Evaluating Approaches to Detect Bottlenecks in the Pipe & Filter Framework TeeTime

Bachelor’s Thesis

Adrian Pegler

March 30, 2016

KIEL UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
SOFTWARE ENGINEERING GROUP

Advised by: Prof. Dr. Wilhelm Hasselbring
M.Sc. Christian Wulf
Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,


ii
Abstract

Bottleneck detection is a wide field, vital for increasing the performance of software. For several decades, scientists all around the world have published different approaches to find bottlenecks. So far, nobody seems to have thought about combining different approaches.

In this thesis we show situations that need a combination of approaches to accurately detect the major bottleneck of a system. Therefore, we study an example application built with TeeTime, a Pipe & Filter framework for Java. The following two aspects are special about this case: Firstly, it has a complex execution scheduling, and secondly, the bottleneck occurred during a version update. Thus, we do not only need to find any restrictive component, but the specific one that led to the performance decrease in the version 2.0 compared to 1.1.2. We introduce and discuss frequently-used approaches and show how it is possible to combine them in one single representation. From this, we draw conclusions by considering different approaches at once.

We evaluate our combined approach by measuring and comparing the activity states of the application's components and by this show the insufficiency of using only a single approach for this situation. We also conclude which component led to the performance loss of version 2.0 and introduce a solution. The solution will be evaluated to ensure its effectiveness. This is done by measuring the new activity states of the application's components and the total execution time. To show that the solution not only increases the performance of version 2.0, but actually restores the performance lost during the update, we compare it both with the previous one and with the performance of version 1.1.2.
1 Introduction 1
  1.1 Motivation 1
  1.2 Goals 2
  1.3 Document Structure 3

2 Foundations and Technologies 5
  2.1 The Pipe & Filter Architectural Style 5
      2.1.1 Filters 6
      2.1.2 Pipes 6
  2.2 The Pipe & Filter Framework TeeTime 7
      2.2.1 Overview 7
      2.2.2 TeeTime Stages 7
      2.2.3 TeeTime Pipes 9
      2.2.4 The WordCounter Application 10
  2.3 Bottleneck Detection Approaches 12
      2.3.1 The Term Bottleneck 12
      2.3.2 A Classification of Bottleneck Detection Approaches 13

3 Examples for Bottleneck Detection Approaches 15
  3.1 Possible Scenarios for TeeTime 15
      3.1.1 Execution on Different Numbers of Threads 15
  3.2 Coefficients for TeeTime 16
      3.2.1 Utilization 16
      3.2.2 Waiting Time or Queue Length 17
      3.2.3 Blocking, Starvation and the Turning Point 17
      3.2.4 General Prerequisites 18

4 An Extended and Combined Approach 21
  4.1 Challenges of TeeTime 21
      4.1.1 An Extended Approach: New State Introduced 21
      4.1.2 A Combined Approach: Only a Matter of Representation 21
  4.2 Logging the States 22
      4.2.1 The Locations of State Changes 22
      4.2.2 The Storage of the Data 24
      4.2.3 The StateLogger 24
## Contents

5 Applying our Approach to TeeTime 27  
5.1 Scenarios ........................................ 27  
5.2 Experimental Setup ............................ 27  
5.3 Results & Discussion ............................ 28  
5.4 Threats to Validity .............................. 31  

6 A Solution for the TeeTime Bottleneck 33  
6.1 The Culpable Code and a Solution .......... 33  
6.2 Evaluation of the Solution .................... 34  
   6.2.1 Scenarios and Experimental Setup ...... 34  
   6.2.2 Results & Discussion ..................... 34  
   6.2.3 Threats to Validity ....................... 34  

7 Conclusions and Future Work 37  
7.1 Conclusions ..................................... 37  
7.2 Future Work .................................... 37  

Bibliography 39
Chapter 1

Introduction

1.1 Motivation

Studies show the reaction of end users to performance decreases of website applications.\footnote{http://blog.kissmetrics.com/loading-time/} Amazon for example experiences a 1% decrease in sales for each additional 100 ms delay in response time, and Google reports 20% drop in traffic due to 500 ms delay in response time [Ibidunmoye et al. 2015]. These are only two examples for the importance of high performance for end user acceptance. This relation applies not only to web applications but to software in general.

Before one can put the performance of a software to the test, one needs to specify what performance actually means. Often it is characterized in terms of the duration taken to perform a given set of tasks or the rate at which these tasks are performed with respect to the amount of system resources consumed within a time interval [Gregg 2013]. The subject of this thesis is a type of software with a special perspective on performance: frameworks. Since any overhead induced by the framework will decrease the performance of any software built with it, the creators need to take appropriate care of the performance of their framework, which means to minimize such an overhead [Waller and Hasselbring 2012].

In particular we investigate a performance decrease in an application built with TeeTime, a framework for modelling high-throughput Pipe & Filter applications in Java. Comparing release 1.1.2 and 2.0, we measure a significant increase in the execution time of the test application WordCounter (see Figure 1.1). The exact functioning will be explained in Section 2.2.4. As seen in Figure 1.2 the execution time of version 2.0 is nearly 15% higher than the one of version 1.1.2. The figure shows the average of 5 executions each.

In this thesis we illustrate a way to search for the cause of the particular bottleneck leading to the performance loss of the WordCounter application of TeeTime 2.0. This serves as a case study for bottleneck detection in frameworks in general. Additionally, we focus on special situations caused by the effective use of concurrency.
1. Introduction

![Diagram of WordCounter configuration]

**Figure 1.1.** Test configuration: WordCounter. The inner components represent filters. Connecting pipes are shown as arrows, and include the data type that is transported. Filters that are included in one box run on the same thread.

![Execution time comparison graph]

**Figure 1.2.** Execution time differences of the word counter configuration (see Figure 1.1) for the versions 1.1.2 and 2.0.

1.2 Goals

a. **Explain examples for approaches to detect bottlenecks**
   We introduce and explain existing approaches to detect performance bottlenecks and discuss their advantages and drawbacks in general, but also in our particular situation (compare: Chapter 3). An overview and a classification of these approaches are given in Section 2.3.

b. **Apply bottleneck detection approaches to the WordCounter**
   After that, we apply the approaches to the WordCounter application to find the bottleneck that caused the performance decrease of version 2.0 compared to 1.1.2. In particular, we introduce a combined and extended approach based on the results of the previous discussion. Additionally, we show that each single approach might fail in our situation and the detection can only be achieved by combining the approaches.

c. **Introduce a solution**
   The bottleneck that is identified by our approach is solved.
Finally, we evaluate the impact of the solutions on the performance of TeeTime. We show that our solution restores the performance of version 1.1.2 on version 2.0.

1.3 Document Structure

First, we introduce all necessary foundations for our work in Chapter 2. In particular we familiarize with the Pipe & Filter architectural style and the framework TeeTime. Subsequently, we give an overview on bottleneck detection approaches, respectively a coarse grain classification.

Chapter 3 gives several selected examples of bottleneck detection approaches, including a discussion on their practicability for our specific situation.

Taking the results from the previous chapter, we then introduce and explain our own approach in Chapter 4. We show what changes are necessary to fit the special needs of TeeTime, and introduce a way to combine or switch between the previously mentioned approaches with ease.

In Chapter 5 we discuss the collected data and show which part of the framework causes the previously mentioned execution time overhead (compare Figure 1.2).

Finally, we summarize our work, draw final conclusions and point out possible future work in Chapter 7.
Chapter 2

Foundations and Technologies

2.1 The Pipe & Filter Architectural Style

Pipe & Filter is an architectural style that provides a structure for systems that process streams of data. The whole process is disaggregated into its single processing steps which are each encapsulated in one separate unit called filter. These disjoint units are connected by pipes through which the data streams are passed. The sequence of filters connected by pipes is called a processing pipeline. Figure 2.1 shows example pipelines with different complexity. This pattern provides flexibility by allowing reordering, recombination and targeted replacement of units. This is further supported by a fine division of processing steps, which makes it easier to reuse them in different systems. Furthermore the processing steps are well separated which reduces synchronization to a bare minimum within the pipes. This supports genuine concurrency. [Schmidt et al. 2013]

With growing possibilities for storing data and the need of processing them, Pipe & Filter gained an increased popularity both in industry and in research since it potentially combines high modularity with a high throughput and small memory usage. [Wulf and

![Diagram of simple example pipeline.](image1)

![Diagram of complex example pipeline.](image2)

![Diagram of hierarchical filter.](image3)

(a) Simple example pipeline. [van Bergen 1999]

(b) Complex example pipeline, featuring: Multiple sources and sinks, branching data flow and a feedback loop. [van Bergen 1999]

(c) Hierarchical filter. [van Bergen 1999]

Figure 2.1. Example pipelines illustrating different aspects of the Pipe & Filter architectural style.

The term pump is used as a synonym for data source.
2. Foundations and Technologies

Hasselbring 2015]

2.1.1 Filters

Filters are the processing units of the system. The name exemplifies that most of these units already produce output incrementally while they still consume input and do their work, just like a liquid that passes through a filter. Thereby a low latency and a real parallelism are achievable. The filters may enrich the data by computing and adding information, refine them by concentrating or extracting information, or transform the data into other representations. Every filter can contain every one of these three possible basic principles. [Schmidt et al. 2013]

Only adjacent filters share information, namely their access to the same pipe. So the only methods that need synchronization are the pushes and pulls at each single pipe. By keeping the need for synchronization this low, concurrency is supported. A question likewise concerning concurrency is the point of time when a filter is doing its work. Therefore, filters are divided into active and passive. Active filters are commonly looping and computing without further activation while pulling input data, respectively pushing output data. Passive filters on the other hand are activated by adjacent pipeline elements either by pulling or pushing data from or to the passive filter. [Schmidt et al. 2013]

In Section 2.2 we will describe how TeeTime accomplishes this.

Special filters are the data source and data sink. The data source provides the input to the system while the data sink collects it. Several different sources and sinks are possible in one single system. As shown in Figure 2.1 b, a pipeline can contain an arbitrary number of sources and sinks. [Schmidt et al. 2013]

Hierarchical filters as illustrated in Figure 2.1 c are a way to reduce complexity in large processing pipelines and to keep track of relations between filters. [Schmidt et al. 2013]

2.1.2 Pipes

Pipes interconnect the filters. As explained before, this ensures keeping the need for synchronization to a minimum. The implementation of pipes can vary greatly depending on the system’s environment, for example a distributed network system or a single shared memory system [Wulf and Hasselbring 2015]. But despite these differences, pipes often need to implement some first-in-first-out buffer to allow the storage of data in between two filters. [Schmidt et al. 2013]

If the activity of a filter is controlled by one of the adjacent filters buffers may not be necessary. While some consider to implement the pipes in such a case as direct method calls, this would compromise re-usability and flexibility in general. [Schmidt et al. 2013]

Nevertheless, such thoughts are important when dealing with a high-performance framework. In Section 2.2.3 we will show how TeeTime deals with this difficulty. TeeTime’s solution, however, leads to further obstacles for the bottleneck detection approaches, as we will see later in Chapter 3.
2.2. The Pipe & Filter Framework TeeTime

2.2.1 Overview

TeeTime is a high-throughput Pipe & Filter framework for Java. It provides abstractions to model filters – called stages – and to model processing pipelines. It brings many ready-to-use stages and features an execution model that imposes minimal to no overhead due to the abstractions. [Wulf and Hasselbring 2015]

Figure 2.2 gives an overview of the general associations of the framework’s key components. The stages and pipes are explained in more detail in Section 2.2.2 and Section 2.2.3. Ports are a way to specify the connection between filters and pipes and grant several benefits (see: Section 2.2.2).

As introduced by Abowd et al. [1993] TeeTime uses a configuration as an additional key component (see: Figure 2.2). These describe a collection of filters and their connection via pipes as well as the active status of filters which enables execution (see: Section 2.2.2). [Wulf and Hasselbring 2015]

2.2.2 TeeTime Stages

Filters in TeeTime are called stages. The framework provides all features that are illustrated by Figure 2.1. To allow multiple connections and feedback loops, stages can have an arbitrary number of input and output ports. These ports are typed so that the connecting pipes can ensure that the items pushed by an output port are suitable for the connected input port. CompositStages enable the use of hierarchy within the pipelines. Additionally, this concept is used for the execution of the system. [Wulf and Hasselbring 2015]

Letting every stage be executed by a separate thread might lead to a notable overhead due to context switches. In order to fit modern computer architecture TeeTime provides a possibility for multi-threaded execution while keeping the number of threads variable. Therefore, the stages are passive by default to allow activation by other stages instead of needing a separate thread. To define the number of threads used, any stage can be set active in the configuration. The framework then collects all non-active successor stages and
2. Foundations and Technologies

Figure 2.3. Method calls of any consumer stage. The data source will only skip the receive call.

composes them. This special CompositStage also implements the Java interface Runnable to allow execution by a thread and is therefore called a RunnableStage. Since passive stages in TeeTime are only activated by pushes of previous stages, the data sources of the system always need to be active, because they do not have any predecessor stages. This approach yields further possibilities to obliterate overhead induced by now unnecessary synchronization. Therefore, pipes are chosen according to the placement in the pipeline (compare: Section 2.2.3). [Wulf and Hasselbring 2015].

In Figure 2.3, we see which methods are called in the execution of a ConsumerStage. First, the execution of the stage is triggered. If this stage is the first or the only stage inside the RunnableStage the method executeStage() is directly called by the thread. If it has some prior stages within the RunnableStage, this is done by pipeA (compare: Section 2.2.3, Figure 2.5). Afterwards, a new element is received from one or more of the input ports. For better readability only one pull is depicted, even though an arbitrary number of pulls can be conducted before continuing the actual execution. If the pipe is empty, it will return a null element. If computation cannot be done without a proper element, the stage needs to handle that situation. Most of the stages used in the WordCounter application only have one input port and therefore just cancel their current execution with calling returnNoElement() (compare Figure 4.1) and wait to be triggered again. Of course, this might not be possible for multi-port stages that already pulled data from some pipes. It should be mentioned that this case only occurs for the first stage on a thread. (For more information see Section 2.2.3.) After the computation is done for the received element(s), the result is sent to one or more of the output ports, which transfer it to the next pipe. In case this is not the last or only stage summarized in the thread, this might include triggering the next stage (compare: Section 2.2.3, Figure 2.5).
2.2. The Pipe & Filter Framework TeeTime

A data source, called ProducerStage, has a similar structure but does not provide input ports and therefore cannot pull elements. As a consequence, the method structure is kept more simple and all the work is done within the execute() method without the invocation of a separate execute(e).

2.2.3 TeeTime Pipes

Pipes allow the connection of stages. Like ports, pipes are typed, thus allowing only suitable ports to be connected and thereby granting type safety.

As mentioned before, TeeTime allows active and passive stages to grant the possibility to balance benefits and drawbacks of multi-threaded execution. As a consequence, some pipes connect stages within a single thread, while others need to connect stages running on different threads. To allow two threads to access data, one to push and the other pull the data, the pipe needs to implement some kind of buffer. Therefore TeeTime uses a SpScArrayQueue with bounded capacity to store the data. This makes the method calls rather simple as we can see in Figure 2.4; however, these calls need to be synchronized. While pipes connecting stages within a single thread could be implemented the same way, TeeTime uses a different pipe to eliminate overhead induced by the unnecessary synchronization. These intra-thread pipes only store the reference to a single element and then activate the subsequent stage. This is illustrated in Figure 2.5. This approach ensures back-pressure, meaning that every item is pushed as far as possible through the pipeline as the item is processed. The appropriate pipe is chosen automatically by the framework, as mentioned in Section 2.2.2. [Wulf and Hasselbring 2015]

Figure 2.4. Method calls for pipes between two adjacent threads.
Due to these differences in pipe implementation, only some stages can actually be blocked in certain situations. While intra-thread pipes cannot lead to the blocking of a stage, the first and last stage within a runnable stage can be blocked. This occurs either when trying to pull an element from an empty pipe or when trying to push an element to a full pipe.

While an individual blocking strategy can be configured, TeeTime uses a busy-waiting strategy by default. This optimizes high-throughput execution and prevents high frequency of blocking and unblocking of hardware threads. Thus, the so-called blocking of a stage does not necessarily correlate to a blocking of the executing thread. [Wulf and Hasselbring 2015]

Pipes connecting multiple producers or multiple consumers are not supported. This would involve logic to decide how elements are distributed or collected. Nevertheless, distribution to multiple consumers or collection from multiple sources is an indispensable component in certain systems. Therefore, TeeTime features the Distributor and Merger stages. These are 1-to-n, respectively n-to-1 stages, with configurable strategies to spread or collect the elements from or to the adjacent pipes. [Wulf and Hasselbring 2015]

### 2.2.4 The WordCounter Application

The WordCounter application is a test pipeline included in TeeTime which revealed an increase of execution time of TeeTime 2.0 compared to TeeTime 1.1.2. The application counts the number of occurrences for each word in a text file which results in a hash map.
2.2. The Pipe & Filter Framework TeeTime

Figure 2.6. The test configuration WordCounter including activity of stages (small squares), Runnable stages (outer rectangles) and stages contained in the composite stage WordCounter.

containing the words and their count. The WordCounter pipeline includes many features that we described above. Figure 2.6 gives an overview of its structure.

The pipeline has a single data source represented by an InitialElementProducer, which provides a file. After the file is provided by the InitialElementProducer, the File2SeqOfWords stage pre-dissects it into chunks of whole words and up to 512 characters total. These are distributed to a configurable number of WordCounter stages. This distribution is done by the Distributor which uses a round robin strategy. While the InitialElementProducer is only executed once at the very beginning, the two other stages within the first thread alternate in execution and provide a continuous stream of word sequences until the whole file is read.

To exploit multi-core systems, the WordCounter pipeline provides an arbitrary number of WordCounter stages. These are composite stages which consist of a Tokenizer, a ToLowerCase and a MappingCounter each. The Tokenizer further dissects the provided sequence of words into strings containing exactly one word. In order to count words as identical independently from capitalization, the ToLowerCase stage is included before counting. The MappingCounter finally stores each word and its specific count into a hash map. The WordCounter stage can be seen as a subsystem with the Tokenizer as the data source and the MappingCounter as the sink.

A peculiarity of the MappingCounter and therefore of the WordCounter stage subsystem
2. Foundations and Technologies

is that the resulting map will not be delivered until the whole file is finished. Thus, the
Merger and CountingMapMerger that form the last RunnableStage will only execute at the
very end of the total execution. They merge all previous counts and words into one big
hash map.

2.3 Bottleneck Detection Approaches

2.3.1 The Term Bottleneck

We would like to define what a so called bottleneck is before we discuss approaches to
detect them. Although the general idea is simply a restrictive component in a system,
unfortunately, the term bottleneck does not have a uniform definition. Some definitions
often found in literature are:

a) If a machine has the smallest isolated production rate, this machine is the bottleneck.
The production rate is defined as the average number of parts produced by a machine
per cycle of time. [Kuo et al. 1996]
b) If the work-in-process inventory in a buffer is the largest, then the machine right after
this buffer is the bottleneck. [Kuo et al. 1996; Lawrence and Buss 1995]
c) If the production capacity of a machine is minimum relative to demand this machine is
the bottleneck. [Lawrence and Buss 1995]
d) If a machine has the maximum ratio of overall system throughput increment to its own
standalone throughput increment, then this machine is the bottleneck. [Li et al. 2007]

Here, a machine is the collective term for a part of a system that does any kind of work.
For production lines in a factory, this could for example be actual machines, as well as
employees or automated vehicles. [Lawrence and Buss 1995; Kuo et al. 1996; Li et al. 2007;
Roser et al. 2001]

To argue about the feasibility of a definition for the term bottleneck it is necessary to
know that there may be different types. In some systems there is exactly one single bottleneck
while in others multiple bottlenecks can be identified. Furthermore different bottlenecks might
be identified at different times, which are therefore called shifting bottlenecks. [Ibidunmoye
et al. 2015]

In the definition examples above we observe that the definition of the bottleneck is
often formulated in respect to the data that is collected for its detection. As for definition
a by Kuo et al. [1996], one should measure the production rate, or for definition b found
in Lawrence and Buss [1995], one should measure the buffer inventory and define the
bottleneck in respect to fit this data. The most general one at first sight is definition d from
Li et al. [2007]. It does not restrict or give a direct hint on how the performance increment is
to be measured. Yet it strongly correlates to the scenario-based class of approaches to detect
2.3. Bottleneck Detection Approaches

bottlenecks, also introduced by Li et al. [2007] (see: Section 2.3.2). Additionally, it only identifies a bottleneck that can be eliminated as the solution is included in the definition. Furthermore, it is very strict in the way that only one component is called bottleneck, so it neglects multiple and shifting bottlenecks.

The definition we want to use in this thesis is a rather intuitive one, yet covering all so far mentioned drawbacks. Gregg [2013] defines: A bottleneck is a resource or component that limits the performance of a system. This definition is not restricted to one specific approach and covers all three types of bottlenecks. Furthermore, it illustrates the term in a way that is understandable even for someone who is not familiar with the topic.

2.3.2 A Classification of Bottleneck Detection Approaches

Static Approaches

One class of approaches is to statically investigate the system’s behaviour. Static means that the system itself is not changed throughout the investigations.

One possibility to gain insight on the systems behaviour without actually touching it, is by changing input data. This is an often-used method in validation and verification of software. However, at least one drawback is that it would only give a rather coarse grained overview of the system’s behaviour. [Roser et al. 2001]

Another approach introduced by Cox III and Spencer [1997] is a structure analysis of the system. According to Roser et al. [2001] this is a complex and difficult manual task. Yet a statical system analysis that was necessary for the specific situation of the WordCounter was a diff on the code to identify which passages in the code actually changed (compare: Chapter 6).

Scenario-Based Approaches

An approach introduced by Li et al. [2007] is the scenario-based approach. A specific configuration of the studied system is called a scenario. After identifying critical components of the system it is simulated or actually executed with changing scenarios. Finally, the impacts of the scenarios on the item are compared and the underlying configuration of the one with the greatest positive effect is called the bottleneck. [Li et al. 2007]

While of course the corresponding bottleneck definition given by Li et al. [2007] is fitted to this kind of approach our definition by Gregg [2013] is still feasible as obviously a restrictive component was found and solved.

Coefficient-Based Approaches

A more diverse and sensitive approach is to search for bottlenecks based on simulation output data, called coefficients. Examples for such coefficients are the following:
2. Foundations and Technologies

- **Utilization** [Roser et al. 2001; Lima et al. 2008] is a common coefficient that measures the percentage of time a machine is active. The component with the highest percentage of active time is considered to be least influenced by others, respectively influences others the most, and thus is likely to be the limiting factor for the system.

- **Waiting Time** [Roser et al. 2001; Lima et al. 2008] calculates the cumulative time other components wait for availability of each component. The component producing the greatest waiting time is considered the bottleneck.

- **Shifting-Bottleneck Method** [Roser et al. 2001] is a coefficient that measures active time similar to the utilization approach. But additionally, it also observes the overlap of the active period of a component with the adjacent components. This represents the shift of the bottleneck. Both perspectives are considered to detect bottlenecks.

  The necessary data can be acquired by either externally monitoring the system, or simply by logging the system’s work. [Roser et al. 2001]
3.1 Possible Scenarios for TeeTime

We can identify several scenarios for the WordCounter application. Actually, the configuration itself encourages to change the number of WordCounter stages used for counting, which defines one sort of scenarios for the application. Other scenarios considering different I/O workloads could change the quantity or size of the test file. As described in Section 2.2.3 and Section 2.2.4, some stages like the File2SeqOfWords stage use specific sizes for the sequence of words, and TeeTime’s inter-thread pipes are bounded. Changing these numbers also defines some scenarios. Although the two latter stated types of scenarios might be interesting in general, they are not promising for our specific situation. Neither I/O workload nor buffer or chunk sizes were changed between both versions of TeeTime, yet both versions still differ in execution time. So for now, we will only explicate the first one.

3.1.1 Execution on Different Numbers of Threads

Scenarios

For both versions of TeeTime the WordCounter application was executed with different numbers of WordCounter stages, ranging from 1 to 24. This leads to a different number of threads with a minimum of 3 up to 26.

Experimental Setup

The experimental setup is the same as used in Section 5.2 for our main evaluation. The only difference is that we only conducted two JVM runs because this is only a pre-evaluation and the results were still stable.

Results & Discussion

Figure 3.1 shows the execution time of the WordCounter applications on the y-axis and the number of used WordCounter stages on the x-axis.
3. Examples for Bottleneck Detection Approaches

![Figure 3.1](image-url) Execution time of both versions of the WordCounter application with respect to different quantity of WordCounterStages

<table>
<thead>
<tr>
<th># counting stages</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>v 1.1.2</td>
</tr>
<tr>
<td>1</td>
<td>42.9</td>
</tr>
<tr>
<td>2</td>
<td>22.5</td>
</tr>
<tr>
<td>4</td>
<td>13.8</td>
</tr>
<tr>
<td>6</td>
<td>10.7</td>
</tr>
<tr>
<td>8</td>
<td>10.1</td>
</tr>
<tr>
<td>12</td>
<td>10.5</td>
</tr>
<tr>
<td>16</td>
<td>10.7</td>
</tr>
<tr>
<td>24</td>
<td>12.5</td>
</tr>
</tbody>
</table>

We can see that increasing the number of threads has a great positive impact on the execution time, at first. Yet, exceeding the number of threads the processor can manage at once, decreases this effect again. Although this conclusion is not too stunning, we can still get a hint for the part of the system that might bear the problem. We can see in the diagram as well as in the table of Figure 3.1 that the execution times converge. Thus we may conclude that the culpable code can be found within the WordCounter stages. Otherwise, the extension of execution time would be constant.

Unfortunately, the WordCounter subsystem itself consists of 3 single stages. So this is a first hint but, does not get us far in solving the problem.

Threats to Validity

Threats to validity are also listed in Section 5.4.

3.2 Coefficients for TeeTime

Many coefficients have been discussed in literature (compare: [Roser and Nakano 2015; Roser et al. 2001; Li et al. 2007; Lima et al. 2008; Ibidunmoye et al. 2015]). In the following sections, we explain several examples and argue how well they might be feasible for our purpose. In Section 3.2.4 we point out similarities and common prerequisites to clear the way for our approach that is introduced in Chapter 4.

3.2.1 Utilization

Until today, one of the most frequent approaches is to identify bottlenecks via utilization. Some variations are described by Law and Kelton [1991]. In general, this approach considers the component that has the highest utilization to be the bottleneck. Utilization can be
measured as the percentage of active time, while sometimes also the longest average cycle time is used. [Roser and Nakano 2015]

Roser et al. [2001] criticise the percentage of active time as inaccurate because different behaviour can still lead to the same percentage of active time. Therefore, they introduce the average active time. Later, Roser and Nakano [2015] show that both approaches might fail to identify the bottleneck when it comes to shifting. Nevertheless, this might be a feasible approach for our situation, because we need to search for differences in two versions of the same software. Therefore a greater active time could help to find the bottleneck. Although the shifting of bottlenecks can not be excluded, it is very unlikely due to the concurrency and constancy of data in the WordCounter application. We will resume arguments in Section 3.2.4.

3.2.2 Waiting Time or Queue Length

Some approaches use the behaviour of buffers or more precisely of items inside the buffers to detect the bottleneck. In literature, some similar approaches were introduced. For example total waiting time [Law and Kelton 1991], average waiting time [Pollett 2000], length of the queue [Lawrence and Buss 1994] or a combination thereof [Elmasry and McCann 2003]. Roser and Nakano [2015] show that these approaches fail in their test applications. Already Roser et al. [2001] argued that especially in systems with bounded buffers this approach will fail due to the tendency for the buffers to reach maximum or minimum level of inventory.

Additionally, it would be more extensive to implement this method than the ones that only log stages. We will see in Section 4.2.1 that logging the state of stages is quite simple. Logging the entry and exit time for each item in each pipe would not only be more memory-expensive, it would also require a different data structure than the one that is used. The data structure would need to have the ability to log the times, which is not supported by the current one. Therefore, we would change the system we want to monitor, likely corrupting the whole result. Furthermore we illustrated the structure of intra-thread pipes which only store one element that is immediately pulled by the next stage. This further disqualifies this approach.

Nevertheless, these approaches highly correlate with the blocking and starvation of components, at least for bounded buffer systems.

3.2.3 Blocking, Starvation and the Turning Point

Approaches like the arrow method [Kuo et al. 1996] and the turning point approach [Li et al. 2009] are based on the blocking and starvation of components. According to Roser and Nakano [2015], both approaches are highly inaccurate. However, a similar approach introduced by Roser et al. [2014] named bottleneck walk has detected the bottlenecks in their test systems quite well. The approach is based on three simple assumptions.
3. Examples for Bottleneck Detection Approaches

1. A component that is waiting, for the time being, is not the bottleneck.

2. When a component is waiting for parts (starved) the bottleneck must be upstream.

3. When a component is unable to deliver due to a full buffer (blocked) the bottleneck must be downstream.

Therefore, by observing the blocking behaviour of the stages, we can know the direction we need to search for the bottleneck. This approach also has the ability to identify multiple bottlenecks. If different points in time are considered, also shifting of bottlenecks might be detected.

Nevertheless, the same argument as before applies here. Due to the single element intra-thread pipes, this approach will only work on a coarse grained level. It can only identify the subsystem(s) containing the bottleneck(s) due to the pipe implementation. We will see this in Section 5.3.

3.2.4 General Prerequisites

In contrast to the static or scenario-based approaches, the coefficient-based approaches need a greater extent of data collected during the execution. Therefore, one could choose to log the system’s behaviour manually or use external monitoring software.

We already mentioned in Section 2.2.2 that the states of TeeTimes stages may not correlate with the states of threads. Due to the busy waiting policy and several stages running on a single thread, these will be active all the time and will never actually be blocked by the system itself. Therefore, we would not benefit from monitoring software in this way. Monitoring software would have to monitor the single methods of each stage to get a clue about the actual behaviour of the stage. Thus, a log of the stage’s work as proposed by Roser et al. [2001] is more feasible for our situation.

So to get a clue of what data we need to store we will make some observations. All previously explained approaches are in some sort of way based on time and state. Excluding waiting time and queue length due to its correlation to blocking, all approaches need the state of a stage and the corresponding timing. Thus, we observe that all coefficients mentioned above could be calculated if we just store a time stamp for every state change for each stage. In Section 4.2.1 we show the exact locations where these state changes occur.

The next similarity is that all approaches divide the behaviour of a component into active and passive or blocked. Most define a component as blocked when it is waiting, either for a new item or for the possibility to deliver the finished item, and as active when it is doing work (compare: [Roser et al. 2001] and [Roser and Nakano 2015]). In TeeTime we additionally have the different kinds of connections. Stages connected to buffered pipes can effectively be blocked due to an empty pipe when pulling, or due to a full pipe when trying to push. A stage connected to a second one on the same thread, on the other hand, also needs to wait for the next stages to finish. But this is actually the wanted behaviour, as we want the finished item to be pushed as far downstream as possible. So as long as we
3.2. Coefficients for TeeTime

want to keep the blocking of a stage as a kind of failure state, we need to consider a third state. This will be introduced in detail in Section 4.1.1.

Furthermore, the approaches can be combined. Instead of only examining the active times of a component like the utilization approach dictates, or the blocking and starvation of components as in the bottleneck walk, we could just represent both at the same time and draw our conclusions with greater confidence. While the utilization may especially point out the bottleneck within an autonomous subsystem, the bottleneck walk might be more feasible in showing the subsystem that contains the greatest bottleneck, or even give a ranking in the perspective of the overall system.

Additionally, we observe that especially the percentage of active time and average active time share a common denominator. Both can be calculated from the cumulative active time. This is why we will use this representation in our approach in Section 5.3 but provide possibilities to also represent the data as percentage or average times (compare: Section 4.2.3).
An Extended and Combined Approach

As we have seen in the previous Chapter 3, we need an approach that also manages TeeTime’s specific states, such as active waiting. Furthermore, we pointed out similarities in some of the coefficients. So it should be possible to easily switch between one or the other representation, as well as to argue about complementing approaches at the same time. Our approach is based on the previously discussed coefficient-based approaches and extends them by the features presented above, as well as combining coefficients that are easily exchangeable or complementing.

4.1 Challenges of TeeTime

4.1.1 An Extended Approach: New State Introduced

The TeeTime framework features an execution model that can summarize an arbitrary number of stages to be executed by one thread (compare: Section 2.2.2). This can be used to prevent a system from switching the context too often, as well as to exploit the possibilities of a system. However, this also leads to special states. Beside the basic active and blocked states we also have to consider a stage waiting for a previous or subsequent stage that is run by the same thread to complete its work. Of course, this stage is not blocked when the thread is actively doing its work and the stage does behave correctly. But we also cannot consider it active, as we have discussed in Section 3.2.4.

Therefore, we introduce the additional state active waiting. This gives us the opportunity to get the real execution times of each stage while still keeping this separate from blocking. Where exactly the state changes occur and what new problems this bears for the implementation will be explained in Section 4.2.1 and solved in Section 4.2.2.

The new list of states and when they are present is shown in Table 4.1.

4.1.2 A Combined Approach: Only a Matter of Representation

The single coefficient-based approaches in Section 3.2 always demand data in a specific representation and then conclude what the bottleneck might be. Some authors like Roser et al. [2001] argue about benefits and drawbacks of those representations and they might have a point (compare: Roser et al. [2001] or Roser and Nakano [2015]). But first and foremost
4. An Extended and Combined Approach

Table 4.1. