MPC-C Series. `http://www.maxeler.com/products/mpc-cseries/`.

[48] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceeding ACM SOSP Conference*, 2013.

[49] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: A universal execution engine for distributed data-flow computing. In *Proceedings of the USENIX NSDI Conference*, 2011.

[50] OCCI-WG.org. OCCI, accessed on August 4th 2014. `http://occi-wg.org/`.

[51] A.-M. Oprescu, T. Kielmann, and H. Leahu. Budget estimation and control for bag-of-tasks scheduling in clouds. *Parallel Processing Letters*, 21(2), June 2011.

[52] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable performance on heterogeneous architectures. In *Proceedings of ACM ASPLOS Conference*, 2013.

[53] S. Pllana, I. Brandic, and S. Benkner. A survey of the state of the art in performance modeling and prediction of parallel and distributed computing systems. *International Journal of Computational Intelligence Research*, 4(1), 2008.

[54] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the USENIX OSDI Conference*, 2010.

[55] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the HPCA Symposium*, 2007.

[56] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the ACM SOSP Conference*, 2013.

[57] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang. FPMR: MapReduce framework on FPGA. In *Proceedings of the ACM/SIGDA Symposium on Field Programmable Gate Arrays*, 2010.

[58] S. Singh. Computing without processors. *Communications of the ACM*, 54(8):46–54, Aug. 2011.

[59] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the WWW Conference*, 2011.

[60] T.-W. Sze. The two quadrillionth bit of Pi is 0! Distributed computation of Pi with Apache Hadoop. In *Proceedings of the IEEE CloudCom Conference*, pages 727–732, 2010.

[61] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular MapReduce for shared-memory systems. In *Proceedings of the Workshop on MapReduce and its Applications*, 2011.

[62] W. Tang, N. Desai, D. Buettner, and Z. Lan. Job scheduling with adjusted runtime estimates on production supercomputers. *Elsevier Journal of Parallel and Distributed Computing*, 73(7):926–938, 2013.

[63] N. Vasic, D. Novakovic, S. Miucin, D. Kostic, and R. Bianchini. DejaVu: Accelerating resource allocation in virtualized environments. In *Proceedings of the ACM ASPLOS Conference*, Mar. 2012.

[64] Wikipedia.org. Simulated annealing. `http://en.wikipedia.org/wiki/Simulated_annealing`.

[65] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *Proceedings of the IEEE Symposium on Workload Characterization*, 2009.

[66] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the USENIX OSDI Conference*, 2008.

[67] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user MapReduce clusters. Technical Report UCB/EECS-2009-55, University of California, Berkeley, April 2009.

[68] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the USENIX NSDI Conference*, 2012.

[69] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the ACM SOSP Conference*, 2013.