FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Predicting Scalability of Standalone Applications in Cloud Environments

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Gregor Schauer
Matrikelnummer 0926086

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar
Mitwirkung: Dipl.-Ing. Michael Vögler
Dr. Rostyslav Zabolotnyi

Wien, 01.01.2016

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Predicting Scalability of Standalone Applications in Cloud Environments

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Gregor Schauer
Registration Number 0926086

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dr. Schahram Dustdar
Assistance: Dipl.-Ing. Michael Vögler
Dr. Rostyslav Zabolotnyi

Vienna, 01.01.2016

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Gregor Schauer
Bendikstraße 1/40, 4300 St. Valentin

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Danksagung

Die vorliegende Diplomarbeit läutet das Ende meines Informatikstudiums ein. An dieser Stelle möchte ich mich bei jenen Menschen bedanken, die maßgeblich zum Erfolg beigetragen haben. Mein Dank gebührt insbesondere meinen Eltern, die mich in jeglicher Hinsicht unterstützt und mir viel Geduld und Verständnis entgegengebracht haben. Sie haben mir den Weg geebnet, sodass ich jetzt auf eine erfolgreiche Studienzeit zurückblicken kann. Deshalb möchte ich mich nochmal herzlich für alles bedanken, was sie für mich getan haben.
Desweiteren möchte ich mich für die Möglichkeit die Diplomarbeit am Institut für Informationssysteme zu schreiben bei Prof. Schahram Dustdar sowie bei Michael Vögler für die umfassende Betreuung bedanken. Ebenso möchte ich mich bei Philipp Leitner und Rostyslav Zabolotnyi für die Betreuung und das konstruktive Feedback während der ersten Phase der Diplomarbeit bedanken.
Ein großes Dankeschön gilt auch Freunden und Studienkollegen, die mich während der Studienzeit unterstützt haben. Allen voran Dominik Strasser und Bernhard Nickel. Wir haben die meisten Lehrveranstaltungen gemeinsam absolviert und haben dabei viel gelernt und auch abseits des Studiums viel erlebt.
Weiters möchte ich mich bei meinen Arbeitskollegen Stefan Mader und Stefan Sevelda für das Korrekturlesen der Diplomarbeit bedanken. Mein Dank gebührt natürlich auch meinem Chef Andreas Halwein, der mir ermöglicht hat während der letzten vier Jahre Berufsalltag und Studium unter einen Hut zu bringen. Danke auch allen Arbeitskollegen, von denen ich in dieser Zeit viel lernen durfte.

Abstract

Cloud computing opens up a variety of options, such as improving availability or fault tolerance of applications. However, it might also entail non-negligible overhead. There are different approaches for identifying and reducing overhead (e.g., by using a monitoring software). Most of those solutions are reactive and do not offer a way to do accurate predictions about scalability of applications running on cloud platforms.

This thesis deals with the issue and considers the question of how to predict certain performance characteristics of an application without running it on the target platform. Generally accepted models such as Amdahl's and Gustafson's Law were examined for their applicability to cloud computing. Based on them, a model, which mathematically describes the scalability of applications under consideration of cloud specific properties, was developed.

The model was evaluated on two applications with different load profiles. For this purpose, a lightweight profiler has been implemented to gather runtime information of the distributed applications. The collected data were filtered, clustered by thread, and aggregated by method and class level. Then they were interpolated and compared with the performance predictions. The deviations were analyzed and causes like garbage collection were discussed. The work shows that, for example, minor changes in the application can have a significant impact on the performance characteristics of the application. Models such as the one developed in the context of this thesis provide valuable information (e.g., scalability and technical limits).

Kurzfassung

Cloud Computing eröffnet eine Vielzahl an Möglichkeiten wie z.B. die Verfügbarkeit oder Fehlertoleranz von Applikationen zu verbessern. Dabei entsteht aber auch ein unter Umständen nicht vernachlässigbarer Overhead. Es gibt verschiedene Ansätze um diesen, beispielsweise mittels Monitoring, zu identifizieren und zu vermindern. Diese großteils reaktiven Lösungen bieten jedoch keine Möglichkeit vorab hinreichend genaue Vorhersagen über Skalierbarkeit von Applikationen auf Cloud Plattformen zu treffen.

Die vorliegende Diplomarbeit beschäftigt sich mit dieser Thematik und geht unter anderem der Frage nach ob und wie sich gewisse Performanceeigenschaften einer Applikation vorhersagen lassen ohne diese auf der Zielplattform zu betreiben. Dazu wurden anerkannte Modelle wie jene von Amdahl und Gustafson auf ihre Anwendbarkeit für Cloud Computing untersucht. Darauf aufbauend wurde ein Modell entwickelt, welches die Skalierbarkeit von Programmen unter Berücksichtigung von Cloud-spezifischen Eigenschaften mathematisch beschreibt.

Anhand zweier Applikationen mit unterschiedlichen Lastprofilen wurde dieses Modell evaluiert. Dazu wurde ein leichtgewichtiger Profiler implementiert um Laufzeitinformationen der verteilten Applikationen zu sammeln. Die so gewonnen Daten wurden gefiltert, nach Thread geclustert und auf Methoden- und Klassenebene aggregiert. Anschließend wurden sie interpoliert und mit den Vorhersagen über die zu erwartende Performance verglichen. Desweiteren wurden Abweichungen analysiert und deren Ursachen wie Garbage Collection diskutiert. Die Arbeit zeigt, dass beispielsweise im Zuge einer Applikationsmigration entstehende geringfügige Änderungen die Performancecharakteristika der Applikation signifikant verändern können. Modelle, wie jenes, das im Rahmen der Arbeit entwickelt wurde, können wertvolle Informationen über beispielsweise Skalierbarkeit und technische Limits liefern.

Contents

List of Figures

List of Tables

Listings

CHAPTER 1

Introduction

"Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."

— Lewis Carroll
(Alice in Wonderland)

In the last decade the role of cloud computing has become more and more important. Cloud computing is a computation model where many servers communicate via a network in order to handle high load. The origin of this approach is not known because the basic idea emerged long time before it was possible to build large clouds in a simple way. This is also the reason why there is no single definition. National Institute of Standards and Technology (NIST) published a definition [34], which is accepted generally. According to their definition, cloud computing has the following five characteristics: broad network access, resource pooling, rapid elasticity, on-demand self-service and measured services. Especially the first three characteristics build the foundation for the first publicly known clouds.

Companies, which do e-commerce business or offer other services to many people, have to deal with fluctuating load. For example, on the day when a new product is released, more than ten times as many orders have to be processed than on a normal day. Cloud computing enables companies to handle such situations easily while keeping the overall operational costs as low as possible. Due to the fact that the amount of servers used for handling the requests can be increased within a few minutes by buying more computation power from a cloud provider, a potential overload of the system can be prevented. In other words, it can be ensured that the system remains available all the time.

Cloud platforms can also be classified by their service model. The definition of NIST enumerates three different models, which can be summarized as follows:

- Infrastructure as a Service (IaaS): Only the basic infrastructure such as storage is provided. The client has control over the operating system and all the software running on the machines.
- Platform as a Service (PaaS): The client has no control about hardware and some software like a web server is provided. Applications can be deployed and run on top of the environment.
- Software as a Service (SaaS): The cloud environment is under full control of the provider, who hosts applications that can be used by the client.

While SaaS is a convenient model for end users, PaaS or even IaaS clouds are used whenever certain application properties like consistency, availability and/or fault tolerance play an important role and have to be optimized at all costs. Though, the optimization of one property often entails a degradation of another one. Moreover, distributed computation introduces additional effort like scheduling, synchronization or remote communication. To demonstrate this, let us consider the well known MapReduce model [42]. On the one hand, it supports processing of large amounts of data that would not fit onto a single machine. On the other hand, studies have shown that the performance can vary dramatically [14]. For example, the authors have shown that the selection of a different scheduling strategy may reduce execution time by more than 25%. This and other wrong configuration decisions could be avoided rather easily. In contrast to that, some types of distribution overhead such as network latency often cannot be avoided. These aspects not only have to be considered when it comes to scaling, but also before an application is modified for being run in a cloud environment.

1.1 Motivation

Due to their nature, distributed applications have quite different requirements regarding computational resources such as Central Processing Units (CPUs) or memory. For example, it is quite common to build highly optimized clusters for applications that simulate complex real-world situations such as the impact of certain environment conditions on living beings. When talking about simulations, there could even be opposing requirements for different tasks. For example, in case of predicting the impact of an earthquake on oceanic regions, it is important to aim for speed in order to be able to warn people of a potential tsunami. Therefore, the application is optimized for using all computational resources for delivering the result as fast as possible. Another approach is the usage of all available resources for processing more data in the same period. For instance, when simulating the genetic evolution of a bacteria population, researchers may be interested in getting as accurate results as possible. Using additional computation power for running the same simulation on a bigger population may lead to more knowledge.

As a matter of fact, the costs for building, operating and maintaining clouds are usually quite high. Therefore, cloud providers want to keep costs as low as possible. In many cases, one way to achieve this goal is to reduce the computation power, which also reduces the overhead caused by the distributed system itself. However, finding the right balance between resource provisioning and costs is not that simple. The ability to make accurate predictions of the performance and the overhead of applications running in a cloud environment based on measurements of non-distributed execution influences different aspects in cloud computing a lot. The outcome could be that it is more economic to perform certain tasks in a distributed way whenever a certain threshold is reached. Alternatively, one might conclude that even when the performance increases linearly with the number of computation nodes, the overhead increases exponentially and therefore it might even be advisable to violate Service Level Agreements (SLAs) in order to save costs.

This thesis mainly focuses on scenarios where it is important to aim for performance and/or throughput (see also Chapter 2) and not for economic goals. Since cloud performance optimization is not a trivial topic, lots of research has been done in the past and is still ongoing. Although we go back a step and ask a fundamental question:

Does it actually make sense to make an application cloud-aware?
If yes, how big is the performance improvement expected to be?

In other words, we would like to derive runtime information based on the given conditions and the facts we know from similar types of applications. By having this information available, one could reason about cost amortization and whether there is a potential for optimization or not.

1.2 Research Questions

The challenges outlined in Section 1.1 constitute the necessity of having models that allow the determination of performance properties of applications as accurately as possible without running them in the target environment. The following questions, which are related to the fundamental question in the previous section, are researched in this thesis:

1. How can the speedup of distributable applications running in a cloud environment be predicted by using metrics from non-distributed execution? Do some types of applications have a general model that describes their cloud performance?

2. Is it possible to reason about the point where cloud execution outweighs the distribution overhead including scaling, scheduling and communication?

3. Are there any indications that an application will not scale well in the cloud and how can they be verified? Do they correspond to static models like Amdahl's Law [1] or are there any other factors that have to be considered?

1.3 Methodological Approach

Based on empirically collected runtime information of non-distributed applications and assumptions about the overhead caused by remote communication, synchronization, etc., we attempt to derive information about the scalability of the applications running in a cloud environment. More concretely, the execution time shall be reduced to a minimum by using additional resources effectively. Models like Amdahl's and Gustafson's Law [11] build the foundation for our mathematical model that describes the performance of applications depending on the available resources and amount of data to process. We claim that this approach allows to predict application performance much more precisely than conventional methods like asymptotic analysis. By applying our model to selected applications with high CPU, memory and/or Input/Output (I/O) requirements, we would like to see upfront under which conditions the benefits of parallelism outweigh the cloud overhead. The profiler implemented for evaluating the accuracy of our predictions combines common profiling techniques with Aspect Oriented Programming (AOP) in order to get precise runtime information. Finally, we discuss the deviations between the predictions and the measured results and explain impacts of indeterministic memory management and other observations.

1.4 Scientific Contribution

In order to answer the research questions asked in Section 1.2, the following contributions to the state of the art in cloud computing performance analysis have been made:

- *Contribution 1: an extension to Amdahl's Law as well as Gustafson's Law for reasoning about the expected performance of cloud applications under deterministic conditions.*
  The basic ideas of those laws are also applicable to cloud computing. However, it is necessary to consider cloud-specific properties such as communication overhead and adapt them accordingly. In Chapter 4 the modified rules for *cloud speedup* are introduced.

- *Contribution 2: a case study, which shows the negative impact of inefficient scheduling of small amounts of tasks as well as natural limitations for asymptotic infinite tasks.*
  In order to validate the statements regarding performance predictions, scenarios based on real-world applications have been evaluated. More information about them can be found in Chapter 6 and Chapter 7.

- *Contribution 3: a Java-based profiling library for analyzing runtime performance of distributed applications.*
  Low-level performance information is very crucial for understanding which parts of the application can be parallelized and which have to be invoked sequentially. The *ALASCA* profiler has been implemented for getting in-depth information about applications running in cloud environments and is introduced in Chapter 5. Even though it is very lightweight, it provides all the features required for analyzing runtime performance focusing on CPU and memory utilization of distributed applications. In the evaluation it is used for predicting the scaling capabilities of the application and later on for verification of the predictions.

1.5 Thesis Organization

The remainder of this thesis is organized as follows:

- Chapter 2 gives a short introduction into the terminology and explains the most relevant related concepts and research approaches. Additionally, it describes state-of-the-art computation models and cloud technologies.
- Chapter 3 continues with a selection of relevant related work of performance analysis in the area of cloud computing.
- Chapter 4 discusses theoretical concepts from the previous chapter and presents the ideas and the approach of this thesis. Especially the statistical models used for predicting the performance of applications running in a cloud environment are explained and the impact of the distribution overhead is elucidated.
- Chapter 5 provides a description along with an illustration of the architecture of the profiler implemented within the frame of this thesis.
- Based on the models proposed in Chapter 4, the cloud performance of selected types of applications are predicted in Chapter 6. Then the profiler introduced in Chapter 5 is used to verify or falsify the predictions.
- Chapter 7 concludes the thesis by discussing the accuracy of the predictions and their relevance for similar applications. Last but not least it answers the research questions of Section 1.2.

CHAPTER 2

State of the Art

Where would we be if we all just sat there and said 'where would we be now', but nobody was prepared to go and find out where we would be if we actually went?

— Kurt Marti
(english translation)

Regardless of the optimization goal of a distributed application, there are several ways to classify it. As described in Section 1.1, in this thesis we do not focus on the economic point of view. Instead we are interested whether certain types of applications benefit from cloud computing and what the performance potential is expected to be. Assuming that an application is suited for distributed execution, one has two choices that are orthogonal to each other. On the one hand, one may want to get as much speedup as possible (i.e., reduce the execution time to a minimum). On the other hand, one may use the additional computation power to process more data. While it is a legitimate goal to achieve both, this is not possible without having any trade-off. However, as both approaches have drawbacks, their combination might be more efficient than optimizing for one or the other.

In this chapter, we first describe the more natural approach for scalability (reducing the execution time), which is described by Amdahl's Law. Secondly, we have a look at Gustafson's Law, which is probably more suitable for data-intensive applications [16]. Subsequently common computation models are explained. Finally, the chapter concludes with a brief introduction into some technologies used in cloud environments.

2.1 Amdahl's Law

Amdahl's Law [1] describes the potential speedup of applications given a fixed sized problem using various machines, which support parallelism differently. Amdahl and his colleagues defined *speedup* as the sequential run time divided by the parallel run time:

Definition 1 (speedup)**.**

$$speedup = \frac{s+p}{s+\frac{p}{N}} = \frac{1}{s+\frac{p}{N}}$$

Whereas N is the number of processing units, s and p denote the fraction of the instructions that are processed in serial order and in parallel, respectively. The larger the fraction of the code that supports parallelism, the smaller is the execution time using the machine that supports parallelism best. Note that for the sake of simplicity, it is assumed that every instruction can be processed in the same amount of time.

The total execution time t_t can be defined by using the definition of speedup after rearranging the equation and multiplying it by t_t:

$$t_t = t_t \cdot (s + \frac{p}{N}) = (t_t \cdot s) + (t_t \cdot \frac{p}{N}) = t_s + t_p$$

Figure 2.1 depicts the relative execution time with different amounts of parallelizable instructions and an increasing number of processing units compared to serial processing. As it can be seen, the higher the amount of parallelizable instructions, the bigger is the speedup and the less is the execution time.
Furthermore, it can be concluded that the relative execution time converges against the time that is spent for serial processing t_s. The proof can be found in Chapter 4.

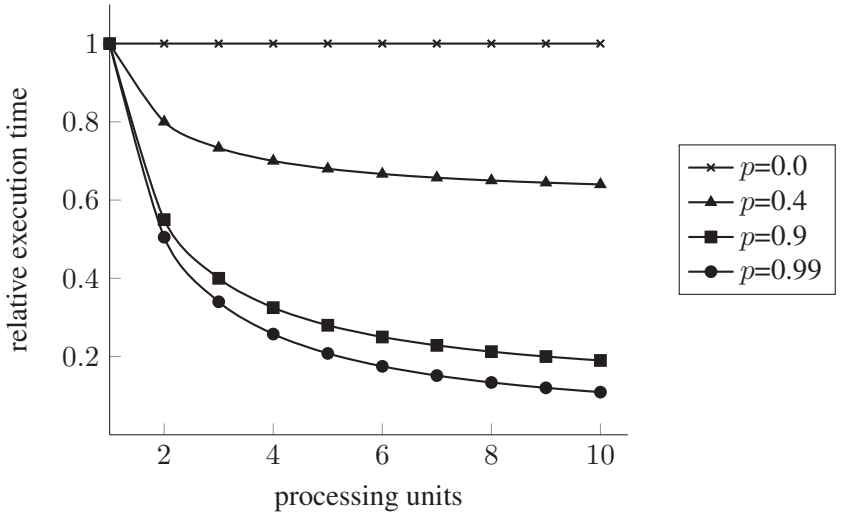

Figure 2.1: Relative execution time with an increasing number of processing units

2.2 Gustafson's Law

Gustafson et al. state that with an increasing number of processing units, the problem size also scales [11]. In other words, the additional computation power can be used to process more data instead of reducing the execution time. Thus the total execution time t_t can be defined as follows:

$$t_t = t_s + \left\lceil \frac{t_p \cdot n}{N} \right\rceil$$

Where N is the number of available computation nodes, n is the amount of parallel tasks, and t_p and t_s represent the amount of time of a single task and spent for serial processing, respectively. For this approach the term speedup has to be defined differently because this time, the size of the problem scales up as well. To distinguish between them, this kind of speedup is called *scaled speedup* and is defined in [11] as follows:

Definition 2 (Scaled speedup)**.**

$$\begin{aligned} Scaled\ speedup &= (s + p \cdot N)/(s + p) \\ &= s + p \cdot N \end{aligned}$$

Whereas N is the number of processing units, and s and p denote the fraction of the instructions that are processed in serial order and in parallel, respectively. Figure 2.2 shows the number of tasks that can be processed within a fixed time range with an increasing number of processing units.

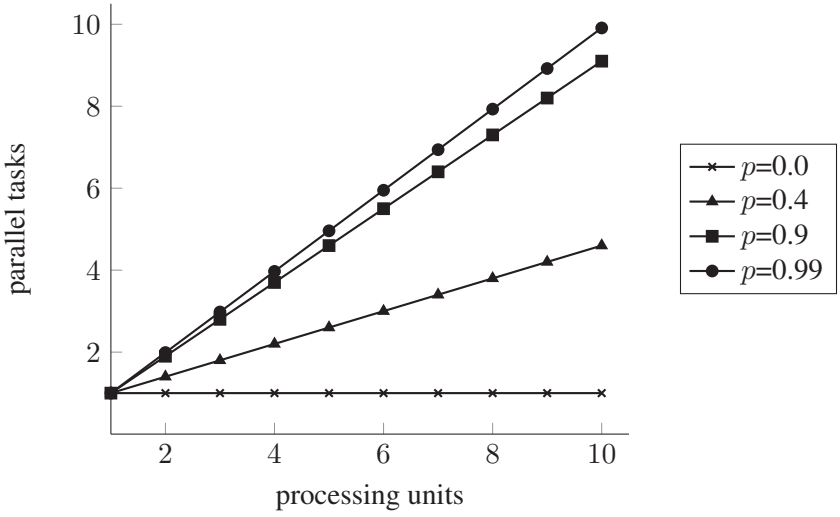

Figure 2.2: Parallel tasks processed with an increasing number of processing units

2.3 Asymptotic Complexity

The execution time of a program is the sum of the duration of executed primitive operations. It is convenient to assume that operations require a constant amount of time. Even when different durations are associated with every operation, the analysis of asymptotic complexity states the runtime effort for *large input*. The notion of input size depends heavily on the type of problem. For example, it can be measured as the number of items to sort or the amount of memory to search for a certain element. Due to the fact that we are interested in problems with large inputs and the running time of many algorithms is directly proportional to the input size, one can omit constants, because they do not influence the asymptotic complexity. Furthermore, we assume that the duration of every operation takes the same amount of time. Besides that, it often is assumed that those operations are machine-independent. In other words, there are no CPU-instructions like vector operations, which violate the assumptions stated above to some extent.

O-notation

Asymptotic complexity describes the order of growth of the running time. Therefore, not only constants can be omitted, but also any terms with a lower order than the term with the highest order. For example, the asymptotic complexity of the term $a \cdot n^2 + b \cdot n + c$ is $a \cdot n^2$ or simply just n^2. The lower terms have no significant impact for large values of n. Although algorithms, which have a lower order of growth, might take longer than algorithms whose order of growth is higher, but the constants are relatively smaller. For example, an algorithm with complexity n^3 is faster than an algorithm with $k \cdot n^2$ for large k and small n.

The O-notation describes the asymptotic upper bound of an algorithm. In this particular example, n^2 is the lowest upper bound for the algorithm (written as $O(n^2)$). Any function with a larger upper bound (e.g., n^k for $k > 2$, $n!$ or n^n) is also an asymptotic upper bound. When talking about the asymptotic upper bound, one refers to the lowest upper bound.

Θ-notation

There is also an asymptotic lower bound. However, the lower bound is less important than the upper bound unless it is equal to the upper bound. In this particular case the asymptotic complexity can be specified in Θ-notation. For example, $\Theta(n^2)$ states that the runtime complexity of the algorithm is quadratic on the size of the input n.

Applicability on Cloud Computing

Asymptotic complexity analysis is often used for describing the general complexity of algorithms. When talking about cloud computing, we are primarily not interested in the complexity, but in the runtime behavior of the actual application. On the one hand, distributed execution entails additional computation effort that might have a significant impact on execution time. On the other hand, the algorithm might not be suitable for being used in a distributed application. Thus the implementation might have a higher runtime complexity than expected.

For example, an application, which implements an algorithm with asymptotic complexity of n^2, could have a runtime complexity of $n^{2.5} + k$ when running in a cloud environment. That means, in addition to some fixed overhead k, the application does not scale with the problem size n. In other words, it is not advisable to run the application in a cloud environment, because the overhead outweighs the performance gain.

2.4 Parallelism

Although Amdahl's and Gustafson's Law describe contrary approaches, they are related to each other and have something in common. Having twice as many resources does not necessarily imply that the computation is twice as fast or twice as much data can be processed. It depends on the runtime complexity of the application. Amdahl and Gustafson also assumed that the serial part does neither grow nor shrink with increasing resources or input.

Besides that, the overhead was omitted in the formulas above. Since the overhead might change with the amount of processing units used or the problem size, it cannot be assumed to be constant and therefore not added to the non-parallelizable computation time. Even though those formulas are not suited for concrete performance predictions of distributed applications, they are the foundation for the model we introduce later. To be more specific, they have to be extended by adding a function that calculates the overhead depending on various parameters like number of processing units or fraction of parallelizable code. This is explained in detail in Chapter 4.

2.5 Computation Models

MapReduce

MapReduce [5] is a distributed computation model for processing large amounts of data. The name is taken from its main steps, which use two functions called *map* and *reduce*. Figure 2.3 gives an overview of the model. The basic idea is to split the data and process it in parallel by multiple workers, which map the parts to an intermediate result. Finally, these independent results are reduced to the final result. Thus MapReduce is ideally suited for use cases where the task can easily be split into hundreds or thousands of subtasks that can be processed independently. Due to the large amount of data, it might be a suitable approach to reduce consistency requirements and avoid costly operations like synchronization in order to gain additional speedup.

One can conclude that for some use cases it is impractical or even impossible to collect a consistent snapshot of up to several terabytes of data and keep it until the computation is finished. However, due to the fact that large data sets contain billions of records many operations like calculating the arithmetic mean of a list of values is precise enough even if some are omitted or have been altered in the meantime. This way, MapReduce is heavily optimized for throughput and is a standard model for such use cases.

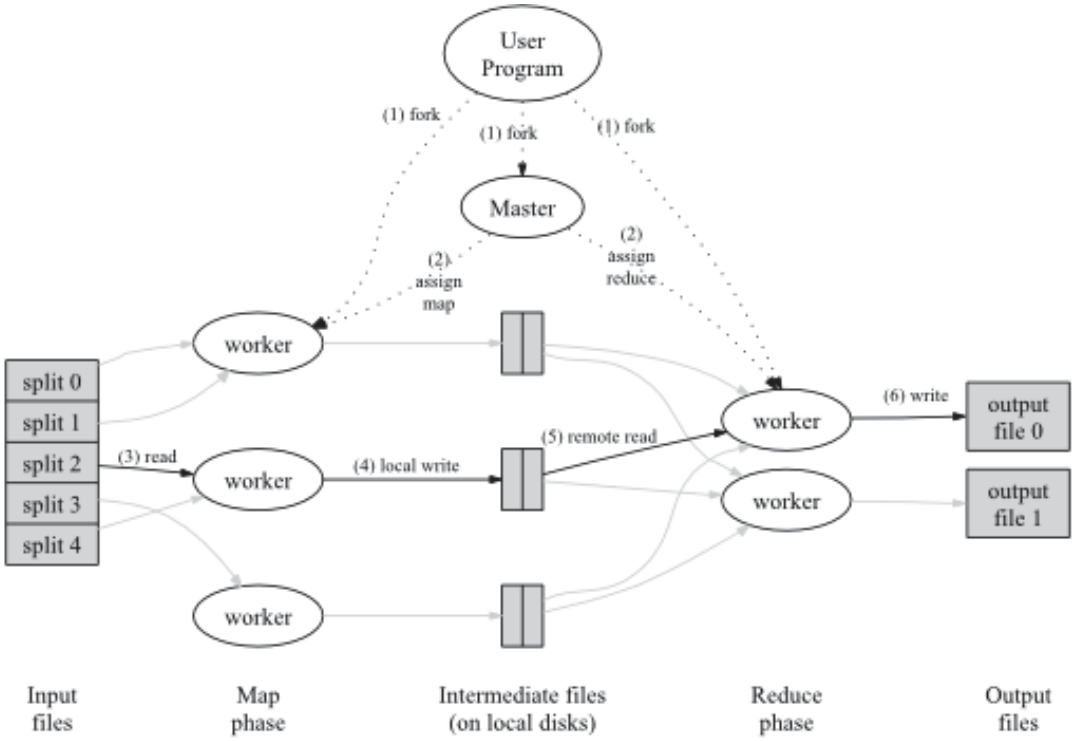

Figure 2.3: MapReduce execution overview [5]

A popular framework, which implements the MapReduce model, is Apache Hadoop[1]. When a master node receives a request (so called job), it splits the data into partitions of equal size (e.g., 128 MB each), and distributes them among the available worker nodes. The workers fetch the data assigned to them and perform the mapping task. First, the map function takes key-value pairs as input and generates intermediate key-value pairs. Afterwards, the reduce function combines all key-value pairs with the same key and produces a result. The data is often stored on a distributed file system like Hadoop Distributed File System (HDFS). It uses the Transmission Control Protocol (TCP) for replicating blocks of data to some other data nodes and reduce potential bottlenecks. After finishing the task, the data is written to the distributed storage until all parts required for the next step are processed. Finally, when all the reduce tasks are completed, the result can be sent back to the client.

Even though many real world problems can be solved by using this approach, the actual performance heavily depends on the scheduling. It is crucial to distribute the tasks in a way such that all nodes are fully utilized. Therefore, different scheduling strategies can be applied. A common strategy is First In, First Out (FIFO). It assigns the tasks in the natural, temporal order to the first available node. Alternatively, priority queues can be used if some tasks are more

[1] `http://hadoop.apache.org/`

important by some means. However, this implies that some tasks might be processed later than expected. In order to solve this kind of unfairness, more complex scheduling strategies have been implemented. One approach is to classify the tasks and group them among certain criteria. By supplying a dedicated set of workers to every pool, it can be ensured that all tasks are treated in a fair way and the response times for smaller jobs can be reduced.

Despite from scheduling tasks, it is also important to keep the amount of communication overhead as low as possible. Even when using commodity hardware for the worker nodes, it could be the case that it takes longer to transmit the data to and from the worker than the actual computation. This issue is mainly caused by the throughput of the network connections. In other words, the network overhead defines the upper limit for the speedup. Again, there are several techniques to bypass these limitations and increase the performance. For example, some kinds of data such as plain text files can be compressed before they are transmitted. This is a tradeoff between network and CPU utilization, which increases the distribution overhead further, but can lead to more efficient resource utilization. Another technique is to take the location of the data and the expected transfer times into account when doing the scheduling. This approach is called *data locality* and is one of the biggest benefits compared to conventional high performance computing models.

Intelligent Agents

An Intelligent Agent (IA) is a concept in the field of Artificial Intelligence (AI). More concretely, it is an entity that acts autonomously and aims to achieve a certain goal. Every agent has one or more sensors where it receives information about the enclosing environment. Based on the information, the agent can decide on the action to perform in order to react on the current environment. This is done by using a rule engine, an internal model of the environment, a utility function or something similar. According to [45], there are four types of agents, namely simple reflex agents, agents that keep track of the world, goal-based agents and utility-based agents. In the end, there are actuators, which perform the actions decided in the previous step. Furthermore, the actions may have an impact on the environment and therefore the agent will have to adapt itself to the changed conditions. In Chapter 6 this computation model has been applied for evaluating the model proposed in this thesis. Figure 2.4 shows the structure of an utility-based agent. It uses sensors to gather information about the environment and actuators to interact with it. The agent selects and carries out actions based on a function that computes utilities of the outcome of each possible action in the given environment.

Genetic Algorithms

Many problems related to AI such as interaction between agents can be solved with evolutionary algorithms. A genetic algorithm is a special type of evolutionary algorithm that is related to the biological evolution of living beings. Thus such algorithms are used for simulating whether some individuals can survive in an environment with limited resources or dangers. Moreover, individuals have to evolve and try to adapt themselves to the properties of the simulated environment. Like an IA, they also have to interact with others either to build up something which

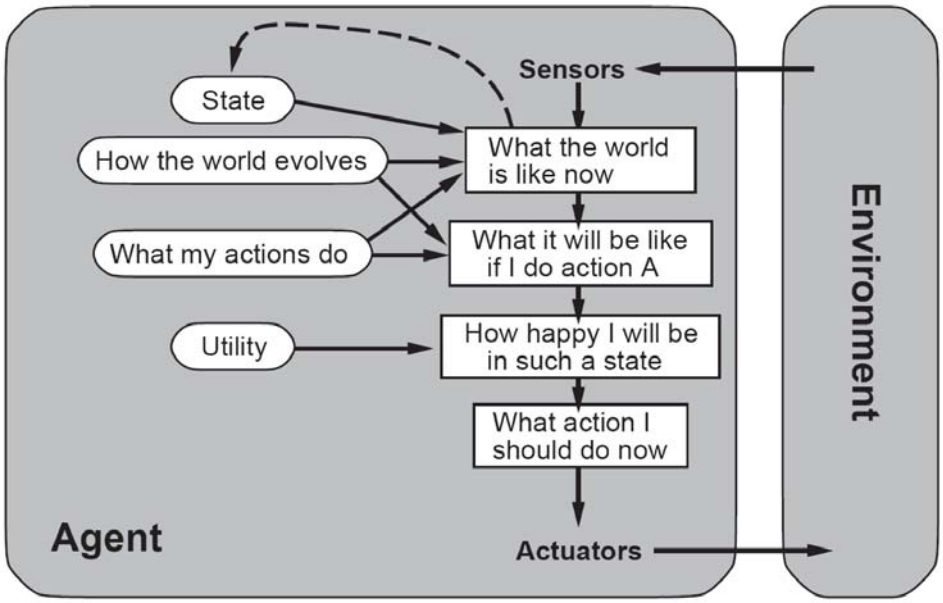

Figure 2.4: A complete utility-based agent [45]

improves their chance for survival or eliminate rivals. By applying biological mechanisms like inheritance, new generations emerge out of successful individuals while others become extinguished. This process is commonly known as *natural selection* or *survival of the fittest*.

One of the implementations for evolutionary algorithms is the Watchmaker[2] framework. It supports different models for simulating the evolution of a species. Many of them have in common that they implement an iterative procedure consisting of separate steps, which are executed one after another. In general, genetic algorithms consist of four phases:

1. Initialization / Genesis
   Before the actual evolution begins, the initial population has to be created. There are different ways for getting the candidates. For example, they can be selected from a predefined pool or they can be generated randomly.

2. Selection
   A fitness function is applied on the population, or on a subset in case the entire population cannot be processed in time. This function calculates their quality by some means, which depends on the actual problem. Note that the fitness function is equivalent to the utility function of an IA.

[2] `http://watchmaker.uncommons.org/`

3. Evolution
   The selected candidates are then evolved by applying genetic operators. There are various operations like crossover, where the elements are recombined, or mutation, which performs slight modifications on every individual element.

4. Termination
   Steps two and three are repeated until a solution, which fulfills a certain condition, is found. Again, the condition depends on the actual problem. Possible conditions can be, for example, a subset of the population, which exhibits a certain fitness, or the result is simply collected after a certain period or number of iterations.

2.6 cgroups

Cgroups allows the Linux kernel to isolate processes and limit their resource usage. Moreover, it can prioritize or even freeze them. By using this fine-grained process control mechanism, it is possible to measure resource utilization on a very technical level. The information can not only be used for scheduling, but also for billing purposes. Customers of cloud providers can rent control groups, where they can run their processes almost without interfering with other processes on the same machine. This approach is more efficient than Virtual Machines (VMs), because the virtualization layer is thinner and the processes can communicate with the underlying hardware in a simpler way. More concretely, instead of running a hypervisor and one or more guest Operating Systems (OSs) on top of the host OS, the applications run directly on the host OS with a small control layer in between. Virtualization solutions such as VMware[3] are still state of the art. Although during the last years many products, which make use of cgroups, emerged. For example, Docker[4] is one of the best known implementations with a rich ecosystem.

Google Borg [48] is a cgroups-based cluster management system for running applications across thousands of machines. In addition to features of cgroups, it incorporates other resource handling capabilities, task scheduling and real-time monitoring. This is achieved by a declarative job specification language that abstracts low-level resource management details. Agents, so called Borglets, monitor the instances and modify the parameters of the cgroups-containers accordingly. Google has evaluated Borg in detail and published results in [43]. They have also pointed out some weaknesses such as Internet Protocol (IP) address handling. The cluster manager must be able to manage port numbers like resources. Additionally, there are some resources that cannot be managed (e.g., the CPU cache or the bus connecting the memory). Processes, which exhibit a high I/O rate, can have a negative impact on other processes that have to access data from the memory with low latency. Nevertheless, these performance considerations can be neglected for many applications, because other factors such as data locality or scheduling usually have a bigger impact on performance.

[3] http://www.vmware.com/
[4] https://www.docker.com/

2.7 Google App Engine

Google offers a PaaS environment called Google App Engine[5]. Clients can deploy and run their applications on top of managed nodes. Therefore, a proprietary Software Development Kit (SDK), which contains the Application Programming Interfaces (APIs) and tools for building and deploying applications, is needed. Once an application is deployed, additional parameters can be modified by using an administration console. In particular, the client can change the infrastructure by selecting a hardware configuration out of a set of preconfigured settings. Based on that, Google App Engine dynamically provisions computational resources. For example, additional servers are started if necessary and terminated once the application load drops below a certain threshold. Therefore, it measures metrics like latency and counts the requests within a certain timeframe. Based on these values, it attempts to derive the upcoming load. In this way, it is possible to utilize the nodes and achieve efficient resource usage without manual intervention. In general, such automatic approaches cause some overhead and therefore PaaS solutions are typically not as efficient as IaaS clouds. Nevertheless, they provide a convenient way for running an application with volatile load.

In addition to dynamic resource provisioning, Google App Engine supports various services that can be accessed by an application. An important component for many applications is a persistent storage where data can be stored to in order to be accessed concurrently by other nodes or at any time in the future. In general, data storages can be classified in many ways. For example, common Relational Databases (RDBs) can be used as storages. The cloud provider handles the database management so that clients do not have to deal with administrative tasks such as software patches. Another benefit for application developers is that relational databases obey commonly known and standardized interfaces such as Java Database Connectivity (JDBC) for executing Structured Query Language (SQL) queries. Thus storage and retrieval of data happens transparently to the application and independently of the actual database.

Some cloud providers also offer file-base storages, which are used for scenarios where large amounts of data have to be replicated to other nodes or archived until needed. File-based storages provide a limited set of access operations. For example, they do not support fine-grained transaction management as some others do. However, they are ideally suited for use cases where large amounts of data have to be read by different nodes, which process the data and produce a result mostly independent of each other. This approach is used for the video rendering scenario introduced in Chapter 6.

[5] `https://appengine.google.com/`

2.8 Cloud Performance Benchmarking

PerfKit Benchmarker[6] is a tool suite for comparing the performance of cloud platforms. Predefined benchmarks can be executed on a cloud in order to get deeper information about its performance characteristics by monitoring the entire application life cycle including startup and shutdown of the nodes. More concretely, it supports measurement of common metrics including latency, network throughput and resource provisioning. PerfKit comes with a graphical performance analysis tool for interpreting the results. It runs on Google App Engine, Amazon Web Services[7] and Microsoft Azure[8]. Thus it allows comparing different aspects of cloud platforms. In [20] processing, memory and disk performance as well as network transfer rates of Google Compute Engine[9] and Amazon Web Services are compared. The authors give an overview about benchmarking solutions including PerfKit Benchmarker and measured performance metrics of various server configurations. According to their observations, there are three to five different machine types among the set of analyzed offerings. Machines belonging to the same type have similar performance characteristics. They can be identified quite easily so that developers are able to select the right type of machine for their purposes.

2.9 NoSQL Databases

During the last years, Not-only SQL (NoSQL) databases have emerged. Unlike RDBs, they do not rely on the relational approach. Instead, they use other mechanisms for storing and retrieving data. Although some implementations support a subset of SQL or provide an SQL-like query language. NoSQL databases can be categorized by their storage technology as follows:

1. Graph database: a storage where relationships between entities can be handled efficiently.
2. Column-oriented database: a column is a tuple consisting of a unique identifier, a value and a timestamp.
3. Key-value store: every entry is a tuple consisting of a unique key and the associated value.
4. Document-oriented database: typically used for storing semi-structured information such as Extensible Markup Language (XML) or JavaScript Object Notation (JSON).

Among the NoSQL database types listed above, there are other approaches like object databases. However, many NoSQL databases used for cloud computing belong to the categories mentioned above.

[6] https://github.com/GoogleCloudPlatform/PerfKitBenchmarker/
[7] http://aws.amazon.com/
[8] http://azure.microsoft.com/
[9] https://cloud.google.com/compute/

Graph Databases

A graph database is a database that makes use of graph theory in order to fetch related data efficiently. More concretely, instead of storing data in tables and logically link them together via foreign key relationships, the data is modeled as a graph consisting of nodes and edges. Nodes represent entities such as people or companies whereas edges describe their relationship (e.g., ”knows” or ”works for”). Nodes and edges can have additional attributes called properties where the actual data is stored. This storage format enables users to query for entities using relationship information in addition to conventional entity properties. For example, it is possible to look for a person that has a third-degree connection with someone who works at a certain company without knowing any personal facts about the people. When executing similar queries on RDBs, they often have to load lots of data before a few rows, which match the given criteria, can be fetched. In comparison, a graph database starts at one or more nodes and explores the graph by traversing adjacent nodes in order to find the result set. Thus loading a small block of data, which contains the next node, into main memory is significantly faster than loading entire database tables.

Column-oriented Database

Unlike traditional RDBs, column-oriented databases store records in columns instead of rows. In other words, a table is stored as a sequence of columns rather than a sequence of rows. Due to the transposed data structure, some operations can be done faster than on an equivalent row-based storage. Especially the aggregation of values of a single column is much more efficient because column loading is supported natively. On a physical level the query engine looks for a particular column on the hard disk and reads the entire column sequentially. With increasing popularity and lower prices of Solid-State Drives (SSDs), the performance penalty caused by random disk access becomes irrelevant. Thus by using SSDs, the decision for a row-based or a column-based data storage becomes less important. Although when using rotating Hard Disk Drives (HDDs), column-oriented database will perform better for many Online Analytical Processing (OLAP) operations.

Key-Value Store

Key-Value stores are databases that consist of associative arrays rather than tables. That means a record is either a scalar value such as a number or a string, or a collection of those types, identified by a unique key. During the last years, they have become quite popular along with large-scale distributed systems. They allow to store records, which contain sparse data, more efficiently than RDBs. Thus, the data needs less space on a hard disk and therefore less data has to be loaded into main memory. This circumstance can lead to better performance when aggregating values. Since key-value stores do not support relationships like RDBs do, they allow easier replication in cluster environments.

Document-oriented Database

Document-oriented databases are key-value stores that work on semi-structured objects with several attributes containing large amounts of data. Document-oriented Database Management Systems (DBMSs) often store XML or JSON files, but also allow binary data like images. Typically, they do not provide several types of locking and isolation levels like RDBs do. Instead, they focus on fast retrieval of large amounts of coherent data. Every document is identified by a globally unique key. In order to locate documents fast, the database maintains an index on the keys. Many implementations also offer query languages such as XQuery for extracting selected parts of data. Besides that, Representational State Transfer (REST) APIs are quite common, because most document queries can be expressed naturally as REST Uniform Resource Locators (URLs). Another huge benefit of documented-oriented databases compared to RDBs is their flexibility regarding the structure of the data. There are no constraints like a schema definition, which describes the structure of every record. Thus, it is possible to extend a specific document by adding attributes as well as removing unnecessary attributes at any time without the necessity of modifying other documents.

CHAPTER 3

Related Work

"If I have seen further it is only by standing on the shoulders of Giants.

— Isaac Newton

Since cloud computing does not only have technical challenges, but is also a crucial optimization problem from an economic point of view, many papers in this area introduce cost models for describing the monetary aspect and help identifying cost drivers. This includes the costs for computational resources, storage as well as network transfer.

3.1 Resource Allocation

There are several works that focus on optimal resource allocation like [46] and [4]. The first one maximizes the financial profit on a per-user base with a fixed budget and a limited amount of cloud resources. The authors clearly focus on economics and revenue, while [4] introduces a model-driven resource allocation framework. The scheduling of data-intensive tasks relies on both time and cost restrictions. Thus this approach provides great accuracy for some types of applications. However, both approaches assume that the applications are scalable in a deterministic way. In other words, they solely focus on optimizing application runtime. In contrast, this thesis deals with the question whether an application is sufficiently scalable and under which circumstances a distributed execution of an application is more advantageous than a non-distributed execution. So the outcome can be treated as relative costs, which can be mapped to absolute costs, but such considerations are not within the scope of this thesis.

3.2 Scheduling

In [41] the impact of Quality of Service (QoS) restrictions on job scheduling strategies is discussed. For the purpose of finishing jobs according to the users' needs, the tradeoff between costs and time is discussed. In order to come up with an efficient strategy, n individual tasks are assigned to m resources. Additionally, a directed acyclic graph is built for describing the dependencies of tasks. This graph defines the initial order of the tasks and restrictions on the actual scheduling. The authors calculate the processing costs of the task execution in a time-based manner. Next, it is checked whether the tasks are executed in time. If one is slower than expected, it is put into a queue for tasks that should be executed as fast as possible and if one is faster than necessary, it is put into a queue that has lower priority. This way the tasks are classified by their priority and form a sequence for processing. Depending on the nature of the job and the environment, it might be necessary to reschedule the tasks regularly. This job scheduling strategy attempts to optimize for both costs and execution time. As described by the authors, the algorithm does neither take communication overhead nor costs for waiting time into account. As discussed in Chapter 4, the overhead for scheduling and communication can have a big impact on the scheduling strategy and the performance in general. The model introduced in the very same chapter considers those factors for calculating the speedup of an application when running in a cloud environment. However, this thesis does not take QoS requirements or SLAs into account, but instead it focuses on performance and reliable predictions.

When running several tasks on a single host concurrently, it is crucial to control the amount of resources claimed by every task. Especially, this is true for cumulative resources like memory. The utilization of cumulative resources is the sum of the resource utilization of every task. The value can be predicted rather easily, but there is a hard limit (e.g., the amount of memory of a host). Nevertheless, scheduling cumulative resources is simpler than competitive resources such as CPUs. Those resources are divided among the tasks executed concurrently and underprovisioning has a negative impact on the performance of all tasks. In [49] an approach for scheduling tasks of a producer-consumer application is introduced. The authors use a distributed profiler for gathering runtime information of the application. The resource profile is later used by another component for task scheduling. If the system maintains a long and precise task schedule plan, it can react on fluctuating load in time. On the one hand, it can start new hosts and distribute the overall load by running tasks there. Alternatively, it can suspend some tasks to handle resource peaks and prevent the application from crashing. On the other hand, it can free some resources (e.g., shut down hosts with low utilization and reduce costs). In Chapter 5 a similar profiler is introduced, which captures runtime information actively, but distinguishes between workload and overhead. The information is also used to predict the load profile of the application.

Another way to categorize metrics is described in [32]. The authors present a model-driven approach called Cloud Metric Classification, which includes four models, namely application-based, measurement-based, implementation-based and nature-based categories. The application-based model defines whether a metric can be measured for any application or applies to the capabilities of a specific application. In general, some metrics can be measured directly, but some can only be derived based on various factors. This circumstance is handled by the measurement-based model. The implementation-based model defines how a metric can be measured. Last but not least, metrics can be distinguished by their nature i.e., whether they describe quantitative or qualitative properties. The benefits of this approach have been shown by implementing an agent-based monitoring solution that is capable of collecting several application level metrics in a generic way.

SPEEDL [51] is a declarative Domain Specific Language (DSL) [47] for defining scaling behavior of cloud applications. Instead of governing the application based on pro-active monitoring, it is based on a rule engine and makes use of Complex Event Processing (CEP) [29]. A rule consists of an event definition in form of a sequence of events, guarding conditions and actions. Whenever it receives a sequence of events, which matches a rule, the guarding conditions are evaluated and in case they are fulfilled, the actions are performed. For example, whenever the application has to handle 50 requests within one second, a new computation node is started unless there are already ten instances running.

An overview of various common scheduling algorithms can be found in [12]. The traditional scheduling algorithms Min-Min and Max-Min as well as Genetic Algorithm are discussed in [21]. The authors propose to combine the latter one with Min-Min and Max-Min. They show the positive impact on execution time by using those algorithms for task assignment. Moreover, the authors argue that it is possible to optimize for other factors (e.g., execution costs) by modifying the fitness function accordingly.

3.3 Middleware

A common approach for executing arbitrary tasks is the usage of a middleware, which monitors the state of the computation nodes and either provides hints for scheduling or does it transparently on its own as described in [3] and [18]. These solutions process workload changes in realtime or near-realtime in order to provide a good QoS for all users. Thus they can be considered as state-of-the-art approaches for efficient task management.

For the evaluation of the model introduced in this thesis, a middleware is used for task execution, namely JCloudScale[1]. It is a Java-based framework for building transparently scaling cloud applications. Therefore, it is ideally suited for predicting the performance of distributed execution without having an application that is specifically developed for it. Additionally, a profiler is attached to it in order to measure the overhead and collect some performance metrics. Further information about this approach can be found in Chapter 4 and Chapter 5.

[1] `https://github.com/xLeitix/jcloudscale/`

3.4 Profiling

As described in [6], when profiling Java applications, it is important to measure not only the time a Java Virtual Machine (JVM) spends on executing byte code, but also the time for executing native code (e.g., code written in C and provided as an external library). The authors use their tool for predicting the performance of arbitrary applications. Its main purpose is to allow fine-tuning of a single JVM, but it does not provide a holistic view on the performance of a distributed application. Even though it is not necessary to distinguish between Java byte code and native code, the profiler introduced in this thesis has to monitor the application by capturing per-thread statistics and classify the utilization. In other words, it has to determine the amount of time a thread has spent on processing the actual task, doing remote calls or waiting for data sent by another computation node.

A framework for execution time estimation of logic programs is proposed in [35]. It enriches static cost analysis with one-time profiling. Based on that, a cost model, which is used for deriving application-specific cost functions, can be created. The actual cost function of an application is parametrized by platform-specific constants collected in the previous steps. By using this approach, it is possible to estimate the execution time of logic programs without profiling them separately. The approach discussed in Chapter 4 is similar to the one of [35]. Profiling information is used to increase the accuracy of performance predictions of an application. However, instead of estimating execution time of logic programs, this thesis focuses on applications that obey a distributed programming model such as MapReduce.

3.5 MapReduce and Genetic Algorithm

In [22] a parallel evolutionary algorithm for cloud environments is introduced. The authors propose to use MapReduce for processing large amounts of genetic information in parallel. By using this parallel computation model, they achieve up to 9.1 times performance speedup under certain circumstances. Similar to this thesis, the authors refer to Amdahl's Law for calculating the speedup. Moreover, they focus on the design of the algorithm and evaluate the performance among different datasets. The authors conclude that MapReduce and similar computation models are ideally suited for parallel evolutionary algorithms, which iteratively improve candidate solutions.

In [28] MapReduce is compared to data-intensive flow computing. Different types of genetic algorithms are used for comparing those two approaches. Even though they are related to each other, they have different characteristics. Experiments show that Hadoop provides good performance in general, but performance suffers when running out of resources. In contrast to Hadoop, the dataflow execution environment Meandre[2] has a linear speedup. It's performance is more predictable under high load than the performance of the Hadoop-based implementation.

[2] `http://www.seasr.org/meandre/`

One of the scenarios introduced in Chapter 6 uses a genetic algorithm. Thus the simulation of the evaluation of a bacteria population uses a MapReduce-like computation model. We also came to the conclusion that this approach fits naturally to this type of problems. However, we also encountered different runtime behavior partly caused by parametrization.

3.6 Costs

When talking about cloud computing, costs are one of the driving factors in one or the other form. However, there are huge differences in definition, calculation as well as measurement. For example, as shown by [26] resource usage in terms of machines and energy costs may correlate, but they may also vary significantly. The authors propose a dynamic round-robin algorithm that is able to detect overprovisioning of resources and schedule tasks in a way that enables the system to shut down physical hosts. It makes use of code migration by moving VMs between hosts in order to speed up resource deallocation. By running multiple test cases using the standard algorithms of Eucalyptus[3], the authors show that their algorithm is way more efficient than the built-in round robin and greedy algorithm.

Moreover, it is crucial to select the right task scheduling algorithm for a particular problem as described in [39]. Besides genetic algorithms [27] mentioned previously, various common algorithms including priority based job scheduling [10] are compared and categorized in terms of cost and time. For this thesis, the approaches used for the applications introduced in Chapter 6 have been chosen in a way that reflects the nature of the problem definition. That means, for the video rendering application (dynamic) round robin is used for utilizing the hosts as good as possible. For the simulation of genetic evolution a genetic algorithm fits best.

The estimation of resource costs of data-intensive workloads is not a trivial task. The model introduced in [37] considers all the different resources enumerated before. Since it focuses on query processing in distributed databases, it also takes distributed transactions into account. Furthermore, it supports SLAs in order to be able to calculate costs using IaaS clouds. Instead of having a model-based approach for mapping resource costs, [17] predicts the cost amortization for data structures, like materialized views, in distributed databases.
Even if those works address a different research area i.e., databases, the underlying idea is similar to the one of this thesis: measuring resource usage for certain tasks and mapping them to a model for predicting cost amortization i.e., when does cloud execution outweigh the additional costs for scheduling, remote method invocations, etc.

[3] `https://www.eucalyptus.com/`

In [2] the authors propose an approach for calculating costs as decision base for job assignment. They convert the costs of resources like CPU and memory into an artificial cost value. In order to obtain a semi-optimal job assignment, the tasks are assigned in a way that minimizes the overall costs. The authors do a comprehensive analysis using Parallel Virtual Machine (PVM)[4] and Mosix[5], compare the results and conclude that the opportunity cost approach leads to good results compared to naive online heuristics.

The problems of moving applications to a cloud environment are discussed in [33]. More concretely, the authors introduce a methodology for doing trade-off analysis taking various costs into account. The costs can be divided into four categories, namely complexity, performance, resource and energy. Complexity includes all migration efforts such as environment setup, manual reconfiguration or even reprogramming. The performance aspect includes performance metrics and penalties caused by virtualization. Another crucial cost driver is resource utilization, where it is important to optimize for resource utilization and keep overprovisioning as low as possible at the same time. Last but not least, power consumption can be measured for every physical machine, which results in operations costs that grow with the size of the cloud. Based on the results of the analysis, it can be decided which infrastructure setup provides the best cost-benefit ratio.

This thesis solely focuses on technical aspects such as scalability. In other words, the proposed model gives a good indication whether it makes sense to run an application in a distributed way or if there are any limitations. However, when deciding whether a cloud-based approach shall be used for a concrete application, additional factors like setup and operations costs have to be taken into account as well.

