Detecting Bottlenecks in Parallel DAG-based Data Flow Programs

Dominic Battré, Matthias Hovestadt, Björn Lohrmann, Alexander Stanik and Daniel Warneke
Technische Universität Berlin
Email: {firstname}.{lastname}@tu-berlin.de

***Abstract*—In recent years, several frameworks have been introduced to facilitate massively-parallel data processing on shared-nothing architectures like compute clouds. While these frameworks generally offer good support in terms of task deployment and fault tolerance, they only provide poor assistance in finding reasonable degrees of parallelization for the tasks to be executed. However, as billing models of clouds enable the utilization of many resources for a short period of time for the same cost as utilizing few resources for a long time, proper levels of parallelization are crucial to achieve short processing times while maintaining good resource utilization and therefore good cost efficiency.**

In this paper, we present and evaluate a solution for detecting CPU and I/O bottlenecks in parallel DAG-based data flow programs assuming capacity constrained communication channels. The detection of bottlenecks represents an important foundation for manually or automatically scaling out and tuning parallel data flow programs in order to increase performance and cost efficiency.

I. Introduction

Scientific computing today faces datasets which are increasing exponentially in both complexity and size. The challenge of processing these datasets in a reasonable amount of time has made a case for new approaches in the area of parallel and distributed computing. Recently, terms like many-task computing (MTC) [1] or data intensive scalable computing (DISC) [2] have been coined to describe a computational paradigm in which complex processing jobs are split into a large number of loosely-coupled tasks and executed in a parallel fashion.

While several data processing frameworks have already demonstrated good support for MTC-like jobs [1] (in terms of scalability, fault tolerance, etc.), they only offer very little support with respect to choosing a reasonable level of parallelization for those jobs. Current tutorials on this topic (e.g. [3]) mainly propose back-of-the-envelope calculations and ignore the characteristics of the job, e.g. its computational complexity. Furthermore, in many cases MTC-like processing jobs consist of several individual tasks. However, the challenge of adjusting the individual tasks' degree of parallelization according to their computational complexity is not addressed at all at the moment.

This lack of assistance is problematic because exactly this relationship between a task's computational characteristics and its degree of parallelization is a major factor towards high processing throughput and system utilization. Since the tasks of such MTC-like jobs are typically executed following a classic producer-consumer pattern [4], [5], [6], each task depends on sufficient amounts of input data in order to reach its optimal throughput. Therefore, for computationally demanding tasks, a low degree of parallelization may easily result in CPU bottlenecks, which slow down the execution of successive tasks and leave large parts of the compute resources underutilized. In contrast to that, choosing a degree of parallelization that is too high for computationally modest tasks can lead to significant overhead for task deployment and instantiation.

In the area of cloud computing, operators of Infrastructure-as-a-Service (IaaS) clouds like Amazon EC2 [7] or the Rackspace Cloud [8] let their customers rent a virtually unlimited number of CPU cores and amount of main memory and charge them for the usage of the resources on an hourly basis. In their cost model using a thousand CPU cores for an hour is no longer more expensive than using a single CPU core for a thousand hours. Therefore it is tempting for researchers to strive for shorter completion time of jobs by increasing their level of parallelization. Of course, the vast majority of compute jobs, especially the data-intensive ones, cannot be parallelized indefinitely. At some level of parallelization, the I/O subsystem of the rented cloud resources (i.e. the bandwidth of the hard disks and the network links) will become an insuperable bottleneck. Parallelization beyond that point will leave the rented CPU cores underutilized and will unnecessarily increase the cost for executing the job. However, at what level of parallelization this ultimate bottleneck will occur is hard to predict.

But even in classic cluster environments the appropriate degree of parallelization is often not easy to choose. Limiting factors such as the number of available CPU cores or the available amount of main memory typically provide a natural upper bound for a reasonable number of parallel tasks. However, choosing the individual scale-out of interdependent tasks, which run on the cluster in a distributed fashion, quickly becomes a considerable obstacle.

In this paper we approach the problem of finding sensible levels of parallelization for MTC-like data processing jobs which can be described as directed acyclic graphs (DAGs). Therefore we devise a mechanism to detect CPU and I/O bottlenecks during the job runtime and assist the developer in improving the scale-out of his application through successive runs of the same job. Our overall optimization goal is to

978-1-4244-9705-8/10/$26.00 ©2010 IEEE

discover the maximum level of parallelization for each of the processing job's tasks which still yields a high overall system utilization. The presented approach is computationally lightweight and requires neither any modifications of nor cooperation from the tasks themselves.

Section II provides a thorough description of the class of processing jobs our work is applicable to. After a brief problem statement in Section III, we discuss the concrete bottleneck detection algorithms in Section IV. Section V puts the theoretical consideration of the bottleneck problem into practice using our execution framework Nephele [9]. In Section VI we demonstrate the usefulness of our approach with the help of a concrete use case. Finally, Section VII discusses related work while Section VIII concludes the paper and sheds some light on our future endeavors with regard to optimizing parallel data flow programs.

II. Prerequisites

The bottleneck detection approach we will present in this paper can be applied to arbitrary MTC-like processing jobs which fulfill the assumptions described in the following:

- **Assumption 1:** The processing job can be modeled as a DAG $G = (V_G, E_G)$. Therein, each vertex $v \in V_G$ of the DAG represents a separate *task* of the overall processing job. The directed edges $e \in E_G$ between the vertices model the *communication channels* through which data is passed on from one task to the next. Figure 1 a) illustrates an example DAG with four vertices, which represent four distinct tasks.
- **Assumption 2:** The interaction between the individual tasks of the processing job follows a producer-consumer pattern. Tasks exchange data through communication channels in distinct units we will call *records*. We assume all communication between tasks takes place through communication channels modeled in the DAG.
- **Assumption 3:** A communication channel is unidirectional and can be modeled as a buffered queue (FIFO) of records. The channel's *capacity* shall be the size of the buffer, i.e., an arbitrary but fixed value which states the number of records a producing task can write to the channel without having another task consuming the data. Any attempts to write records to a channel beyond its capacity limit will cause the producing task to be blocked until the consuming task has removed at least one record from the channel. Analogous to this, any attempt to read from a channel which currently does not hold any records will cause the consuming task to be blocked until at least one record is written to the channel.
- **Assumption 4:** Each task of the DAG consists of sequential code. It can be parallelized in a single program, multiple data (SPMD) manner so that each *task instance* receives a subset of the task's overall input data. Figure 1 b) illustrates a possible job DAG with those parallel task instances.
- **Assumption 5:** At runtime, each task instance is in one of the following states: PROCESSING or WAITING. A state change is always triggered by one of its connected communication channels. A task instance is in state WAITING when it is either waiting to write records to an outgoing channel or waiting for records to arrive from an incoming channel; otherwise it is in state PROCESSING. Hence, if sufficient input records are available and capacity is left to write out the result records, a task instance will not enter the WAITING state. The current state of a task instance can be accessed anytime during its execution. Note that even waiting for I/O other than communication channel I/O (e.g. reading or writing from/to hard disk) is therefore considered as processing time.
- **Assumption 6:** At runtime, each communication channel is either in the state SATURATED when its buffer's capacity limit has been reached; otherwise it is in state READY. Similar to the tasks, we assume the current state of a channel can be accessed anytime throughout its lifetime.
- **Assumption 7:** If a specific record is processed by the same task in different job executions, the performance characteristics (processing time, value and size of produced output) remain the same. This assumption allows profiling a job and using the gained knowledge to improve a second execution of the job.

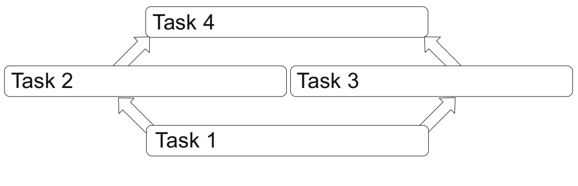
III. Problem Description

The general idea of a bottleneck is that it is either a task or a communication channel in the job's DAG that slows down other parts of the processing chain and that the processing time of the entire job would improve if the bottleneck were to be alleviated in some way. We shall define two different types of bottlenecks:

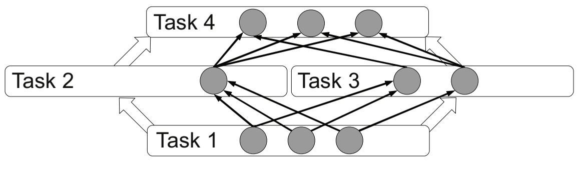
- **CPU bottlenecks** are tasks whose throughput is limited by the CPU resources they can utilize. CPU bottlenecks are distinguished by the fact that they have sufficient amounts of input data to process, however, subsequent tasks in the processing chain suffer from a lack thereof.
- **I/O bottlenecks** are those communication channels which are requested to transport more records per time unit than the underlying transport infrastructure (e.g. network interconnects) can handle.

The problem is to detect such bottlenecks using only the information on the task and channel states which have been described in the previous section. Through this abstraction our approach becomes independent of the concrete physical compute and communication resource, which may be hard to observe in (shared) virtualized environments like IaaS clouds.

It is important to point out that our bottleneck detection approach considers the individual tasks of a job DAG. This means, we do not aim at detecting CPU or I/O bottlenecks at individual parallel instances of a task. Bottlenecks at individual task instances typically indicate load balancing problems. They do not provide any clues about bottlenecks which stem from inappropriate levels of parallelization of distinct tasks, which is the focus of this paper. So although a task may be executed as hundreds or thousands of parallel instances, algorithmically we will treat it as a single task.



(a)



(b)

Fig. 1. An exemplary directed acyclic graph from a non-parallelized, conceptual point of view (a) and a parallelized, execution point of view (b)

Multiple bottlenecks might exist within a job at runtime. Moreover, eliminating a bottleneck at one specific task might create another bottleneck at a different task. Hence, we regard bottleneck *elimination* essentially as an iterative process, that always starts with bottleneck *detection*. The latter is the problem we deal with in this paper.

IV. Detecting Bottlenecks

We will now present an algorithm that is capable of detecting both CPU and I/O bottlenecks in DAG-based parallel data flow programs. The algorithm is applicable to any data processing job which fits the model and assumptions presented in Section II. It is triggered periodically during the job execution.

Algorithm 1 illustrates the overall approach of our bottleneck detection algorithm. The algorithm is passed the DAG G which represents the currently executed job. Initially, the function $ReverseTopologicalSort(G)$ (line 1) creates and returns a list L_{RTS} with all vertices of G. The order of the vertices within the list corresponds to a reverse topological ordering, i.e. vertices with no outgoing edges appear first in the list.

Algorithm 1 DetectBottlenecks($G := (V_G, E_G)$)

```
1:  $L_{RTS} \leftarrow ReverseTopologicalSort(G)$
2:  for all $v$ in $L_{RTS}$ do
3:      $v.isCpuBottleneck \leftarrow IsCpuBottleneck(v, G)$
4:  end for
5:  if $\nexists v \in L_{RTS} : v.isCpuBottleneck$ then
6:      for all $v$ in $L_{RTS}$ do
7:          $E_v = \{(v,w) | w \in V_G \wedge (v,w) \in E_G\}$
8:          for all $e \in E_v$ do
9:              $e.isIoBottleneck \leftarrow IsIoBottleneck(e, G)$
10:         end for
11:     end for
12: end if
```

The list L_{RTS} is then traversed from the beginning to the end. For each vertex v we check whether v is considered a CPU bottleneck. The particular properties for a vertex to meet the CPU bottleneck condition are checked within the function $IsCpuBottleneck(v, G)$ (line 3), which is explained later in this section. The result of the check is returned and stored in the variable $v.isCpuBottleneck$.

In order to detect I/O bottlenecks, we take a similar approach. Again, we traverse each vertex v of the job DAG G according to their reverse topological order. For each outgoing edge $e = (v, w)$ of v we check whether e meets the conditions of an I/O bottleneck. However, we only perform the detection if no CPU bottleneck has been discovered before. The discussion of Algorithm 3 later in this section will clarify the necessity for this constraint.

Algorithm 2 describes how we check whether a particular vertex $v \in V_G$ is a CPU bottleneck. The algorithm checks for two conditions which must be fulfilled in order to classify v as a CPU bottleneck.

A crucial prerequisite for a CPU bottleneck is that the task represented by vertex v spends almost the entire CPU time given to it in the state PROCESSING. We introduce the function $pt(v)$ which shall be defined as the arithmetic mean of the fractions of time the task's instances spent in the state PROCESSING during the last time unit of their execution. Synchronization issues may cause individual task instances to spend short periods of time in the state WAITING. For this reason we introduce a threshold α for $pt(v)$ which must be exceeded so that v is considered a bottleneck. In our practical experiments we found 90% to be a reasonable value for α.

Algorithm 2 IsCpuBottleneck(v, G)

```
1: if $pt(v) \leq \alpha$ then
2:     return $FALSE$
3: end if
4: if $\exists s \in vsucc^*(v, G) : s.isCpuBottleneck$ then
5:     return $FALSE$
6: end if
7: return $TRUE$
```

The second condition for a CPU bottleneck considers the set of v's successors, $vsucc^*(v, G)$, i.e. the vertices which can be reached from v. Formally, a vertex s is in $vsucc^*(v, G)$ if there exists a path $p = (v_1, ..., v_n)$ such that $v_1 = v$, $v_n = s$ and $(v_i, v_{i+1}) \in E_G$, $1 \leq i < n$. For each such successor s we check if s has been classified as a CPU bottleneck. The order in which we traverse the vertices in the job DAG G guarantees that the CPU bottleneck flag $s.isCpuBottleneck$ of all of vertex v's successors has been updated before the function $IsCpuBottleneck$ is called with v itself.

The necessity for this second condition becomes apparent when recalling our definition of a CPU bottleneck. According to that definition, a CPU bottleneck is characterized by high CPU load and the fact that it provides successor vertices with insufficient amounts of input data. However, if any successor s of vertex v was already identified as a CPU bottleneck, this would mean s does not suffer from insufficient amounts of input data because the amount of input data s receives is sufficient to max out its CPU time. As a result, classifying vertex v as a CPU bottleneck requires all of its successors not to be classified as CPU bottlenecks.

Algorithm 3 IsIoBottleneck($e := (v, w)$, G)

```
1: if $st(e) \leq \beta$ then
2:     return $FALSE$
3: end if
4: if $\exists s \in esucc^*(v, G) : s.isIoBottleneck$ then
5:     return $FALSE$
6: end if
7: return $TRUE$
```

After all CPU bottleneck flags have been updated Algorithm 3 checks whether an edge should be considered an I/O bottleneck. For an edge $e = (v, w) \in E_G$, $st(e)$ denotes the arithmetic mean of the fractions of time the communication channels represented by the edge e spent in the state SATURATED during the last time unit of v's execution.

Similar to CPU bottlenecks, we consider two conditions for I/O bottlenecks. First, $st(e)$ must be above a threshold β which indicates that the communication edges represented by the edge e spent the majority of the considered time interval in the state SATURATED. In practice we found 90% to be a reasonable threshold for β, so temporary fluctuations in the channel utilization do not lead to wrong bottleneck decisions.

The second condition again considers the successors of an edge e. By $esucc^*(e, G)$ we shall denote the set of successor edges of e. Formally, an edge $s = (t, u)$ is in $esucc^*(e, G)$ if there exists a path $p = ((v_0, v_1), ..., (v_{n-1}, v_n))$ such that $(v_i, v_{i+1}) \in E_G$, $0 \leq i < n$ and $v_0 = v$, $v_1 = w$, $v_{n-1} = t$, and $v_n = u$. An edge e is only classified as an I/O bottleneck if no successor edge has been classified as an I/O bottleneck before. Again, the order in which we traverse the edges ensures the appropriate update of the I/O bottleneck flags.

The I/O bottleneck approach bears some discussion. Generally, there exist two possible reasons for high values of $st(e)$. The first reason is that the maximum throughput rate of the underlying transport infrastructure which backs the communication channel has been reached. This corresponds to our definition of an I/O bottleneck in Section III. The second possible, however spurious, reason is an insufficient consumption rate of the task which consumes data from e. This, in turn, can be caused by two circumstances: First, a CPU bottleneck in the DAG could affect the consumption rate of the respective task. However, since we only check for I/O bottlenecks if no CPU bottleneck has been detected before, this cause can be eliminated. Second, another I/O bottleneck could exist in the remainder of the processing chain. Yet, this is impossible because of the second condition (line 4) of Algorithm 3.

V. Practical Implementation

After having described our bottleneck detection algorithm in theory we will now highlight its practical implementation as part of our data processing framework Nephele [9]. This includes a brief introduction of Nephele's fundamental concepts as well as a discussion on how the respective state of tasks and their communication channels can be obtained at runtime. Finally we will present a graphical user interface (GUI) which provides immediate visual feedback of the detected bottlenecks during the job's execution and thereby assists developers to determine a reasonable scale-out for their jobs.

A. The Nephele Execution Framework

The basis for the implementation of our bottleneck detection approach is the Nephele framework [9]. Nephele executes parallel data flow programs which are expressed as DAGs on large sets of shared-nothing servers, e.g. IaaS clouds. It keeps track of task scheduling and setting up the required communication channels. In addition, it dynamically allocates the compute resources during program execution and helps to mitigate the impact of temporary or permanent hardware outages.

Nephele's architecture is based on a classic master-worker pattern. A central component called the *Job Manager* coordinates and distributes tasks among of a set of workers, which are called *Task Managers*.

Following the model introduced in Section II, each vertex of a Nephele DAG represents a task of the overall processing job. Each task can be assigned to different (types of) compute nodes. In addition, the level of parallelization can be adapted for each task of the DAG individually. E.g., it is possible to create a large number of parallel instances for compute-intensive tasks while maintaining a low number of parallel instances for those tasks which are mainly I/O bound.

The edges of the DAG represent the communication channels between the tasks, which are created by Nephele at runtime. Through these communication channels the parallel instances of a task can either consume incoming data from their preceding tasks in the processing chain or forward data to succeeding tasks. Currently, Nephele supports three different types of channels: network, in-memory, and file channels. Each channel operates on distinct data units, similar to the notion of records discussed in Section II.

File channels store the data that is written to them either on a local or a distributed file system. The data is not passed on to the consuming task until the last piece of data has been written by the producing task.

Network channels build upon regular TCP connections to transport data. Data produced by one task is immediately shipped to the consuming task. As a result of the network connection, the producing and the consuming task can be scheduled to run on different compute nodes.

In-memory channels exchange data through the compute nodes' main memory. Unlike network or file channels, in-memory channels do not have to convert between the object and the byte-stream representation of the records they transport because only references to the data are transported through the channels. Consequently, in-memory channels achieve a substantially higher throughput in terms of records per second than network channels. However, the producing and consuming tasks are required to run on the same compute nodes and even within the same operating system processes.

A more thorough discussion of Nephele, in particular in terms of its scheduling capabilities, can be found in [9].

B. Profiling Nephele Jobs

A key requirement to apply the bottleneck detection algorithm as described in Section IV is the ability to access the state information of tasks and their communication channels at runtime, since these values are the foundation for implementing the utilization functions $pt(v)$ and $st(e)$. This process is referred to as job profiling.

On an operating system level collecting data about the CPU utilization of individual processes is easy. Under Linux, e.g., the `/proc/` interfaces offer detailed and easily accessible statistics on the individual operating system processes.

However, in order to facilitate fast memory-based communication between two tasks Nephele cannot always map different tasks to different operating system processes. Instead, the usage of in-memory channels forces Nephele to instantiate different tasks as different threads within the same process. With respect to collecting the profiling information this means that we also have to be able to monitor the CPU utilization of individual threads. Since most parts of Nephele are written in Java, there are several different options to achieve this goal.

Our first profiling approach was based on the Java Virtual Machine Tool Interface (JVMTI). JVMTI provides access to the internal state of the Java Virtual Machine (JVM). It allows writing so-called agents in a native language like C or C++, so unlike Java itself the profiling extension is platform dependent. The agent is then executed in the same process as the JVM and is notified about occurring events via callback functions.

Our second profiling approach relied on the Java Management Extension (JMX). JMX comprises the MXBeans platform package which provides access to, among other things, the JVM's runtime, thread and memory subsystem. In particular, we used the class `ThreadMXBean` to determine the CPU time of individual threads.

In order to evaluate the impact of both profiling approaches on the actual task execution, we implemented both approaches and devised a CPU-intensive sample job. We executed the sample job several times without the profiling component as well as with the JVMTI or JMX-based profiling component enabled. Each version of the profiling component queried the information on the CPU utilization of the monitored task thread every second. The results of the comparison are depicted in Figure 2.

Without profiling the mean execution time was around 82 seconds. The JMX-based profiling component proved to be very lightweight. It only increased the mean execution time by less than 1%. In contrast to that, the JVMTI-based component led to a significant drop in execution speed. On average the tasks' completion time was increased by almost 74%. A closer examination revealed that the frequent calls of the native agent code were the main reason for the performance penalty.

As a result of the first performance evaluation, we implemented the functions $pt(v)$ and $st(e)$ based on the lightweight JMX approach. In order to generate the values of $pt(v)$ we query the JMX interface every 5 seconds for statistics on every

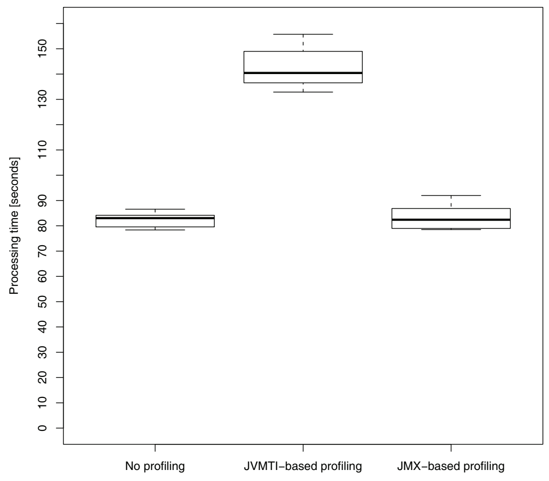


Fig. 2. Profiling overhead using the JVMTI- and JMX-based approach

task thread. The statistics let us derive detailed information on the distribution of the thread's CPU time among the different internal states of the JVM. These internal states are:

- **USR:** The amount of time the monitored task thread was executed in user mode.
- **SYS:** The amount of time the monitored task thread was executed in system mode.
- **BLK:** The amount of time the monitored task thread was blocked because of mutual exclusion.
- **WAIT:** The amount of time the monitored task thread was intentionally instructed to wait.

Since the threads of Nephele tasks only enter the WAIT state as a result of congested or starved communication channels, we can map this state directly to the WAITING state of our bottleneck algorithm. The other three internal JVM states (USR, SYS, BLK) are mapped to the PROCESSING state. This also complies with Assumption 5 of Section II.

In order to determine the utilization of communication channels, we simply store a timestamp for the respective channel whenever a task thread either attempts to read from or write to the channel and the attempt leads to the task thread switching its state from PROCESSING to WAITING. Moreover, we store a timestamp whenever new incoming data from the considered channel or the completed transmission of outgoing data allow the task to switch from state WAITING back to the state PROCESSING (cf. Assumption 5). Based on these timestamps we can then calculate how much time the channel spent in the states SATURATED and READY.

After having calculated the utilization statistics for each task instance locally at the respective Task Managers, the profiling data is forwarded to Nephele's central management component, the Job Manager. The Job Manager then calculates an arithmetic mean of the individual task instance statistics. During the experiments on our mid-size cloud testbed, the amount of data that was generated by our profiling subsystem was negligibly small and did not account for any observable load on the Job Manager. In larger setups, when scalability might become a bigger concern, the profiling overhead can be reduced by increasing the reporting interval.

C. Graphical Representation of Bottlenecks

In order to inform the job's developer about detected bottlenecks at execution time we devised a special GUI for Nephele, which is depicted in Figure 3. The GUI contacts the Job Manager through an RPC interface to receive recent profiling information about currently executed jobs and the compute nodes involved in the execution.

For each job the GUI displays detailed information on each of the job's individual tasks and, assuming a task is executed in parallel, the task instances. Developers can examine the utilization of each task instance and its communication channels in a chronological sequence and thereby track down problems with regard to workload distribution. Tasks or communication channels which are considered CPU or I/O bottlenecks respectively are visually highlighted.

The feedback about the utilization of a job's individual tasks is complemented by a graphical presentation of the compute nodes which are involved in executing the respective job. Here the developer can examine charts about the CPU, memory, and network utilization of individual nodes or view summary charts which display the average utilization of all those compute nodes.

VI. Evaluation

In this section we want to evaluate the usefulness of our bottleneck detection approach with the help of a concrete use case. The use case we consider picks up the idea from the introduction of this paper: A developer intends to use an IaaS cloud like Amazon EC2 to repeatedly run an MTC-like processing job. The job consists of several individual tasks that interact with each other. The developer strives to finish the processing job as fast as possible without paying for unutilized compute resources.

A. Evaluation Use Case

The job we devised for our use case is inspired by the famous Hadoop job of the New York Times, which was used to convert their 4 TB large article archive from TIFF images to PDF using 100 virtual machines on EC2 [10].

In our case the conversion job consists of six distinct tasks as depicted in Figure 4. The first task, *File Reader*, initially reads the individual image files from disk and sends each image as a separate record to the second task, *OCR Task*. The OCR Task then applies a text recognition algorithm to the received image. The result of the text recognition, a regular string, is then processed in a twofold manner. First, the recognized text pattern of the image is passed to the task *PDF Creator*. Second, each word within the text pattern is forwarded to a task *Inverted Index Task* together with the name of the original image file.

The task PDF Creator receives the recognized text pattern of each original image as a separate record. The text pattern is

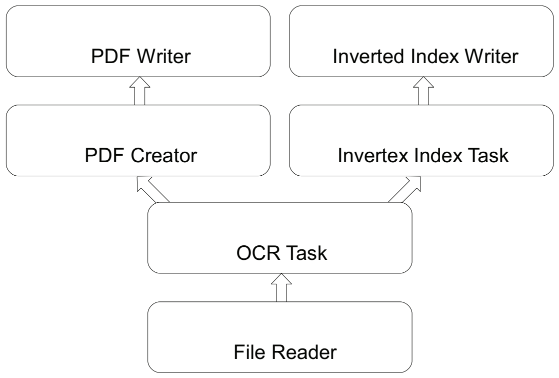


Fig. 4. The Nephele job used for the evaluation

then converted to a PDF document and emitted to the task *PDF Writer*, which eventually writes the received PDF document back to disk.

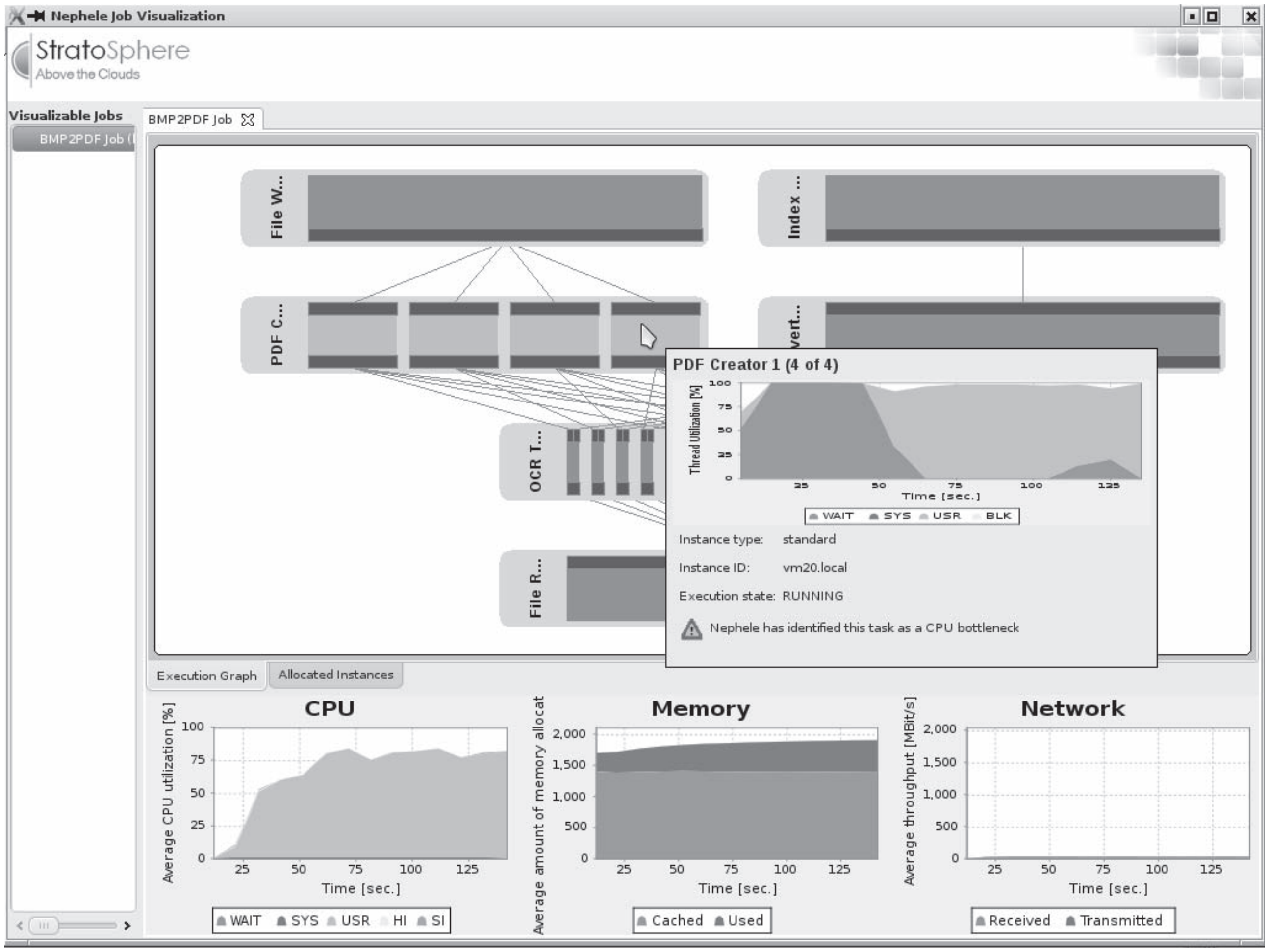
The Inverted Index Task receives tuples of words from the recognized text patterns and the names of the originating image files. As its name implies, the task uses the received tuples to construct an inverted index. This inverted index can later be used to facilitate a keyword search on the created PDF file set. In our evaluation the created index has been small enough to fit into the node's main memory. After having received all input tuples, the task sends out tuples of each indexed word together with a list of filenames in which the word occurs to the task *Inverted Index Writer*. Inverted Index Writer then writes the received tuples back to disk.

Conceptually, the processing job is interesting because the tasks OCR Task, PDF Creator, and Inverted Index Task suggest having different computational complexities. In order to achieve a busy processing pipeline together with an economic job execution on the cloud, each task's degree of parallelization must be carefully balanced with respect to the other task.

Since many IaaS providers offer their virtual machines with ephemeral storage, i.e. it is not possible to store data inside the virtual machine beyond its termination, we assume the set of images we use as input data is stored on a persistent storage service similar to Amazon Elastic Block Storage [11]. Right before the start of the processing job the storage service is mounted inside one particular virtual machine. This virtual machine then executes the tasks File Reader, PDF Writer, and Inverted Index Writer, so the input and output data is directly read or written from/to the storage service. The remaining tasks are executed on other virtual machines inside the cloud. Our overall goal is to find the largest possible scale-out of the job with high resource utilization before the virtual machine serving the input data becomes an insuperable I/O bottleneck.

B. Evaluation Setup

The evaluation experiments were conducted on our local compute cloud of commodity servers. Each server is equipped with two Intel Xeon E5430 2.66 GHz CPUs (type E5430, 4 CPU cores each) and a total main memory of 32 GB. All servers are connected through regular 1 GBit/s Ethernet links. As host operating system we installed Gentoo Linux (kernel



Instance type: standard

Instance ID: vm20.local

Execution state: RUNNING

Nephele has identified this task as a CPU bottleneck

Fig. 3. Nephele's job visualization displaying the live profiling information and highlighting a CPU bottleneck

version 2.6.32) with KVM [12] (version 0.12.5) and used virtio [13] to provide virtual I/O access.

In order to provision virtual machines we used the Eucalyptus cloud system [14]. Each parallel instance of the tasks OCR Task, PDF Creator, and Inverted Index Task has been executed on a separate virtual machine with one CPU core, 2 GB of main memory and 60 GB disk space. Amazon EC2 offers virtual machines with comparable characteristics (except for the disk space) at a price of approximately 0.10 USD per hour. Each virtual machine booted a standard Ubuntu Linux image (kernel version 2.6.31) with no additional software but a Java runtime environment (version 1.6.0_20), which is required by Nephele's Task Manager. Throughout the entire evaluation all tasks were connected via Nephele's network channels.

The input data set for the sample job consisted of 4000 bitmap files. Each bitmap file contained a regular page of single-column text and had a size of approximately 10 MB. As a result, the overall size of the input data set was 40 GB.

The PDF documents were created using the iText library [15]. To mimic the persistent storage service mentioned in the use case we set up a regular NFS server. The server has been connected with 1 GBit/s to the rest of the network.

C. Evaluation Results

The results of our evaluation are depicted in Figure 5. The figure shows selected runs of our sample job with different degrees of parallelization. For each of those runs we depict the average CPU utilization of all compute nodes involved in the execution over the time. The CPU utilization was captured by successively querying the `/proc/stat` interface on each node and then sending the obtained values to Nephele's Job Manager to compute the global average for the respective point in time. Besides the CPU utilization chart, we illustrate the respective CPU or I/O bottlenecks which have been reported by our bottleneck detection algorithm in the course of the processing. Boxes with shaded areas refer to CPU bottlenecks while boxes with solid areas refer to I/O bottlenecks at the outgoing communication channels.

Figure 5 a) depicts the first of our evaluation runs. As a first approach we used a parallelization level of 1 for all the six tasks of the sample job. As a result, the job execution comprised four virtual machine with the File Reader, the PDF Writer and the Inverted Index Writer running on one virtual machine and the OCR Task, the PDF Creator and the Inverted Index Task each running on a separate virtual machine.

The processing time of the job was about 5 hours and 10 minutes. In the entire processing period the average CPU

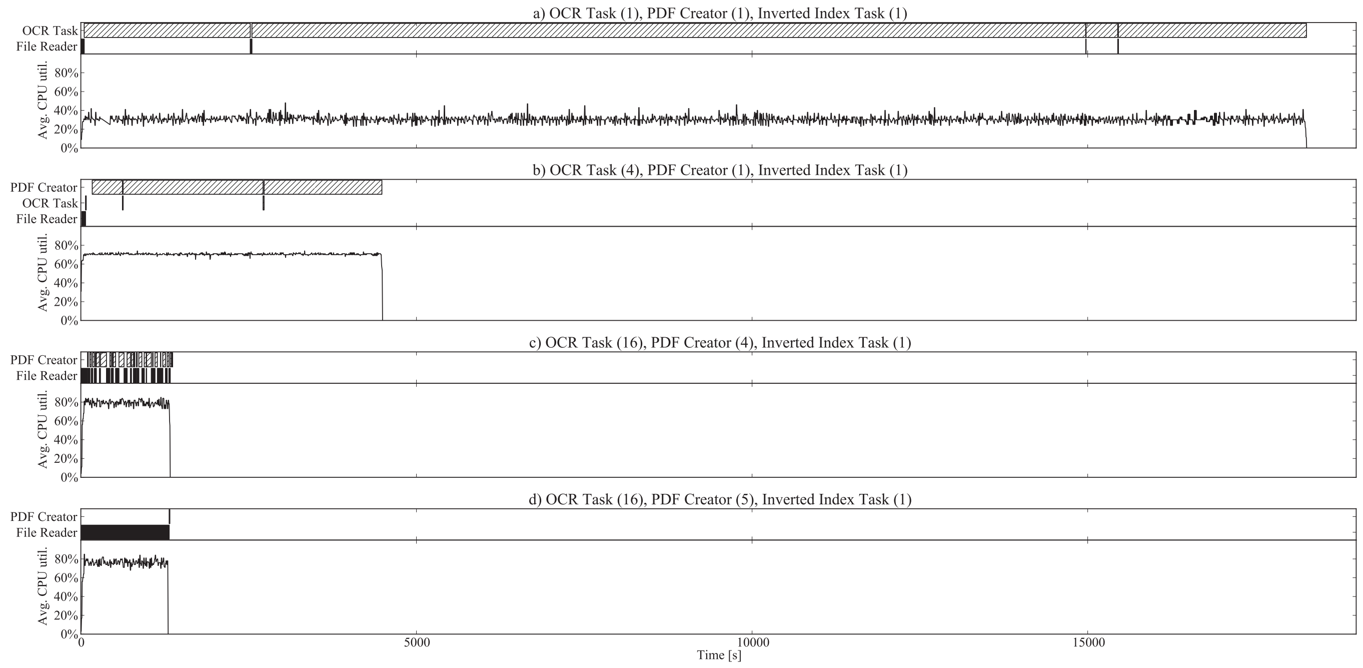


Fig. 5. Avg. CPU utilization and detected bottlenecks (shaded areas are CPU bottlenecks, solid areas are I/O bottlenecks) for different scale-outs

utilization ranged between 30% and 40%. The reason for this poor resource utilization becomes apparent when looking at the bottleneck chart for the run. Almost the entire time the OCR Task has been identified as a CPU bottleneck by our bottleneck detection algorithm.

As a response to the observation of the first run, we followed the strategy to first improve the average CPU utilization by balancing the individual tasks' degree of parallelization according to their relative complexity. After having determined a reasonable parallelization ratio among the tasks we began to scale out. This approach requires that the computational characteristics of a task are independent of its level of parallelization (cf. Assumption 7).

We continuously increased the degree of parallelization for the OCR task and reexecuted the job. The average CPU utilization continued to improve up to the point where the OCR task had 4 parallel instances (see Figure 5 b)). Up to level 3 the OCR task remained the permanent CPU bottleneck. However, at parallelization level 4, the PDF Creator task became the dominant bottleneck. In this configuration, with 7 virtual machines, the overall processing time had decreased to 1 hour and 15 minutes while the average CPU utilization had climbed up to approximately 70% throughout the entire execution time. Note that we did not have to wait for the intermediate runs to complete in order to deduce the final parallelization ratio between the OCR and the PDF Creator task. Since we knew the computational characteristics of both tasks would not change during the processing time, it was sufficient to observe only several seconds of each run and then to proceed to the next level of parallelization. For jobs consisting of several distinct processing phases, which interrupt the processing pipeline, a longer observation might be necessary.

After having done the initial balancing we began to scale out both the OCR and the PDF Creator task at a ratio of 4 to 1. In a configuration with 16 parallel instances of the OCR Task and 4 parallel instance of the PDF Creator task (see Figure 5 c)) we again encountered a change in the bottleneck situation. We witnessed frequent changes between the PDF Creator task as a CPU bottleneck and the communication channels of the File Reader task as an I/O bottleneck. The changes were caused by Nephele's internal buffer strategy for network channels. In order to achieve a reasonable TCP throughput, data is shipped in blocks of at least 16 KB size. The text patterns recognized by the parallel instances of the OCR Task had an average size of about 4 KB, so the text pattern sometimes arrived at the parallel instances of the PDF Creator task in batch and caused a temporary CPU bottleneck.

However, despite the frequent changes, the bottleneck bars in the diagram also indicate that the communication edge between the File Reader task and the OCR Task has essentially become an I/O bottleneck which renders further parallelization of successive tasks unnecessary. This is confirmed by our final run depicted in Figure 5 d). After adding another parallel instance to the PDF Creator task we observed the communication channel between the File Reader and OCR Task to be a permanent I/O bottleneck.

Interestingly, the Inverted Index Task had no significant effect on the job execution. In comparison to the OCR Task and the PDF Creator task its computational complexity turned out to be too low. Moreover, the 4000 documents we used to populate the index only accounted for a memory consumption of a few MB. The channel congestion which may have occurred when transferring the index to the Inverted Index Writer task was too short to be detected by our system.

In sum, we think the evaluation provides a good example for the usefulness of our bottleneck detection approach. The initial job execution without parallelization (Figure 5 a)) took over 5 hours on 4 virtual machines to complete. Assuming an hourly cost of 0.10 USD per virtual machine, this amounts to a processing cost of 2.40 USD. Through the assistance of our bottleneck detection algorithm we could follow specific indications to scale out our sample job according to the complexity of each individual task. Although the final evaluation run (Figure 5 d)) spanned 23 virtual machines, the job already finished after approximately 24 minutes. This marks a comparable processing cost, however, at considerably savings in processing time.

VII. Related Work

Performance analysis and bottleneck detection can be conducted on three different levels of abstraction.

The lowest level of abstraction addresses the profiling of distributed applications (VampirTrace, TAU, KOJAK, Paradyn, and others). This covers the instrumentation of code or capturing of messages in order to record events as well as timestamps of events. This generates very detailed insight into the performance of an application but is very difficult to translate into useful knowledge about performance bottlenecks due to the sheer amounts of data produced.

In order to assist the developer at the discovery of bottlenecks, the middle level of abstraction assumes the use of a certain programming paradigm such as Map/Reduce, Master/Worker, pipelined execution, etc. that is enriched with user-contributed code. A framework for the programming paradigm can define measurement points from which metrics can be derived that hint to specific performance issues in the user code or the degree of scale-out [16], [17].

The highest level of abstraction does not require knowledge of a specific programming paradigm but rather considers the parallel application as a whole with a generic performance indicator like e.g. a service response time or other Service Level Objectives of an n-tier system [18], [19], [20].

Espinosa et al. present in [21] a generic classification of causes for performance bottlenecks in parallel execution environments. This comprises slow communication, blocked sender, multiple output problem (one sender communicating with several receivers requires serial sending of message), lack of parallelization, and problems at a barrier. Many of these problems can be addressed at a framework level that realizes a scalable parallel programming paradigm, such that users of the framework do not need to worry about these details. Yet, parallel environments rarely achieve linear scaling such that they need to decide which components to scale out and to which degree they shall be scaled out. This is addressed e.g. by the following papers.

Li and Malony describe in [17] a performance tuning tool that operates in several phases. The parallel program to be tuned is instrumented and profiled by their TAU profiler. The recorded events are then aligned and matched to an abstract description of the communication patterns of the Master/Worker paradigm. Based on this alignment it is possible to create a performance model which defines e.g. the time between receiving a task and sending the result as the computation time. By this, several metrics can be defined (e.g. computation time) and derived (e.g. worker efficiency). These metrics are then passed into a rule system that infers the causes for bad performance and presents them to the user.

The related problem of scheduling pipelined workflows on a static set of compute resources has been thoroughly studied over the past decades. Many approaches in this area consider more restricted workflow topologies such as linear chains instead of DAGs (e.g. [22]). Additionally, the most common optimization objectives are throughput and/or latency given a fixed set of compute resources. In [23] Vydyanathan et al. present a heuristic that uses estimations of processing and transfer times of data items to schedule a DAG-shaped workflow on a fixed set of homogeneous processors in a latency-optimal manner, while satisfying throughput requirements. In contrast to this work, our approach strives to maximize the system utilization on a variable set of compute resources.

Benoit et al. present in [16] a model for pipeline applications in Grids. The pipeline model assumes a set of stages, each of which consists of a data receiving, data processing and data sending phase. Assuming a set of characteristics such as latencies and compute power, the model is capable of suggesting assignments of stages to processors. The model makes several limiting assumptions that make it significantly more restrictive than our approach. Such assumptions are e.g. that each stage processes the same number of tasks. Furthermore it does not discuss the core question addressed in this paper, i.e. detecting bottlenecks in order to infer scaling strategies. Cesar et al. describe in [24] comparable work on modeling the master/worker paradigm and use this to find decisions on the number of workers in a master/worker environment.

Malkowski et al. describe in [25] the detection of multi-bottlenecks. The basic discovery is that n-tier systems can suffer from increasing workload and show increasing response times without having a single permanent bottleneck. The declining performance can instead result from temporary bottlenecks that are either oscillatory or concurrent on different compute nodes. Histograms or kernel density estimation can be used to visualize the distribution of degrees of utilization of resources over time to indicate multi-bottlenecks.

Chanda et al. discuss in [26] how to provide end-to-end profiles of transactions in multi-tier applications. They consider applications in which client requests are processed by a series of different stages. A stage may be a different process, a thread, an event-handler, or a stage worker thread. Through the algorithms and techniques introduced in their paper, the authors are able to track client requests through each of these stages and infer the amount of time the requests spend in them.

Apart from the field of distributed systems, bottleneck detection also plays an important role in other practical areas of computer science. E.g. in [27] Kannan et al. present an approach to detect performance bottlenecks in Multi-Processor System-on-a-Chip environments. Based on the idea of the dynamic critical path [28], their work aims at identifying components which contribute significantly to the end-to-end computation delay.

VIII. Conclusion

In this paper we presented an approach to detect bottlenecks in parallel DAG-based data flow programs. The algorithm we introduced is capable of detecting CPU as well as I/O bottlenecks and therefore assists developers in finding reasonable scale-outs for their jobs.

We introduced a simple processing model which abstracts from the concrete compute resources and determines bottlenecks solely through the relationship among the tasks. Based on our processing framework Nephele we evaluated different strategies to obtain the task characteristics required by our model at runtime. A first evaluation of our work suggests that already a small number of iterations is sufficient to discover major performance bottlenecks and improve the levels of parallelization for the tasks involved in a processing job.

For future work we can envision adapting the degree of parallelization for the tasks of a processing job dynamically at runtime. This would render the need of multiple iterations of the same job superfluous. Furthermore, it would allow an adaptive job optimization without any prior assumptions about data distributions. Furthermore, the model can be extended by memory bottlenecks and load balancing issues. A sufficiently precise model might allow to determine an optimal degree of scale-out for tasks after a single or very few profiling runs.

References

[1] I. Raicu, I. Foster, and Y. Zhao, "Many-task computing for grids and supercomputers," in *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*, Nov. 2008, pp. 1–11.

[2] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, "DiskReduce: RAID for data-intensive scalable computing," in *PDSW '09: Proceedings of the 4th Annual Workshop on Petascale Data Storage*. New York, NY, USA: ACM, 2009, pp. 6–10.

[3] The Apache Software Foundation, "Map/Reduce tutorial," http://hadoop.apache.org/common/docs/current/mapred_tutorial.html, 2010.

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2007 EuroSys Conference*, 2007, pp. 59–72.

[5] "Cascading," http://www.cascading.org/.

[6] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/PACTs: A programming model and execution framework for web-scale analytical processing," in *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*. New York, NY, USA: ACM, 2010, pp. 119–130.

[7] Amazon Web Services LLC, "Amazon Elastic Compute Cloud (Amazon EC2)," http://aws.amazon.com/ec2/, 2009.

[8] Rackspace US, Inc., "The Rackspace Cloud," http://www.rackspacecloud.com/, 2010.

[9] D. Warneke and O. Kao, "Nephele: Efficient parallel data processing in the cloud," in *MTAGS '09: Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*. New York, NY, USA: ACM, 2009, pp. 1–10.

[10] D. Gottfrid, "Self-service, prorated super computing fun!" http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/, 2007.

[11] Amazon Web Services LLC, "Amazon Elastic Block Storage (Amazon EBS)," http://aws.amazon.com/ebs/, 2009.

[12] A. Kivity, "kvm: the Linux virtual machine monitor," in *OLS '07: The 2007 Ottawa Linux Symposium*, July 2007, pp. 225–230.

[13] R. Russell, "virtio: towards a de-facto standard for virtual I/O devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 95–103, 2008.

[14] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus open-source cloud-computing system," in *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 124–131.

[15] "iText home page," http://www.itextpdf.com/, 2010.

[16] A. Benoit, M. Cole, S. Gilmore, and J. Hillston, "Evaluating the performance of skeleton-based high level parallel programs," in *The International Conference on Computational Science (ICCS 2004), Part III, LNCS*. Springer Verlag, 2004, pp. 299–306.

[17] L. Li and A. D. Malony, "Model-based performance diagnosis of master-worker parallel computations," in *Euro-Par*, ser. Lecture Notes in Computer Science, W. E. Nagel, W. V. Walter, and W. Lehner, Eds., vol. 4128. Springer, 2006, pp. 35–46.

[18] G. Jung, G. S. Swint, J. Parekh, C. Pu, and A. Sahai, "Detecting bottleneck in *n*-tier it applications through analysis," in *DSOM*, ser. Lecture Notes in Computer Science, R. State, S. van der Meer, D. O'Sullivan, and T. Pfeifer, Eds., vol. 4269. Springer, 2006, pp. 149–160.

[19] S. Malkowski, M. Hedwig, J. Parekh, C. Pu, and A. Sahai, "Bottleneck detection using statistical intervention analysis," in *DSOM'07: Proceedings of the Distributed systems: operations and management 18th IFIP/IEEE international conference on Managing virtualization of networks and services*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 122–134.

[20] S. Malkowski, M. Hedwig, D. Jayasinghe, C. Pu, and D. Neumann, "CloudXplor: a tool for configuration planning in clouds based on empirical data," in *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2010, pp. 391–398.

[21] A. Espinosa, T. Margalef, and E. Luque, "Automatic performance evaluation of parallel programs," in *Parallel and Distributed Processing, 1998. PDP '98. Proceedings of the Sixth Euromicro Workshop on*, Jan. 1998, pp. 43–49.

[22] J. Subhlok and G. Vondran, "Optimal latency-throughput tradeoffs for data parallel pipelines," in *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 1996, pp. 62–71.

[23] N. Vydyanathan, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz, "A duplication based algorithm for optimizing latency under throughput constraints for streaming workflows," in *ICPP '08, the 37th International Conference Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 254–261.

[24] E. Cesar, J. Mesa, J. Sorribes, and E. Luque, "Modeling master-worker applications in POETRIES," in *High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. Ninth International Workshop on*, Apr. 2004, pp. 22–30.

[25] S. Malkowski, M. Hedwig, and C. Pu, "Experimental evaluation of N-tier systems: Observation and analysis of multi-bottlenecks," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct. 2009, pp. 118–127.

[26] A. Chanda, A. L. Cox, and W. Zwaenepoel, "Whodunit: transactional profiling for multi-tier applications," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. New York, NY, USA: ACM, 2007, pp. 17–30.

[27] H. Kannan, M. Budiu, J. D. Davis, and G. Venkataramani, "Tuning SoCs using the global dynamic critical path," in *SOC Conference, 2009. SOCC 2009. IEEE International*, 2009, pp. 407 –411.

[28] G. Venkataramani, M. Budiu, T. Chelcea, and S. C. Goldstein, "Global critical path: a tool for system-level timing analysis," in *DAC '07: Proceedings of the 44th annual Design Automation Conference*. New York, NY, USA: ACM, 2007, pp. 783–786.