3.7 Prediction

The major goal of [24] is similar to the one of this thesis. The authors claim that CloudProphet can predict the performance of a particular application without running it on the target cloud platform. Instead of attempting to categorize the application by its performance characteristics, they use trace-and-replay [36]. This approach allows the generation of reproducible system load for testing purposes (e.g., performance optimizations). First, the tracer records the behavior of an application. This can be done either directly by observing the operations performed by the application to trace, or by monitoring the state of the machine. Second, the replayer simulates the previously recorded states by re-submitting network packages, allocating memory, causing load on the CPUs, etc. In [25] it is shown that the error rate of CloudProphet was below 20% in almost every case. However, the authors state that the applications chosen for evaluation behave quite deterministically. By using trace-and-replay on nondeterministic applications, the accuracy of the predictions is likely to drop. They also postpone memory-intensive applications as future work.

[4] http://www.csm.ornl.gov/pvm/
[5] http://www.mosix.cs.huji.ac.il/index.html

The model introduced in Chapter 4 supports fluctuating cloud overhead, but cannot be applied without further knowledge about the application and the system it is running on. In contrast to that, CloudProphet allows predicting the performance without running the actual application.

Log2cloud [40] is invented for predicting the costs of VMs based on log files. By extracting traces and categorizing them, it is possible to build resource profiles that describe the utilization over time. In the first step of the analysis these profiles are translated to Markov decision processes. In the second step model checking techniques as explained in [15] are applied on them, which results in a cost prediction. On the one hand, this approach does not influence the performance of the application like a profiler, so the measured values are very accurate. On the other hand, log analysis does not lead to detailed information on a low and technical level. Thus it is suited for determining the minimum amount of VMs required for avoiding SLA violations based on historic information. The ALASCA profiler introduced in Section 5.2 also writes log files, which contain information on source code level. Like any profiler, it has a small, but measurable negative impact on performance. Especially if the information about the running threads is collected too frequently, there might be a considerable difference in the runtime behavior of the application.

CHAPTER 4

Conceptual Approach

A person who never made a mistake never tried anything new.

— Albert Einstein

As already mentioned in Chapter 2, Amdahl's and Gustafson's Law describe the scaling of applications on multi-core machines. We argue that the general principles can also be applied to cloud computing. However, as both do not consider overhead as a crucial dimension, they do not provide the required accuracy for such estimations. Therefore, we show the impact of cloud overhead and define *cloud speedup* and *scaled cloud speedup* based on Amdahl's and Gustafson's Law, respectively.

4.1 Model

Cloud Overhead

When we are talking about overhead, we are referring to the cloud overhead we have to accept when running a distributed application. It consists of all the additional effort that would not occur if the application would be running on a single physical machine. To be more specific, we focus on the scaling of machines, remote procedure calls, data transmission and network latency/bandwidth. Depending on the application, some of them can not even be noticed. For example, when running a computation intensive task, which lasts for hours and delivers a single numeric value, the network bandwidth can be neglected. In general, they are measurable and have to be considered when predicting the performance of an application.

In the following, we propose a model that supports all those dimensions. However, any other parameter can be added easily if the following conditions hold:

- The value can be measured quantitatively without having additional impact on the application. In other words, the measurement must neither have an impact on the application itself nor on other measurements (e.g., additional runtime overhead by monitoring has to be considered carefully).

- The measured values can be mapped to compatible units of measurement, if necessary. Typically, most of the measurements will deliver temporal values. In this case, the count of failures or similar information can also be used as long as a temporal value can be associated with them.

- The parameter can be measured deterministically, since indeterminism will lead to inaccurate results or even could make performance predictions impossible.

In order to make reliable performance predictions, it is important to have a deep understanding of the measured values. Therefore, the cloud overhead is classified into two categories: static and dynamic overhead.

Static Overhead

Static overhead subsumes the fixed overhead as well as the one that depends on several static parameters that can be evaluated before runtime.

Every additional effort that does neither scale with the execution environment (e.g., the host count), nor with the problem size (e.g., the amount of data), can be considered as fixed. Despite the fact that most types of overhead do not fit into this category, they can have considerable impact on the execution time. For example, if the computation nodes are started on-demand, the scaling overhead is fixed. Every time a task is started, it takes a certain amount of time to start the required instances. Note that the simultaneous startup of multiple machines increases the overall duration by the startup time of one machine, while the costs are increased by the costs for the sum of the startup times.

Furthermore, cloud overhead, which depends on one or more factors that can be computed statically, is static as well. For example, for every computation node that is monitored, additional effort is introduced, which would not be necessary in case of non-distributed execution. In this case, it increases with the number of computation nodes used.

Dynamic Overhead

Dynamic overhead depends on the execution of a particular application. In other words, it is specific for the application or a type of application. Compared to static overhead, it depends on runtime parameters and therefore varies. This includes all effort that cannot be determined statically (e.g., the amount of messages passed between computation nodes over time). Therefore, dynamic overhead is harder to predict than static overhead.

Effective Overhead

The effective overhead o is equal to the sum of the static and the dynamic overhead:

$$o = o_{static} + o_{dynamic}$$

4.2 Amdahl's Law Revisited

Originally, performance properties were observed for some programs running on CPUs supporting different features (e.g., instruction pipelining or vector operations), but Amdahl's Law can be applied to cloud computing as well. Instead of heaving p being the fraction of parallel instructions, it denotes the fraction of the overall computation that can be executed in parallel. Additionally, N is not a physical processing unit (i.e., a CPU), but a computation node with one or more virtual or physical CPUs.

***Proof**: $t_t = t_s$ for large N.*
As we already concluded, for any fixed size problem and a possibly infinite amount of computation resources N, the total execution time t_t is equal to the execution time of the part that cannot be parallelized t_s.

$$\lim_{N\to\infty} t_t = \lim_{N\to\infty} ((t_t \cdot s) + (t_t \cdot \frac{p}{N})) = t_t \cdot s = t_s$$

This fact seems to be trivial, however, it has a crucial impact on the scalability of applications. Therefore, one would like to determine p for any type of application in order to get an indication what the maximum speedup with a given amount of computation resources can be. Since distributed execution can entail a large overhead, it has a significant impact on the part executed in parallel. Without loss of generality, the overhead changes the speedup linearly like the fraction of the code that is not executed in parallel. Thus, we extend the formula by introducing the cloud overhead o and obtain the following modified formula for the cloud speedup:

$$cloud\ speedup = \frac{s+p}{s+\frac{p}{N}\mathbf{+\,o}} = \frac{1}{s+\frac{p}{N}\mathbf{+\,o}}$$

By considering the cloud overhead like that, the performance prediction is much more accurate. Although knowing the upper bound might not be sufficient if there is a natural boundary. For example, in Chapter 6 we have shown that an exemplary application does indeed scale as expected, but at soon as a certain amount of network traffic is reached, adding additional instances does not pay off. While performance might still increase slightly, the network builds a bottleneck and therefore decreases efficiency by not being able to provide the required data and fully utilize the machines.

4.3 Gustafson's Law Revisited

Similar to Amdahl's Law, cloud overhead has impact on the parallel part in Gustafson's Law as well. Parallelization enables the application to process a larger problem (e.g., more tasks) within the same time frame. Thus, additional effort caused by distribution reduces the amount of time that can be spent on running tasks in parallel.

$$scaled\ cloud\ speedup = \frac{s + (p \text{ - } \mathbf{o}) \cdot N}{s + p} = s + (p \text{ - } \mathbf{o}) \cdot N$$

4.4 Example

Let us assume that 90% of an application can be parallelized among three nodes. The tasks can be scheduled simultaneously, which means that they can be started at the same point in time. Furthermore, we assume that the static cloud overhead o_{static} is 10% for startup time, etc. Therefore, the cloud speedup can be calculated as follows:

$$\begin{aligned} cloud\ speedup &= \frac{1}{s + \frac{p}{N} \mathbf{+ o}} \\ &= \frac{0.1 + 0.9}{0.1 + \frac{0.9}{3} + 0.1} = 2 \end{aligned}$$

By running the application on three nodes, the result can be computed in half the time.

Alternatively, the problem size can be increased while the execution time stays the same. By applying the modified Gustafson's Law, we obtain the following result:

$$\begin{aligned} scaled\ cloud\ speedup &= s + (p \text{ - } \mathbf{o}) \cdot N \\ &= 0.1 + (0.9 - 0.1) \cdot 3 = 2.5 \end{aligned}$$

By running the application on three nodes, 2.5 times as many tasks can be executed within the same time frame.

Figure 4.1 outlines the different approaches. $A1$ to $A3$ and $G1$ to $G3$ are the Cloud Hosts (CHs) performing the tasks according to Amdahl's and Gustafson's approach whereas $Serial$ is a computer doing non-distributed execution. White boxes designate cloud overhead, dotted boxes represent the tasks that can be processed in parallel, black boxes indicate the amount of code that cannot be parallelized and the striped boxes show the additional tasks that can be processed when using the cloud resources entirely. Similar to the definition of Amdahl's Law, it can be seen that the execution time is reduced to a minimum by making full use of the available resources. However, starting three machines takes some time, which would not be necessary

in the non-distributed approach. The total overhead here is approximately 23%. The same observations are true for the application of Gustafson's Law to cloud computing. While the cost of running three CHs for 90% of the time is significantly higher than in the other scenarios, 2.5 times as many tasks can be executed and the amount of overhead is almost 11%.

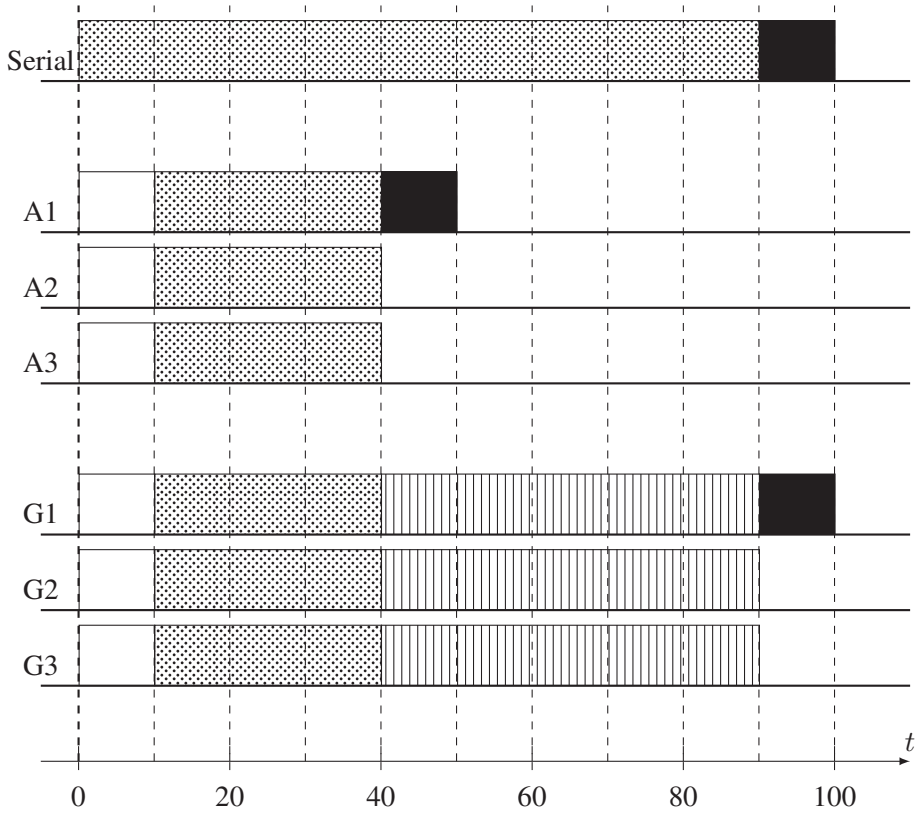

Figure 4.1: Comparison of serial execution, Amdahl's and Gustafson's approach

CHAPTER 5

Implementation

The final test of a theory is its capacity to solve the problems which originated it.

— George Dantzig

5.1 Middleware

A middleware is a software that connects different software components or applications with each other. Typically, middleware systems fulfill various purposes like monitoring or forwarding and translating of messages. As already mentioned in Chapter 3, we used JCloudScale[1] as middleware solution for running exemplary applications in order to evaluate the approach we propose in this thesis. Its main purpose is to enable ordinary Java-based applications to be run in an IaaS cloud environment without considerable changes in code or software architecture [23] [50].

Code Mobility

One of the main concepts of JCloudScale is commonly known as *code mobility* [7]. It is an umbrella term for migrating code or application state between machines. Its purpose is to support operations that cannot be performed on the target machine due to various reasons (e.g., hardware limits), by performing them on another machine and eventually getting the result once the computation is finished. Over the past 15 years more and more types of applications have emerged, where code mobility is used in the one or other form. This includes thin-clients, mobile devices such as smartphones, and last but not least cloud computing.

[1] `https://github.com/xLeitix/jcloudscale/`

One important classification is the distinction between weak and strong code mobility. While the weak variant includes the migration of code, data and all dependencies, the strong variant also includes application state such as the intermediate result of a computation. Various types of approaches and technologies like Remote Method Invocation (RMI) support different aspects of code mobility. For many use cases weak code mobility is sufficient. There is no general solution for supporting strong code mobility for any type of application. This does not imply that it is not supported for some applications. In other words, it heavily depends on the applications nature whether its state can be migrated or not. More precisely, the application must obey the following properties:

- be interruptible at certain points in time
- state, which describes the "progress of the computation", can be observed and collected
- the state is serializable (can be transmitted to another location e.g., another server)
- given the code and a state, the application can continue and produce the same result as if it had not been interrupted

These properties roughly describe what is known as continuation [44]. It can be seen as a snapshot of the current program execution including the program counter, the call stack and variable assignments. Continuations have been described by Adriaan van Wijngaarden in 1964 and since then many programming languages have been adapted to support them as a first-class language construct. For example, they can be used for backtracking (Prolog) or coroutines (Perl). Other languages like Java have no built-in support, however, there are some libraries like Apache Commons Javaflow[2], which enable applications to make use of continuations to some extent. Regardless of the implementation, neither JCloudScale nor any other solution will be able to support code mobility if one or more of the above listed properties are not fulfilled. Moreover, there are some use cases, where it is not possible to interrupt the computation or it is simply impractical to transmit the state due to its size or dependencies.

Code Deployment

JCloudScale supports *code on demand*, which enables CHs to download code from a remote machine, execute it locally and interact transparently with other machines. Therefore, the cloud manager component has to ensure that the code is available on the CHs. Furthermore, all runtime dependencies like files to process or third-party libraries have to be deployed as well. Whenever a CH performs a request, it has to resolve the Cloud Object (CO) containing the code to execute. A CO is an ordinary Java object with methods exposed to other machines. In case it has not been constructed yet, a transitive closure over the dependencies of the CO has to be computed. However, the JVM can load more code later on (on demand) so building a Directed Acyclic Graph (DAG) of the CO and its statically known dependencies might not be sufficient. A special classloader is used for retrieving additional classes from a remote machine whenever they are required. In the worst case, the entire code base, which is also known as classpath, has to

[2] `http://commons.apache.org/sandbox/commons-javaflow/`

be transmitted to the CHs. In the near future, Project Jigsaw[3] is going to enhance Java with built-in module support. It enables developers to modularize code and express dependencies in a declarative way. This will enable a standardized way for deploying Java code across many machines. As of today, frameworks like JCloudScale have to implement remote classloading on their own. Additionally, any resources like files, which will be accessed, have to be available remotely as well. Depending on the type of resource, there may be different solutions for making them available on the remote machine. For example, files can be transmitted along with the code onto the remote machine. Alternatively, file access can be intercepted and redirected to a Network-Attached Storage (NAS) that is available at the CHs, or a distributed file system such as Network File System (NFS) is used.

Scaling

The main purpose of JCloudScale is to simplify development and deployment of ordinary Java applications that should benefit from distributed execution. Ideally, an application needs no modifications in order to utilize the potentially infinite amount of processing power provided by an IaaS cloud. More precisely, JCloudScale enables an application to scale transparently by making use of AOP [19]. JCloudScale provides all the code for deployment and remote communication via Java Message Service (JMS) and weaves it into the application. The byte code is modified at load-time so that the code, which should be executed in a distributed way, is replaced by proxies. At runtime, whenever such a proxy is accessed (i.e., one of its methods is invoked), JCloudScale ensures that the code intended to be executed is executed on a remote CH instead. Thus the byte code and all dependencies required for the execution are transmitted to the CH and instances of the desired COs are created if necessary. This process is done entirely transparently to the caller and all cloud-related aspects like remote communication and scaling are done by the middleware. In particular, it scales up and down CHs by starting and terminating virtual machines, and deploying the application onto them.

Basically two artifacts have to be provided in order to make an application scalable with JCloudScale: the cloud configuration and the scaling policy. The cloud configuration contains all the information required for operating the cloud platform (e.g., the IP address where the middleware can be accessed at), or credentials for a public IaaS cloud like Amazon Elastic Compute Cloud (EC2)[4]. The scaling policy handles the execution of COs by distributing them among the CHs. At any point in time, the scaling policy can trigger the creation of new CH instances in order to provide enough computation nodes to handle the execution. It can also scale down the cloud by terminating CHs and therefore improve utilization and reduce costs. JCloudScale autonomously collects metrics of the CHs such as CPU and memory utilization as well as network I/O. Based on these collected values, a custom scaling policy can be implemented, which fits the requirements of an application that does not have the same scaling characteristic as any of the provided scaling policies. Research has shown that by using an appropriate scaling policy, an application using JCloudScale can perform almost as good as the same application extended with application-specific code for distributed computing and deployed manually [50].

[3] `http://openjdk.java.net/projects/jigsaw/`
[4] `http://aws.amazon.com/ec2/`

Usage of JCloudScale

As described before, JCloudScale has to be parametrized with the cloud configuration and an optional custom scaling policy. Listing 5.1 shows an example of a typical JCloudScale bootstrapping sequence as it was used for the scenarios evaluated in Chapter 6. First, platform specific configuration is loaded from a file. Second, JCloudScale itself is configured by supplying the previously created platform configuration and additional information like the IP address of the JMS server to use. Moreover, a custom scaling policy is provided, which is optimized for a particular application.

```
Properties props = load("openstack.properties");
CloudPlatformConfiguration cloudPlatformConfig =
    new OpenstackCloudPlatformConfiguration(props)
    .withInstanceType("m1.medium");
JCloudScaleConfiguration config =
    new JCloudScaleConfigurationBuilder(cloudPlatformConfig)
    .with(new CustomScalingPolicy())
    .withMQServerHostname(mqServerIpAddress)
    .build();
JCloudScaleClient.setConfiguration(config);
```

Listing 5.1: Setup of JCloudScale

Next, the tasks can be processed like in an ordinary non-distributed application. Every task-based application has (at least) two classes, namely the task itself and a scheduling component, which runs the tasks. For the sake of simplicity, we call them `Scheduler` and `Task`. A `Task` contains the code that does the actual processing of the given data. Listing 5.2 shows a typical CO, which accepts a parameter of a certain type and is destroyed automatically after it is invoked by JCloudScale.

```
@CloudObject
public class Task {
    @DestructCloudObject
    public String work(@ByValueParameter Parameter p) { ... }
}
```

Listing 5.2: Implementation of a cloud object

The `Scheduler` contains code for handling the execution of tasks. For example, it runs them in a thread pool and monitors their execution state. Note that the implementation is tied to the application and may vary significantly in its details. In Listing 5.3 a set of `Task` instances is submitted to the `Scheduler`, which provides a way to retrieve the results of the tasks once they are completed.

```
Scheduler scheduler = new Scheduler();
for (Task task : tasks) {
    scheduler.schedule(task);
}
while (!scheduler.isFinished()) {
    Object result = scheduler.getNextResult();
    // Process result...
}
// Terminate JCloudScale
JCloudScaleClient.closeClient();
```

Listing 5.3: Usage of JCloudScale

Instead of executing `Task` instances within the very same JVM, JCloudScale replaces them with proxies and distributes the work among several CHs as specified by the scaling policy. Figure 5.1 shows the process of invoking a method on a remote CO. Whenever a method of a CO is invoked (e.g., `work()`), JCloudScale intercepts the invocation and requests a CH to perform the actual invocation. It also takes care of parameters and returns the result to the object, which called the method on the proxy. This process is done completely transparently to the caller like it would invoke the method on the actual object.

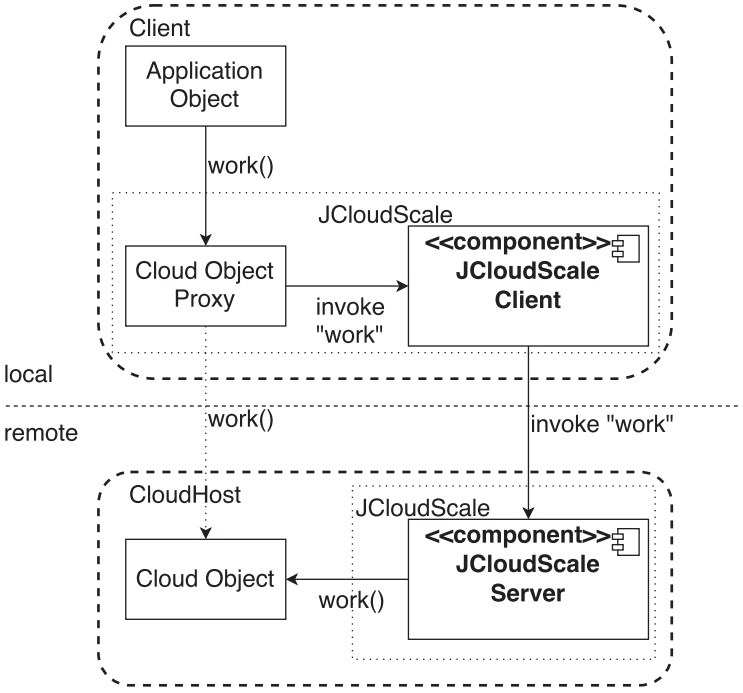

Figure 5.1: Remote method invocation with JCloudScale

5.2 Profiling Applications

In order to inspect an application and get more technical information about the application's runtime behavior, a profiler can be used. First, Java profilers can observe different high-level metrics of a JVM like the number of classes loaded or the number of threads, which are alive at the moment. Second, they track system information such as CPU and memory utilization or garbage collection. Finally, they collect low-level performance data like the frequency and the duration of method calls.
For the purpose of predicting the performance of applications by applying the model introduced in Chapter 4, a profiler was implemented in order to measure low-level performance data as well as system information.

Architecture of the ALASCA Profiler

Most Java profilers provide more features than the core functionality mentioned above (e.g., a Graphical User Interface (GUI) for convenient presentation of the data). However, those features are not necessary for profiling a distributed application and collecting runtime information. Moreover, they can be troublesome, because the monitoring entails additional overhead, which might bias the measured data. Even though Java provides a facility to collect that information resource-efficiently, a custom profiler, which fits the needs of this particular use case, provides more accurate results. Therefore, a profiler called ALASCA[5] (A Lightweight Analyzer for Scalability of Cloud Applications) has been developed. Unlike most other Java profilers, it runs inside the process, which is profiled, and therefore can access all public objects within the very same JVM with almost no overhead. Figure 5.2 shows how a distributed application can be profiled with ALASCA. In every JVM a separate profiler instance keeps track of the components to profile. The collected data is persisted in the file system and can be combined in order to get a holistic view on the application.

ALASCA uses Java Management Extensions (JMX) for retrieving the low-level information like method call duration. JMX is a standardized technology for managing different kinds of resources (e.g., applications and services). Information about those resources (e.g., Random Access Memory (RAM)) can be obtained from so called *Managed Beans* (or *MBeans*). A JMX agent exposes those MBeans for other components or applications, which use a well-defined JMX connector for querying them. Even though MBeans can be accessed remotely, ALASCA collects the information locally to avoid network latency and reduce overall system utilization. Furthermore, the NetBeans Profiler Library[6] has been used to collect CPU profiling snapshots. On the one hand, by using this standalone library it can be ensured that the performance data is collected correctly. On the other hand, the snapshots can be stored in a standardized format, which can later be processed by other monitoring and profiling tools like Java VisualVM[7]. Additionally, ALASCA emits the data in a textual format, which can be analyzed easily in order to detect bottlenecks and measure the cloud overhead.

[5] `http://alasca.googlecode.com/`
[6] `https://netbeans.org/projects/profiler/`
[7] `https://visualvm.java.net/`

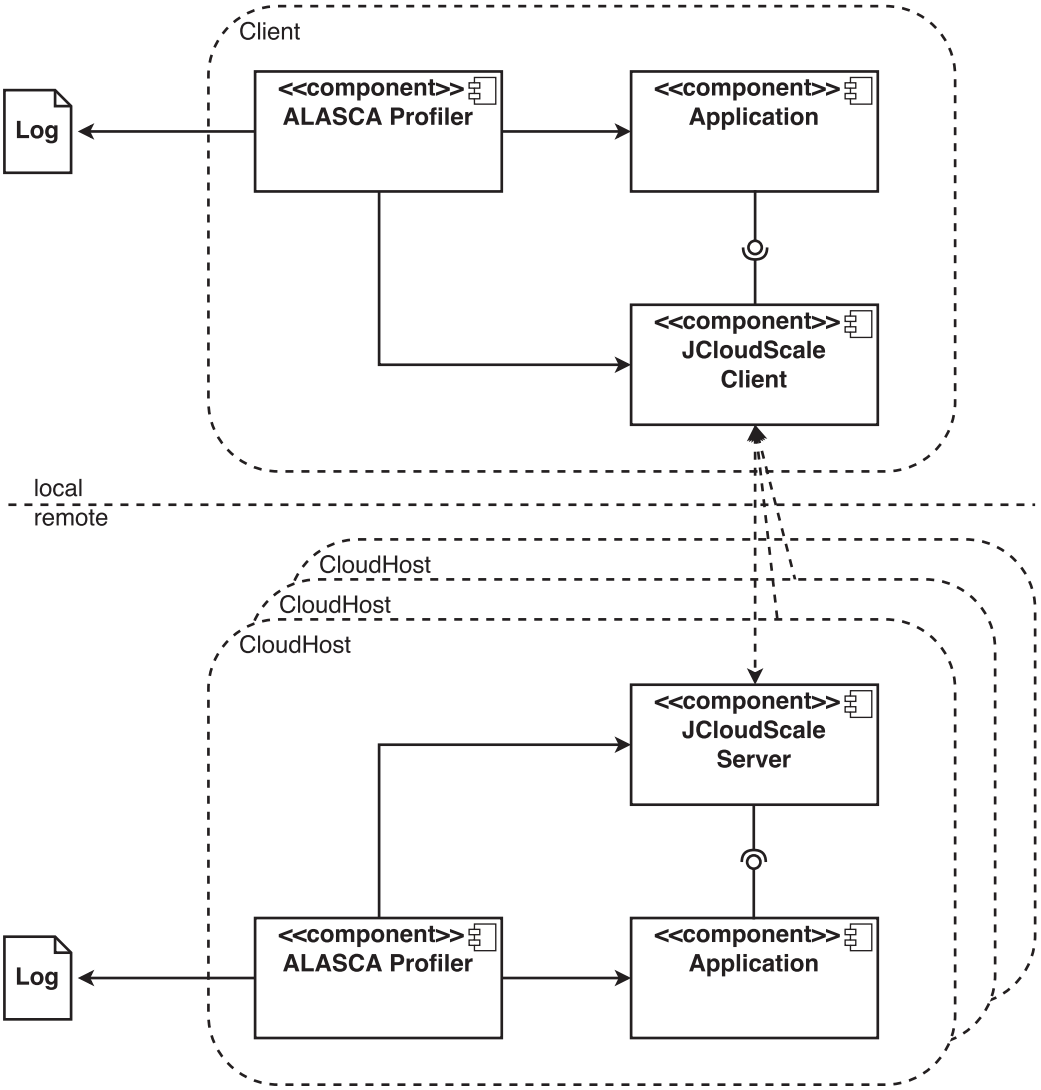

Figure 5.2: Deployment view of JCloudScale used along with ALASCA

ALASCA consists of three components: the sampler, the analyzer and the snapshot storage. The sampler uses the JMX connector for retrieving information about all threads of the JVM, which is profiled. Then the NetBeans Profiler is invoked and collects CPU profiling snapshots, which contain information about the current call stack of the threads to monitor. Next the sampler stores them in the snapshot storage until the profiling is stopped (e.g., the application terminated or the request was processed successfully). Finally, the analyzer filters the collected snapshots and aggregates them on method and class level. The preprocessed data can be used to extrapolate the execution time and derive performance characteristics. Figure 5.3 outlines the architecture of ALASCA.

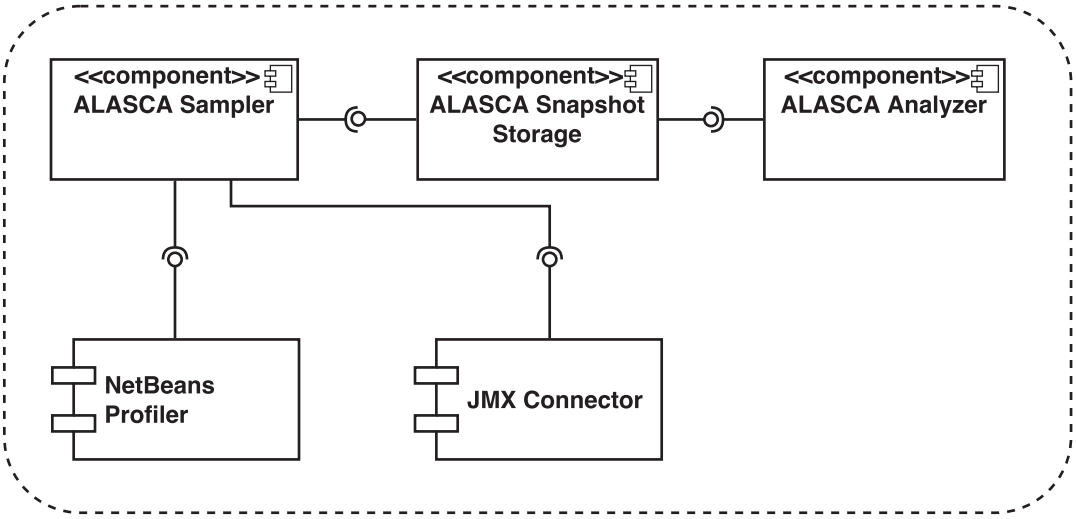

Figure 5.3: Software architecture of the ALASCA profiler

Sampler

The sampler periodically collects information about each thread of the JVM it is running in. Therefore, it uses JMX to get a list of all threads that are alive. A thread is considered to be alive if it has been started and has not yet died. The collected thread dumps contain the thread identifier, the current thread state and its stack trace. The NetBeans profiler keeps track of the last dumps and calculates metrics such as the duration the thread was active. Furthermore, by comparing stack trace elements, it counts the number of method exits and method enters. Since this is done in a fixed time interval (typically a few milliseconds), it can happen that some method invocations will not be recognized. However, this is perfectly fine since we only consider method invocations that take quite some time. The amount of information that can be obtained by reducing the sampling interval is inversely proportional to the overhead. Thus decreasing the interval may not necessarily lead to more accurate results.

Snapshot Storage

By creating a snapshot of all alive threads every few milliseconds, lots of data is collected over time. The collected data is (by default) stored on the local file system for two reasons. First, the application can be distributed among multiple CHs and therefore the collected data has to be transmitted to the node that runs the analyzer. Second, in order to run analytical queries it is more convenient or even necessary to combine the data collected on several machines. The snapshots are stored in a common binary format, which is supported by other Java profilers, or in plain text.

Analyzer

The analyzer performs statistical methods on collected snapshots. By filtering known blocking methods like `sun.misc.Unsafe.park()`, the analyzer collects only methods that consume CPU time. Runtime information about the remaining methods can be used for various appraisals. For example, the fraction of the time spent on network communication can be calculated by summing up the execution times of the methods that contain the code, which is used for communication. Moreover, it is possible to rebuild the call stacks at certain points in time by processing a sequence of snapshots. Thus the number of invocations of a particular method can be counted and inefficient code can be detected.

Usage of ALASCA

As already mentioned before, ALASCA runs within the JVM of the application that should be profiled. Thus it has to be started directly in the code right before the actual application is running. After the analysis is stopped, various statistics can be accessed programmatically and stored in a file for post-processing. Listing 5.4 shows how ALASCA can be used for recording method call stacks and printing them (excluding ActiveMQ[8] methods) to the console.

```
CpuSnapshotBuilder snapshotBuilder = new CpuSnapshotBuilder();
snapshotBuilder.withSamplingInterval(10L).start();

// Run the application...

snapshotBuilder.stop();
String[] excludeFilter = new String[] {"org.apache.activemq."};
snapshotBuilder.printCallTree(excludeFilter);
```

Listing 5.4: Usage of ALASCA profiler

Figure 5.4 shows an example of a control flow of an application using JCloudScale along with ALASCA. It illustrates the lifecycle of COs including creation of remote objects, communication via proxies and destruction. At the same time, ALASCA periodically takes snapshots of selected threads and collects performance metrics. Afterwards, it permanently stores them so that ALASCA's analyzer component or other tools can be used to gain insight into the collected data. Note that the `Scheduler` component (introduced in Figure 5.1) is part of the application and not displayed separately for the sake of simplicity. Furthermore, it might be necessary to profile the client side of the application as well in order to get a comprehensive performance profile of the application.

[8] `http://activemq.apache.org/`

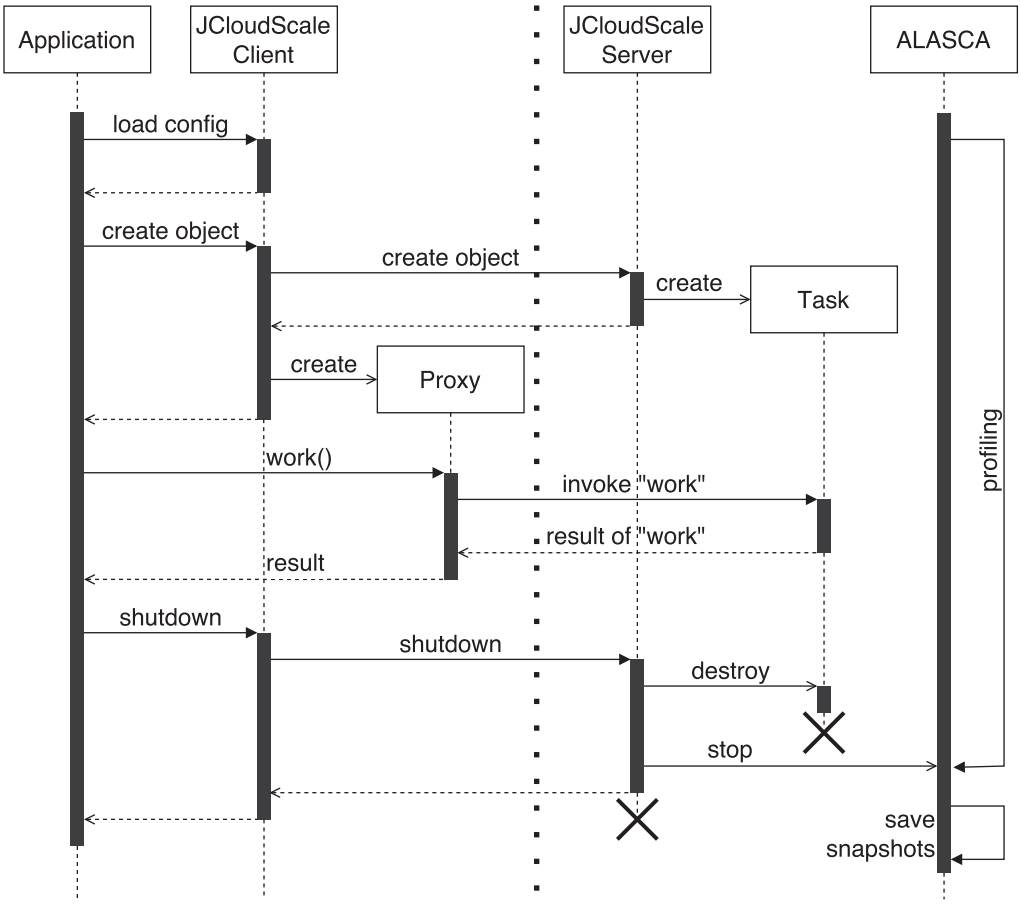

Figure 5.4: Sequence diagram of an application using JCloudScale and ALASCA

Figure A.1 and Figure A.2 show aggregated call stacks of a single thread rendered as call trees. They visualize the methods invoked by other methods and the accumulated amount of time of all method invocations within the profiling period. In Section 6.3 we compare the figures and explain the deviations in the context of the exemplary application introduced in Section 6.1.

CHAPTER 6

Evaluation

The most exciting phrase to hear in science, the one that heralds the most discoveries, is not "Eureka!" (I found it!) but "That's funny..."

— Isaac Asimov

6.1 Scenarios

In order to verify the predictions based on the model introduced in Chapter 4, different types of applications with contrary resource usage were considered. The first scenario primarily relies on I/O throughput, whereas the second one deals with tasks that are both computation and memory intensive.

Scenario 1: Video Processing

Video processing can cause high CPU load and might require a considerable amount of memory. Unlike many typical processing tasks, big amounts of data are read from and written to a data storage, which requires a high I/O rate.
In this scenario, a video file is read from a hard drive, a watermark is added to every frame and finally, the resulting video is stored on the hard drive again. Since the sequence of frames can be split into chunks, which can be processed in parallel, this kind of task naturally scales with the available processing units (i.e., CPU cores). For every frame, the picture information has to be stored in memory. Thus the amount of memory can be taken as constant with respect to the number of processing units. This leads to the conclusion that the amount of parallel tasks heavily depends on the I/O throughput. In other words, there is a point where the speed of the CPUs exceeds the transfer rate of the storage and adding additional CPUs or memory does not result in a performance improvement.

Scenario 2: Genetic Algorithm

The second scenario simulates biological evolution by using genetic algorithms. As described in Section 2.5, a genetic algorithm is a special type of evolutionary algorithm, where a population of living beings such as bacteria are evolved over many generations in a simulated environment. The fitness function is applied to every candidate and those with the highest outcome build the genetic pool for the next generation. Moreover, with a certain probability a genetic mutation happens, which can either result in an advantage or disadvantage for the organism. Therefore, an arbitrary gene is selected and modified randomly. Additionally, one-point crossover is done by selecting a random crossover point for recombining Deoxyribonucleic Acid (DNA) strands.

In this scenario, DNA strands consisting of 4.65 million nucleotides are used. This amount of nucleotides conforms to the average amount of *Escherichia coli*. The algorithm used in this scenario stores a single DNA strand consisting of Adenine (A), Cytosine (C), Guanine (G), Thymine (T) for every bacteria. Due to the encoding of 2 bit per nucleotide base, a single DNA strand requires roughly 1.11 MB of memory.

Such simulations are CPU-intensive and require large amounts of memory. The simulated environment provides resources, which can be consumed by the living beings and their behavior has an impact on the environment. Furthermore, at any time the organisms might interact with each other. For example, some individuals could dominate others.

Algorithm

For this scenario, the MapReduce programming model is used as proposed by [22] and [28]. At first, the entire population is split among the available computation nodes. Then the map function is applied, which computes the fitness of chromosomes. As mentioned in Section 2.5 every individual can be considered as an agent. Furthermore, there is an utility function, which calculates the current fitness value of chromosomes. This function is also part of the map function in the first step (i.e., it assigns a numeric value to every chromosome). The procedure is sketched in Listing 6.1. Note that the invocation of the utility function is omitted because it is part of the framework used for the evaluation.

The reduce step is done by the selection policy. It selects a subset of the individuals based on their fitness and carries out genetic crossover as well as permutations with a certain probability. At the end, the fittest chromosomes are collected from the computation nodes and the fittest one among them is determined. Thus the resulting chromosome is the fittest one of the population after a certain amount of generations. The combine step is outlined in Listing 6.2.

6.2 Environment

The scenarios were run on a private cloud consisting of 12 Dell blade servers with two Intel Xeon E5620 CPUs (2.4 GHz Quad Cores) and 32 GB memory each. The servers are connected

input : The simulation parameters including population size, number of nucleotides, elitism rate and number of generations
output: The chromosomes of all individuals of the population that survived

```
1 geneticAlgorithm ← initGeneticAlgorithm (
2           crossoverPolicy, mutationPolicy, selectionPolicy);
3 stopCondition ← createStopCondition (generations);
4 population ← genesis (speciesProperties);
5 chromosomes ← simulate (geneticAlgorithm, population, stopCondition);
6 return chromosomes;
```

Listing 6.1: Algorithm for the *map* step

input : The chromosomes of several individuals
output: The fittest chromosome of the population according to a utility function

```
1 fittest ← null;
2 foreach chromosome in chromosomes do
3 |   if fitnessOf (chromosome) > fitnessOf (fittest) then
4 |   |   fittest ← chromosome;
5 |   end
6 end
7 return fittest;
```

Listing 6.2: Algorithm for the *combine* step

via a dedicated Gigabit Ethernet. The applications were executed in generic Ubuntu 12.10[1] VM instances running on top of OpenStack[2]. In our evaluation we used one medium-sized cloud instance (two virtual CPUs, 3.7 GB of memory) for the ActiveMQ[3] communication server and up to 16 small instances (one virtual CPU and 960 MB memory each) as CHs. Although the cloud was used exclusively for the test runs, we repeated the measurements several times in order to eliminate one-time effects.

For the second scenario, which is quite memory-intensive, we also did some test runs on a large machine (16 virtual CPUs, 30 GB of memory). This instance performed better for small amounts of tasks, but we encountered similar runtime behavior on both machine types for larger bacteria populations. The effects, which lead to a performance decrease, occurred at a later point in time. Thus the predictions as well as the measurement results refer to the setup with smaller instances.

[1] http://old-releases.ubuntu.com/releases/12.10/
[2] http://www.openstack.org/
[3] http://activemq.apache.org/

6.3 Performance Predictions

Predictions Scenario 1

For this scenario the video was split into parts of 60 seconds each. First, we did some measurements with various files in order to determine the amount of data that can be processed in a given time frame. Table 6.1 lists the measured throughput. On average, 1.88 MB were processed per second.

Files	Total Size [MB]	Time [s]	Throughput [MB/s]
1	25.144	11.614	2.165
2	41.072	19.856	2.068
3	58.120	29.187	1.991
4	71.870	39.381	1.825
5	88.295	48.840	1.808
10	174.002	94.927	1.833

Table 6.1: Measured performance with increasing number of video files

For estimating the cloud overhead, a single measurement was performed with ten files using a single cloud host. The processing took 132.304 seconds, which is approximately 40% more compared to the previous measurements (listed in Table 6.1). Even though we know the amount of overhead for the task, the collected information is not sufficient for predicting the performance for a larger amount of computation nodes. Since the total processing time linearly increased with the amount of data processed (i.e., twice as much data took twice as long) and the application utilized 100% of the CPU, the host had no capacity left. In other words, the total processing time can be reduced only by acquiring more cloud hosts and performing the tasks in parallel.

In order to calculate the expected cloud speedup, the formula introduced in Section 4.1 can be used. For the sake of simplicity, we assume that the fraction of the serializable processing can be neglected, because we are interested in big amounts of data:

$$\begin{aligned} cloud\ speedup &= \frac{1}{s + \frac{p}{N} + o} \\ &= \frac{1}{0 + \frac{1}{N} + o} \end{aligned}$$

Now, the effective cloud overhead o, which is the sum of all types of effort introduced with the distributed processing, has to be calculated. In this example, there was no detailed monitoring, and the scheduling was rather simple, because only a single host was available. Since neither detailed monitoring information nor information about the scheduling strategy is available, it is not possible to predict the overhead for a larger amount of hosts yet.

Even though the prediction uses incomplete information, the increase of execution time caused by network transfer might give a good indication of the performance that can be expected by using a larger amount of hosts. As already stated in the description of this scenario, the limiting resource could be the I/O throughput. Therefore, the network bandwidth was measured in order to determine the upper bound of cloud hosts that can be used.

	Size [MB]	Time [s]	Bandwidth [MB/s]
Upload	174.002	20.004	8.698
Download	160.704	11.913	13.490

Table 6.2: Measured bandwidth between servers

As it can be seen in Table 6.2, the upload into the cloud is a little bit slower than the download. Due to the fact that the speed of upload and download is nearly equal, the weighted mean can be calculated without loss of precision. In this scenario the weighted mean is 10.487 MB/s. Since network usually is volatile compared to other resources (i.e., the available bandwidth varies over time), it is assumed to be 10 MB/s. Note that there could be SLAs, which give a good indication of the network capacity that can be expected. Thus it might not be necessary to measure network performance in detail to get sufficient information for calculating its impact on the scaling capacity of an application. In order to calculate the cloud speedup, we have to determine the degree of parallelization as follows:

$$\begin{aligned} degree\,of\,parallelization &= \frac{avg.\,bandwidth}{rendering\,throughput} \\ &= \frac{10\,\text{MB/s}}{1.88\,\text{MB/s}} \approx 5.32 \end{aligned}$$

Due to the network bandwidth, up to five cloud hosts can be fully utilized. Every host added after the sixth one does not result in any performance improvement. Based on this result and the measured overhead of 40%, we calculate the cloud speedup as follows:

$$\begin{aligned} cloud\,speedup &= \lim_{N \to 5.32} \frac{1}{\frac{1}{N} + o} \\ &= \frac{1}{\frac{1}{5.32} + \frac{0.4}{5.32}} = 3.8 \end{aligned} \tag{6.1}$$

In other words, by using six (or more) computation nodes and optimal scheduling, the computation is up to 3.8 times faster than a non-distributed execution. However, optimal scheduling cannot always be achieved because it is known to be Nondeterministic Polynomial-time hard (NP-hard) [9] and usually depends on many parameters.

Nevertheless, with this kind of information one can calculate the amount of hosts that are necessary to perform a big number of tasks faster than with non-distributed execution. However, this leads to additional overhead, namely the startup and shutdown time of the hosts. The startup phase includes all work that has to be done to run the application. In case of image-based virtual hosts, this includes the loading of the system images containing the operating system and any additional data for running the application (e.g., the application itself). In the shutdown phase all acquired resources have to be freed (e.g., the state of the system image is discarded).

For this particular scenario we assume that it takes up to 60 seconds to start a host. Furthermore, it takes up to 30 seconds to terminate it (e.g., signing off from the cloud and closing connections, discarding the system image, freeing the occupied IP address, etc.). Since we are interested in the point where parallel execution outperforms serial execution, we have to solve the following equation:

$$\begin{aligned} serial\ execution &= parallel\ execution \\ x &= static\ overhead + \frac{x}{cloud\ speedup} \\ x &= 60\ \text{sec} + 30\ \text{sec} + \frac{x}{3.8} \\ x - \frac{x}{3.8} &= 90\ \text{sec} \\ 2.8x &= 90\ \text{sec} \cdot 3.8 \\ x &= 122 + \frac{1}{7}\text{sec} \end{aligned} \tag{6.2}$$

According to Equation 6.2, when serial execution takes longer than 122 seconds, distributed processing pays off even if there is fixed size overhead of 90 seconds.

Measurements Scenario 1

Figure 6.1 illustrates the general approach of scheduling several tasks ($T1$, ..., $T8$) using two computation nodes. As it can be seen, before and after every task, the client (C) has to transmit the data to the hosts ($H1$, $H2$) and retrieve the result, respectively.

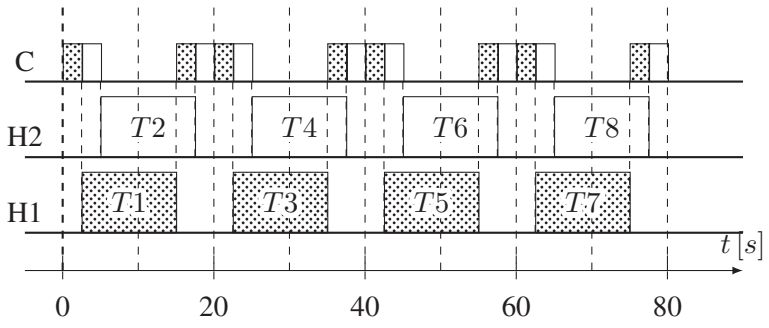

Figure 6.1: Example for overprovisioning (two hosts for eight tasks)

Figure 6.2 shows the total execution time for 20 tasks with increasing number of cloud hosts. In particular, it shows the impact of scheduling on execution time. According to the prediction, the best performance is reached when six machines are used. However, as it can be seen in Figure 6.2, the total execution time for five and seven machines is slightly lower than for six.

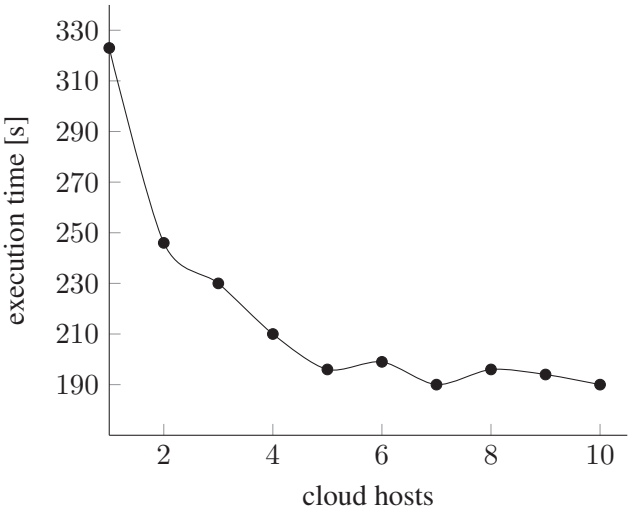

Figure 6.2: Measured execution time with an increasing number of cloud hosts

As already mentioned before, best performance can only be achieved if optimal scheduling is used. When using five computation nodes, the idle times are lower than with six because at most five machines can be fully utilized. In other words, every machine does not have to wait that long for the next task to be assigned. In contrast to that, when using seven computation nodes, the idle times are considerable longer, but 20 tasks can be processed in three iterations, while five or six machines require four iterations. Thus the efficiency is worse, but the performance is better than using five or six instances. Whenever one or more computation nodes do not have a task to process, there will be a decrease of performance. Even though if there are no pauses due to incomplete task assignment, during the last iteration half of the computation nodes will idle on average because the amount of tasks is not a multiple of the number of nodes. In general, the following rule applies to scheduling: the higher the amount of available computation nodes and the less the number of tasks to process, the more machines will be idle at some point in time. Figure 6.3 visualizes this behavior.

Note that the total time of the distributed execution increases sub-linearly compared to non-distributed execution. Nevertheless, the scheduling itself introduces some kind of threshold. If t tasks cannot be distributed uniformly among N computation nodes (i.e., $(N \bmod t) > 0$ is true), an additional iteration is required as it can be seen in Figure 6.4. In theory, distributed execution pays off if 12 tasks (t') have to be processed on 6 (or more) hosts. However, due to inefficient scheduling, 16 tasks (t'') are required to outweigh the time wasted on waiting for task assignment.

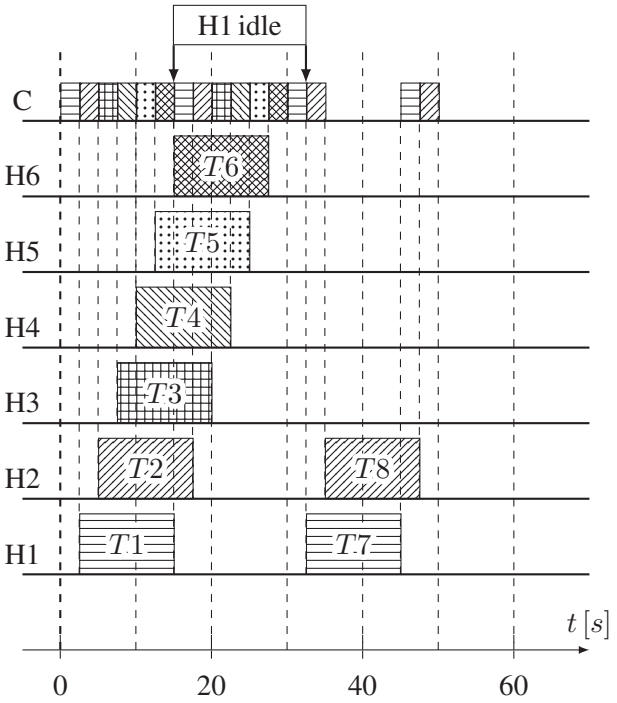

Figure 6.3: Overprovisioning (client schedules eight tasks on six hosts)

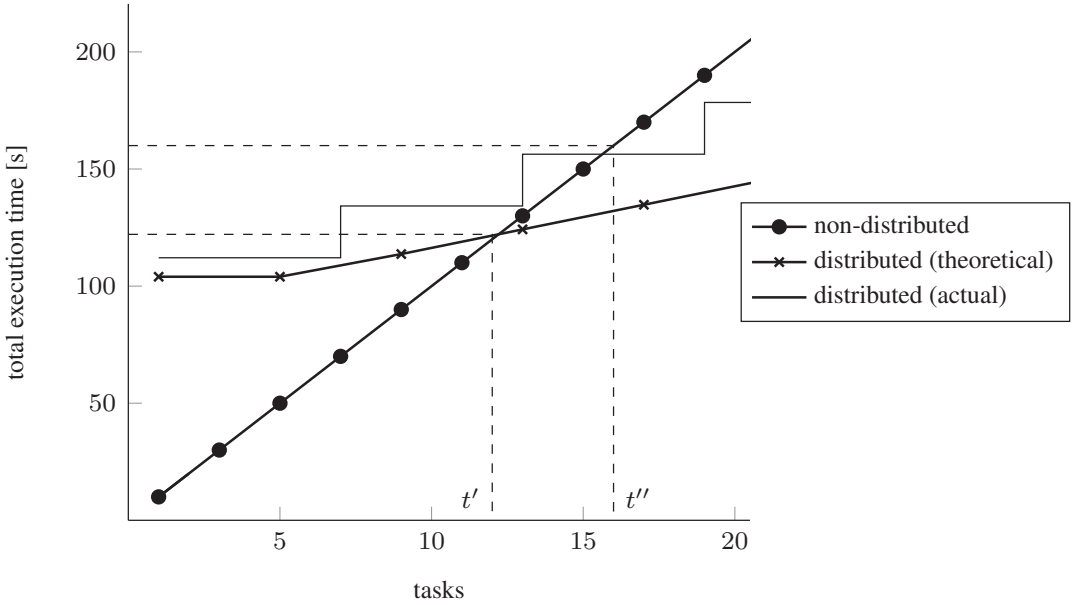

Figure 6.4: Processing of an increasing number of tasks with six computation nodes

Predictions Scenario 2

Populations	Time [s]
1	46.306
2	99.290
4	196.609
8	394.854
16	792.637

Table 6.3: Measured performance with increasing population size

In order to estimate the cloud overhead, a single measurement was performed with two bacteria populations using a single cloud host. The processing took 60.306 seconds, which is almost 40% less compared to non-distributed processing (see Table 6.3). Even though there is not as much network overhead as in the other scenario, distribution overhead cannot be negative by definition. Thus the application behaves quite differently and further investigation had to be done for the purpose of predicting the further behavior of the performance.

Profiling Scenario 2

In order to profile the application, the ALASCA profiler introduced in Chapter 5 was used. The results of the measurements are summarized in Figure A.1 and Figure A.2. Although there is a considerable profiling overhead, there is one anomaly that can easily be seen. While running non-distributed, the execution time of `NucleotideChromosome.mutate()` was 30.529 seconds. By transmitting the single task to another CH, the execution time was reduced to 7.593 seconds. In other words, the mutation of the population is four times slower when running the application not distributed. Due to the fact that a single task cannot be parallelized, the profiling runs have been repeated several times with different JVM memory parametrization in order to determine under which conditions memory management becomes inefficient. However, all runs had in common that the distributed execution had a similar performance or even was considerably faster than non-distributed execution. This leads to the conclusion that efficient memory management is very crucial for this type of application.

Measurements Scenario 2

The application allocates memory whenever a new DNA strand is generated. Thus the mutation rate has a huge impact on the memory management. Figure 6.5 shows the performance degradation of the application with an increasing bacteria population and different mutation rates. It can be seen that the execution time increases non-linearly. Thus it can be concluded that the mutation rate has a larger impact on the performance than the DNA crossover or other aspects of the simulation.

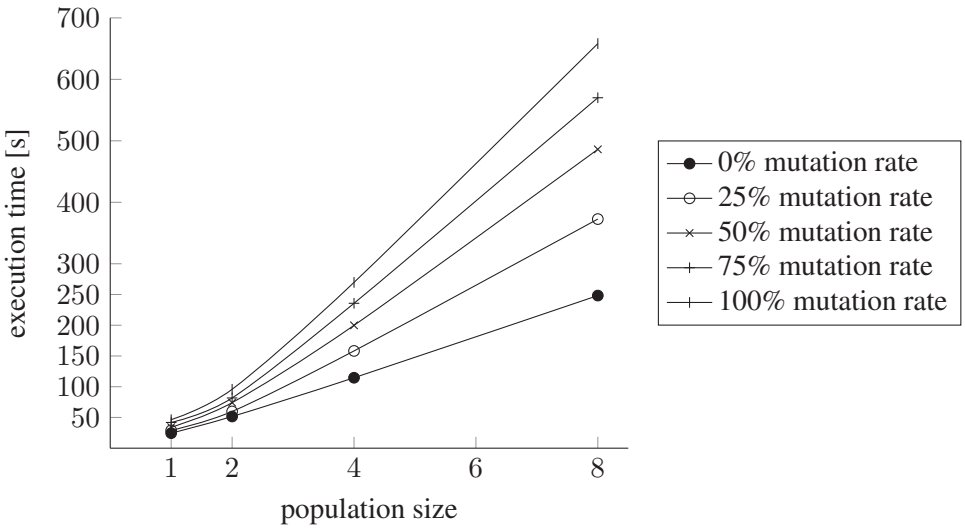

Figure 6.5: Measured execution time with an increasing bacteria population

Now we can apply our model for predicting the performance of the application running in a distributed environment. Since there is (almost) no cloud overhead, the cloud speedup is theoretically infinite:

$$\begin{aligned} cloud\ speedup &= \lim_{N\to\infty} \frac{1}{\frac{1}{N} + o} \\ &= \frac{1}{\frac{1}{\infty} + \frac{0}{\infty}} = \infty \end{aligned}$$

In other words, it scales linearly with the number of CHs. This fact leads to the conclusion that any fixed size overhead (e.g., 90 seconds startup time) can be neglected if there are many tasks that can be executed in parallel. By assuming that an average task takes x seconds, and the number of parallel computations N is equal to the number of tasks T, the following equation holds:

$$\begin{aligned} static\ overhead + \frac{T \cdot x}{cloud\ speedup} &= T \cdot x \\ 90\ \text{sec} + \frac{T \cdot x}{N} &= T \cdot x \\ 90\ \text{sec} + \frac{T}{T}x &= T \cdot x \\ 90\ \text{sec} + x &= T \cdot x \\ \frac{90\ \text{sec}}{x} + 1 &= T \\ \frac{90\ \text{sec}}{24.416\ \text{sec}} + 1 &= T \\ 4.686 &\approx T \end{aligned} \tag{6.3}$$

In this particular example, where the measured time for x is 24.416 seconds, the result would be $T = 5$. This means that the simulation of a five times bigger population, which runs in parallel on five CHs, is faster than running it on a single computation node. However, in order to get an accurate prediction of the scalability, the parametrization of the simulation has to be considered as well. Figure 6.5 has shown that the execution time does not increase linearly with an increasing mutation rate. Therefore, the cloud overhead o cannot be considered to be constant. Instead, it has to be a function that depends on the mutation rate. According to the results from the previous measurements, a simulation, where the mutation rate is 100%, takes approximately 40% longer than a simulation, where no bacteria mutates, when running in a distributed environment. When running the simulation non-distributed, the execution time is even 90% higher compared to a mutation rate of 0%. Even though the execution time increases almost linearly with small deviations, it can be considered to increase linearly, because of the jitter, which is caused by the indeterminism of the memory management of the JVM [38]. Figure 6.6 shows the variation of the total execution time as a function of population size and mutation rate. It can be seen that the execution time strictly increases for higher mutation rate and/or population size. Moreover, doubling the population size has a bigger impact than doubling the mutation rate.

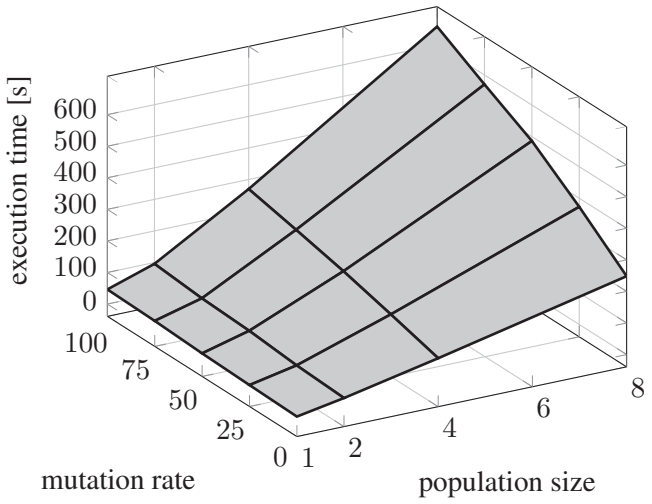

Figure 6.6: Execution time with increasing population size and mutation rate

By adding this information to Equation 6.3, another formula can be obtained, which is much more precise. Let $f(m, s)$ be the function that calculates the (average) execution time of a single task depending on the mutation rate m ($m \in [0, 1]$) and the population size s ($s > 0$):

$$\begin{aligned} 90 + \frac{T \cdot f(m,s)}{cloud\,speedup} &= T \cdot f(m,s) \\ \frac{90}{f(m,s)} + 1 &= T \end{aligned} \tag{6.4}$$

In order to get a better understanding of the impact of the population size and the mutation rate on the execution time, Figure 6.7 shows the respective increase. On the left side, the solid lines represent the relative population size (1×, 2×, 4×, 8× compared to the initial size). The dotted lines show the lower and upper bound of the execution time assuming that it increases linearly with the population size. On the right side the solid lines depict the mutation rates in steps of 25%. The dotted lines show the lower and upper bound of the execution time assuming that execution time increases linearly with the mutation rate.

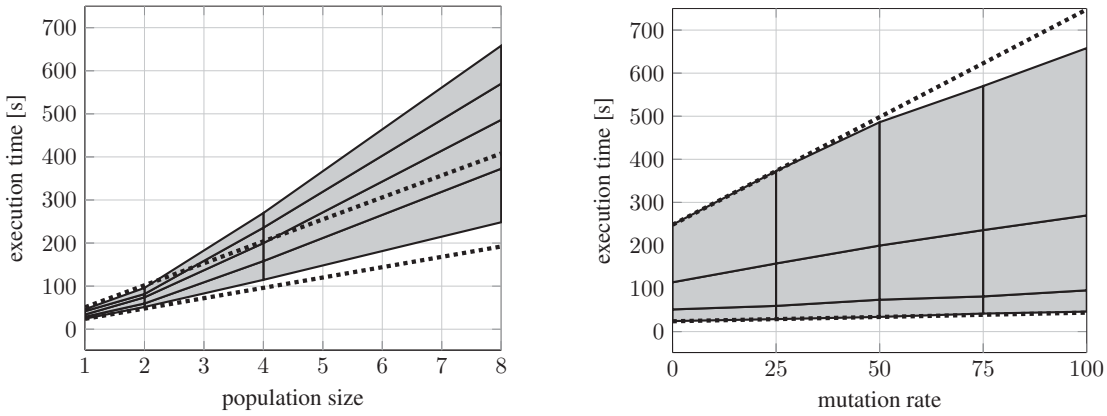

Figure 6.7: Execution time for various fixed mutation rates and increasing population size, and for various fixed population sizes and increasing mutation rate, respectively

The mutation rate has considerable impact on the execution time. Although it can be seen that a linear rise increases the execution time sublinearly. Similar to that, it has been shown that a linear rise of the population size has a superlinear impact on the execution time.

By calculating $f(m, s)$ with the lower and upper bound of m and s, we can determine the number of hosts that have to be used in order to outweigh the cloud overhead. In Equation 6.3 the upper bound for $m = 0$ and $s = 1$ has already been calculated. Equation 6.4 can be used to calculate the lower bound (e.g., $m = 1$ and $s = 8$):

$$\begin{aligned} \frac{90}{f(m_{max}, s_{max})} + 1 \leq T \leq \frac{90}{f(m_{min}, s_{min})} + 1 \\ \frac{90}{f(1, 8)} + 1 \leq T \leq \frac{90}{f(0, 1)} + 1 \\ 2.094 \leq T \leq 4.686 \end{aligned} \tag{6.5}$$

As it can be seen in Equation 6.5, starting two computation nodes for a simulation that contains eight times as many individuals with a high mutation rate almost pays off. The additional setup and shutdown time of 90 seconds in total can be neglected. Better results can be achieved with even larger simulations.

While the application used in the first scenario has linear performance characteristics, the application of this scenario behaves differently in many ways. First, it has no hard limit like network throughput. Second, the high demand of memory entails considerable indeterminism regarding performance metrics. Third, application parameters like mutation rate and population size have non-linear impact on execution time. All those factors have to be considered in order to predict the performance accurately. We have shown that the model proposed in Chapter 4 along with profiling information enables performance predictions of applications running in cloud environments.

CHAPTER 7

Conclusions

In theory, there is no difference
between theory and practice.
But, in practice, there is.

— Jan L. A. van de Snepscheut

This chapter concludes the thesis with a short summary of the results. Furthermore, the research questions from Section 1.2 are revisited. Finally, the relevance of the predictions for similar applications is discussed.

7.1 Summary

First, this thesis discussed cloud computing and outlined its importance as well as the basic idea of our performance prediction model. Second, background information on various related state-of-the-art topics including complexity theory, Amdahl's and Gustafson's Law, computation models, and common cloud technologies and environments were given. In general, lots of research has been done on these topics during the last years. Thus a selected set of related work has been presented. As this thesis focuses on performance, we discussed papers about resource allocation and scheduling, cloud middleware, application profiling, and models for cost approximation. Subsequently, we introduced a general model for predicting scalability of applications running in cloud environments based on Amdahl's and Gustafson's Law and profiling information, and described the procedure with the aid of an exemplary scenario. Next, we introduced the ALASCA profiler, which was implemented for verifying the accuracy of our model. In particular, we explained the architecture of ALASCA and how it is used for profiling distributed applications. Finally, we did experiments on performance predictions by applying the proposed model on selected applications with high I/O, CPU and memory requirements. We collected profiling information and explained deviations from the predictions. In general, our approach

leads to accurate predictions, unless there is some indeterminism or the application behaves quite differently when running in a distributed way. The outcome of this thesis is discussed below.

7.2 Research Questions

This thesis has been guided by the research questions introduced in Section 1.2. The following section summarizes the results of the work that has been done in order to address them.

How can the speedup of distributable applications running in a cloud environment be predicated by using metrics from non-distributed execution? Do some kinds of applications have a general model that describes their cloud performance?

In principle, there are various ways of how an application can be classified. For the scenarios introduced in Section 6.1, both applications consisted of a large part that can be distributed among multiple nodes and executed individually. Unlike other applications, there is almost no need for synchronization or other overhead. Even if this seems to be a good starting point, the predictions in Section 6.3 have shown that there are many factors that have to be taken into consideration. Scheduling, for example, is NP-hard without loss of generality, however, for large problem sets, it is very likely that it can be neglected as long as the overall CH utilization is quite good.

Based on the modified variants of Amdahl's and Gustafson's Law proposed in Chapter 4, we predicted the performance of selected applications. Therefore, we profiled those applications and collected runtime information as input for our calculations. On the one hand, our model has lead to accurate predictions, which have been proven by detailed measurements. On the other hand, we also encountered that indeterminism (e.g., in terms of memory management or even parametrization) can cause considerable inaccuracy. Thus we can conclude that applications, which have a constant runtime profile can be predicted rather easily. For applications, which have a volatile runtime behavior, it is still possible to determine the maximum cloud speedup. It acts as a fixed boundary and may indicate that an application is not suited for distributed execution under the given conditions.

Is it possible to reason about the point where cloud execution outweighs the distribution overhead including scaling, scheduling and communication?

As it has been shown in Section 6.3, scheduling has a significant impact on performance. On the one hand, being not able to fully utilize all nodes results in suboptimal performance. Typical reasons are insufficient network bandwidth or contempt of data locality. On the other hand, going for full utilization at all costs often entails a degradation of efficiency. These observations correlate with the results of [28], where the authors conclude that the MapReduce implementation Hadoop performs well until the simulation ran out of resources.

Despite of that, cloud overhead has been measured precisely using the ALASCA profiler. If the overhead is quite high compared to the overall computation effort, one can conclude that the benefits of executing an application in a distributed way certainly will not outbalance the effort.

Are there any indications that an application will not scale well in the cloud and how can they be verified? Do they correspond to static models like Amdahl's Law or are there any other factors that have to be considered?

Even if the amount of parallelizable code is high, there is no guarantee that the cloud overhead outweighs the theoretical speedup. For instance, the outcome of the analysis of scenario 2 introduced in Section 6.1 would be totally different if the CHs would synchronize their bacteria population onto the other machines at every iteration step. As described in [13], there are different approaches for synchronizing state in distributed systems. Most of them can be classified into two main categories, namely conservative and optimistic mechanisms. Conservative mechanisms can lead to good performance for certain classes of problems. Although optimistic mechanisms can achieve great performance if the worst case (e.g., acquiring exclusive locks) does not happen very frequently [8]. Furthermore, it has been shown in [30] [31] that optimistic approaches like Time Warp may perform poorly for some types of application. Besides that there are various reasons that can lead to bad performance or overproportional high overhead. As we have seen in Section 6.3, resource bottlenecks like insufficient network throughput build hard boundaries. However, typically they are quite easy to detect.

7.3 Future Work

In both Amdahl's and Gustafson's Law the ratio between serial and parallel code is assumed to be constant. While this assumption is perfectly fine for the code of most applications running on a single machine, it might vary when running an application in the cloud. First, the total amount of instructions executed is higher, because of the overhead. In addition, it is very likely that a significant amount of indeterminism is entailed by distributed execution. For example, the scheduling strategy used for assigning tasks behaves differently between two runs. Second, parts of the code that can be executed in parallel are parallelizable to a limited extent. In other words, the cloud provides more computation power than the amount that can be used by the program in some of its sections. Therefore, only a fraction of the parallelizable code can be expressed as a function rather than a constant. Further work will investigate under which circumstances the ratio between serial and parallel code changes over time. Additionally, both laws will be extended to support those factors, which would further improve the quality of the performance predictions.

As already mentioned in Chapter 1, this thesis omits economical aspects such as operational costs. An interesting addition is the deduction of concrete costs based on the results of this thesis. By combining an approach for maximizing financial profit like in [46] with in-depth profiling analysis, it will be possible to make profound assumptions about costs before running an application. In particular this is interesting for companies running applications with different performance characteristics in heterogeneous environments.

APPENDIX A

Measurements

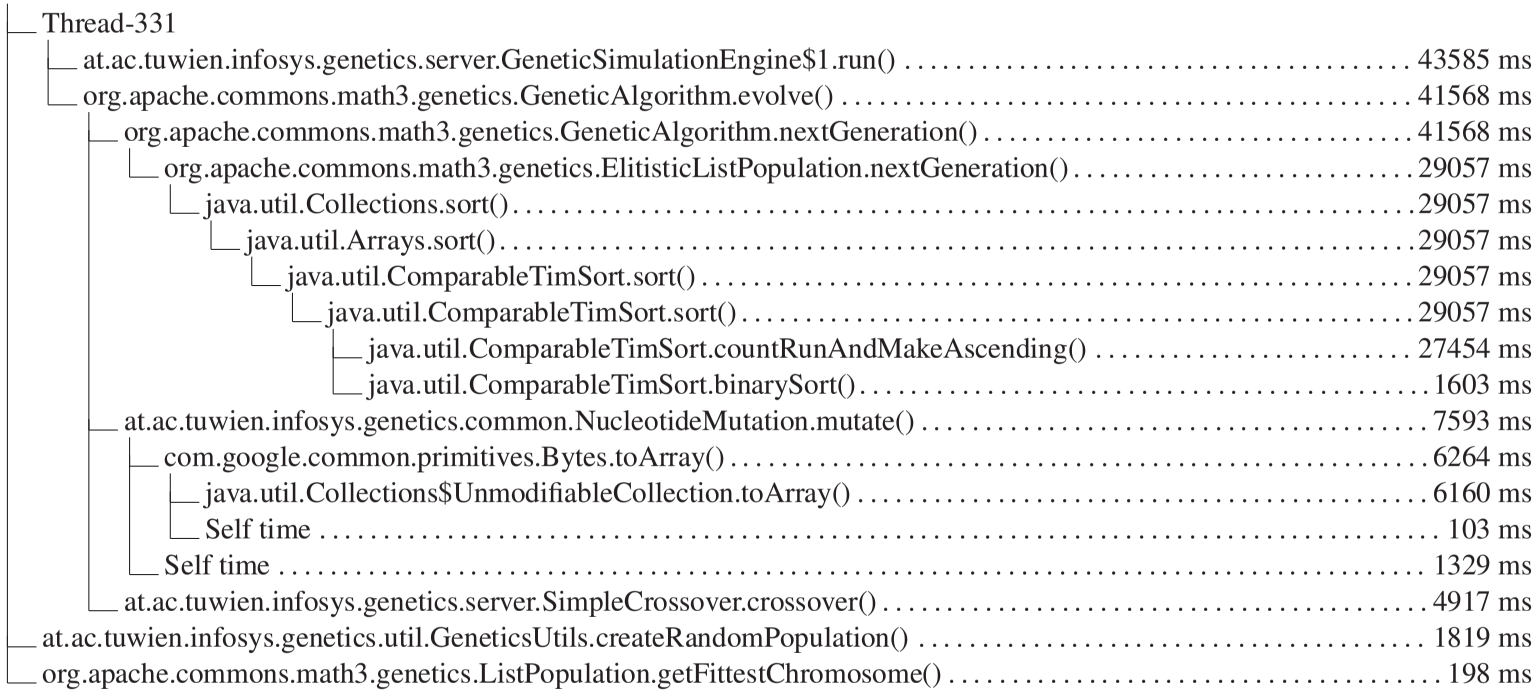

Figure A.1: Profiling information of simulation thread running in a distributed way

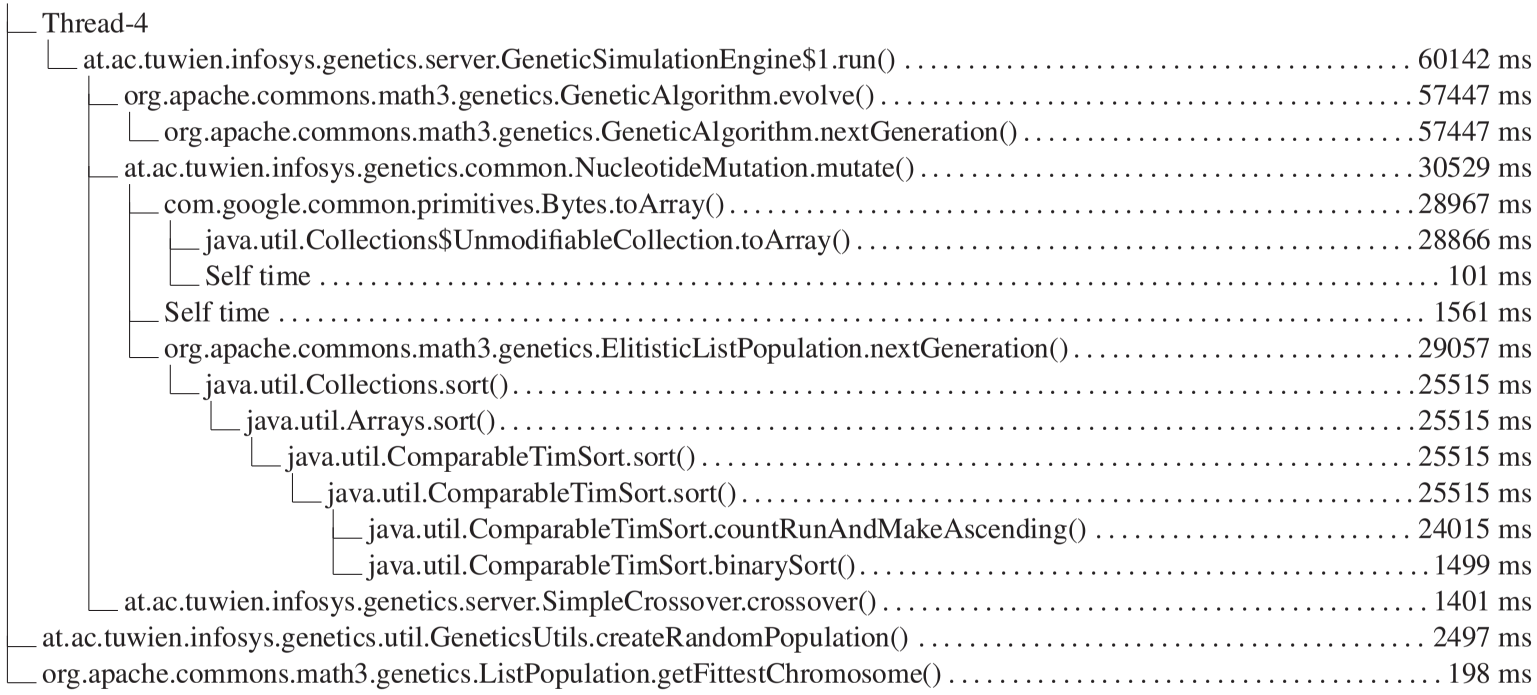

Figure A.2: Profiling information of simulation thread (non-distributed)

APPENDIX B

Acronyms

A Adenine.

AI Artificial Intelligence.

AOP Aspect Oriented Programming.

API Application Programming Interface.

C Cytosine.

CH Cloud Host.

CO Cloud Object.

CPU Central Processing Unit.

DAG Directed Acyclic Graph.

DBMS Database Management System.

DNA Deoxyribonucleic Acid.

EC2 Amazon Elastic Compute Cloud.

FIFO First In, First Out.

G Guanine.

GUI Graphical User Interface.

HDD Hard Disk Drive.

HDFS Hadoop Distributed File System.

I/O Input/Output.

IA Intelligent Agent.

IaaS Infrastructure as a Service.

IP Internet Protocol.

JDBC Java Database Connectivity.

JMS Java Message Service.

JMX Java Management Extensions.

JSON JavaScript Object Notation.

JVM Java Virtual Machine.

NAS Network-Attached Storage.

NFS Network File System.

NIST National Institute of Standards and Technology.

NP-hard Nondeterministic Polynomial-time hard.

OLAP Online Analytical Processing.

ORB Object Request Broker.

PaaS Platform as a Service.

QoS Quality of Service.

RAM Random Access Memory.

RDB Relational Database.

REST Representational State Transfer.

RMI Remote Method Invocation.

RPC Remote Procedure Call.

SaaS Software as a Service.

SDK Software Development Kit.

SLA Service Level Agreement.

SQL Structured Query Language.

SSD Solid-State Drive.

T Thymine.

TCP Transmission Control Protocol.

URL Uniform Resource Locator.

XML Extensible Markup Language.

Bibliography

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. → pages 3 and 8.

[2] Yair Amir, Baruch Awerbuch, Amnon Barak, R. Sean Borgstrom, and Arie Keren. An Opportunity Cost Approach for Job Assignment in a Scalable Computing Cluster. *IEEE Trans. Parallel Distrib. Syst.*, 11(7):760–768, July 2000. → page 26.

[3] Kyoungho An, Subhav Pradhan, Faruk Caglar, and Aniruddha Gokhale. A Publish/Subscribe Middleware for Dependable and Real-time Resource Monitoring in the Cloud. In *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*, SDMCMM '12, pages 3:1–3:6, New York, NY, USA, 2012. ACM. → page 23.

[4] Tekin Bicer, David Chiu, and Gagan Agrawal. Time and Cost Sensitive Data-Intensive Computing on Hybrid Clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 636–643, May 2012. → page 21.

[5] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. → pages xi, 11, and 12.

[6] Parijat Dube, Seetharami Seelam, Yanbin Liu, Megumi Ito, Thomas Ling, Michel Hack, Liana Fong, Graeme Johnson, Michael Dawson, Li Zhang 0002, and Yuqing Gao. A Tool for Scalable Profiling and Tracing of Java and Native Code Interactions. In *QEST*, pages 37–46. IEEE Computer Society, 2011. → page 24.

[7] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Trans. Softw. Eng.*, 24(5):342–361, May 1998. → page 35.

[8] Richard M. Fujimoto, Asad Waqar Malik, and Alfred J. Park. Parallel and Distributed Simulation in the Cloud. *SCS Modeling and Simulation Magazine*, 1, 2010. → page 61.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. → page 49.

[10] Shamsollah Ghanbari and Mohamed Othman. A Priority Based Job Scheduling Algorithm in Cloud Computing. *Procedia Engineering*, 50:778–785, 2012. → page 25.

[11] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988. → pages 4 and 9.

[12] M Hemamalini. Review on Grid Task Scheduling in Distributed Heterogeneous Environment. *International Journal of Computer Applications*, 40(2):24–30, 2012. → page 23.

[13] Shafagh Jafer, Qi Liu, and Gabriel Wainer. Synchronization methods in parallel and distributed discrete-event simulation. *Simulation Modelling Practice and Theory*, 30:54–73, 2013. → page 61.

[14] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The Performance of MapReduce: An In-depth Study. *Proc. VLDB Endow.*, 3(1-2):472–483, September 2010. → page 2.

[15] Kenneth Johnson, Simon Reed, and Radu Calinescu. Specification and Quantitative Analysis of Probabilistic Cloud Deployment Patterns. In *Hardware and Software: Verification and Testing*, pages 145–159. Springer, 2012. → page 27.

[16] Ben H. H. Juurlink and Cor H. Meenderinck. Amdahl's Law for Predicting the Future of Multicores Considered Harmful. *SIGARCH Comput. Archit. News*, 40(2):1–9, May 2012. → page 7.

[17] (Vasiliki) Verena Kantere, Debabrata Dash, Georgios Gratsias, and Anastasia Ailamaki. Predicting Cost Amortization for Query Services. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 325–336, New York, NY, USA, 2011. ACM. → page 25.

[18] Yumiko Kasae and Masato Oguchi. Proposal for an Optimal Job Allocation Method for Data-intensive Applications based on Multiple Costs Balancing in a Hybrid Cloud Environment. In *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '13, pages 5:1–5:8, New York, NY, USA, 2013. ACM. → page 23.

[19] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP*. SpringerVerlag, 1997. → page 37.

[20] Nane Kratzke and Peter-Christian Quint. About Automatic Benchmarking of IaaS Cloud Service Providers for a World of Container Clusters. *Journal of Cloud Computing Research*, 1(1):16–34, 2015. → page 17.

[21] Pardeep Kumar and Amandeep Verma. Scheduling Using Improved Genetic Algorithm in Cloud Computing for Independent Tasks. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, ICACCI '12, pages 137–142, New York, NY, USA, 2012. ACM. → page 23.

[22] Wei-Po Lee, Yu-Ting Hsiao, and Wei-Che Hwang. Designing a parallel evolutionary algorithm for inferring gene networks on the cloud computing environment. *BMC Systems Biology*, 8:5, 2014. → pages 24 and 46.

[23] Philipp Leitner, Benjamin Satzger, Waldemar Hummer, Christian Inzinger, and Schahram Dustdar. CloudScale - a Novel Middleware for Building Transparently Scaling Cloud Applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 434–440, New York, NY, USA, 2012. ACM. → page 35.

[24] Ang Li, Xuanran Zong, Srikanth Kandula, Xiaowei Yang, and Ming Zhang. CloudProphet: Towards Application Performance Prediction in Cloud. *SIGCOMM Comput. Commun. Rev.*, 41(4):426–427, August 2011. → page 26.

[25] Ang Li, Xuanran Zong, Ming Zhang, Srikanth Kandula, and Xiaowei Yang. CloudProphet: Predicting Web Application Performance in the Cloud. *ACM SIGCOMM Poster*, 2011. → page 26.

[26] Ching-Chi Lin, Pangfeng Liu, and Jan-Jan Wu. Energy-Aware Virtual Machine Dynamic Provision and Scheduling for Cloud Computing. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 736–737, July 2011. → page 25.

[27] Jing Liu, Xing-Guo Luo, Xing-Ming Zhang, Fan Zhang, and Bai-Nan Li. Job Scheduling Model for Cloud Computing Based on Multi-Objective Genetic Algorithm. *IJCSI International Journal of Computer Science Issues*, 10(1):134–139, 2013. → page 25.

[28] Xavier Llorà, Abhishek Verma, Roy H. Campbell, and David E. Goldberg. When Huge is Routine: Scaling Genetic Algorithms and Estimation of Distribution Algorithms via Data-Intensive Computing. In *Parallel and Distributed Computational Intelligence*, 2010. → pages 24, 46, and 60.

[29] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. → page 23.

[30] Asad Waqar Malik, Alfred J. Park, and Richard M. Fujimoto. Optimistic Synchronization of Parallel Simulations in Cloud Computing Environments. In *Cloud Computing, 2009. CLOUD '09. IEEE International Conference on*, pages 49–56, Sept 2009. → page 61.

[31] Asad Waqar Malik, Alfred J. Park, and Richard M. Fujimoto. An Optimistic Parallel Simulation Protocol for Cloud Computing Environments. *SCS M&S Magazine*, 4:1–9, 2010. → page 61.

[32] Toni Mastelic, Vincent C. Emeakaroha, Michael Maurer, and Ivona Brandic. M4Cloud - Generic Application Level Monitoring for Resource-shared Cloud Environments. In Frank Leymann, Ivan Ivanov, Marten van Sinderen, and Tony Shan, editors, *CLOSER*, pages 522–532. SciTePress, 2012. → page 23.

[33] Toni Mastelic, Drazen Lucanin, Andreas Ipp, and Ivona Brandic. Methodology for trade-off analysis when moving scientific applications to Cloud. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 281–286, Dec 2012. → page 26.

[34] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), Gaithersburg, MD, September 2011. → page 1.

[35] Edison Mera, Pedro López-García, Germán Puebla, Manuel Carro, and Manuel V. Hermenegildo. Combining Static Analysis and Profiling for Estimating Execution Times. In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages*, PADL'07, pages 140–154, Berlin, Heidelberg, 2007. Springer-Verlag. → page 24.

[36] Michael P. Mesnier, Matthew Wachs, Raja R. Simbasivan, Julio Lopez, James Hendricks, Gregory R. Ganger, and David R. O'Hallaron. //TRACE: Parallel Trace Replay with Approximate Causal Events. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*. USENIX Association, 2007. → page 26.

[37] Rizwan Mian, Patrick Martin, Farhana Zulkernine, and Jose Luis Vazquez-Poletti. Estimating Resource Costs of Data-Intensive Workloads in Public Clouds. In *Proceedings of the 10th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '12, pages 3:1–3:6, New York, NY, USA, 2012. ACM. → page 25.

[38] Sun Microsystems. Memory Management in the Java HotSpot Virtual Machine, 2006. http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf. → page 55.

[39] Pradeep Naik, Surbhi Agrawal, and Srikanta Murthy. A survey on various task scheduling algorithms toward load balancing in public cloud. *American Journal of Applied Mathematics*, 3(1-2):14–17, 2015. → page 25.

[40] Diego Perez-Palacin, Radu Calinescu, and José Merseguer. log2cloud: Log-based Prediction of Cost-Performance Trade-offs for Cloud Deployments. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 397–404, New York, NY, USA, 2013. ACM. → page 27.

[41] Huang Qi-yi and Huang Ting-lei. An Optimistic Job Scheduling Strategy based on QoS for Cloud Computing. In *Intelligent Computing and Integrated Systems (ICISS), 2010 International Conference on*, pages 673–675, Oct 2010. → page 22.

[42] Colby Ranger, Christos Kozyrakis, Ramanan Raghuraman, Arun Penmetsa, and Gary Bradski. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA*, 02/2007 2007. → page 2.

[43] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 7:1–7:13, New York, NY, USA, 2012. ACM. → page 15.

[44] John C. Reynolds. The Discoveries of Continuations. *Lisp Symb. Comput.*, 6(3-4):233–248, November 1993. → page 36.

[45] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003. → pages xi, 13, and 14.

[46] Konstantinos Tsakalozos, Herald Kllapi, Eva Sitaridi, Mema Roussopoulos, Dimitris Paparas, and Alex Delis. Flexible Use of Cloud Resources Through Profit Maximization and Price Discrimination. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 75–86, Washington, DC, USA, 2011. IEEE Computer Society. → pages 21 and 61.

[47] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific Languages: An Annotated Bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000. → page 23.

[48] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015. → page 15.

[49] Rostyslav Zabolotnyi, Philipp Leitner, and Schahram Dustdar. Profiling-Based Task Scheduling for Factory-Worker Applications in Infrastructure-as-a-Service Clouds. In *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, pages 119–126, Aug 2014. → page 22.

[50] Rostyslav Zabolotnyi, Philipp Leitner, Waldemar Hummer, and Schahram Dustdar. JCloudScale: Closing the Gap Between IaaS and PaaS. *ACM Transactions on Internet Technology*, 15(3):10:1–10:20, July 2015. → pages 35 and 37.

[51] Rostyslav Zabolotnyi, Philipp Leitner, Stefan Schulte, and Schahram Dustdar. SPEEDL - A Declarative Event-Based Language for Cloud Scaling Definition. In *The Future of Software Engineering For and In Cloud, Visionary Track of IEEE Services*, IEEE Computer Society, JUN 2015. s.n. → page 23.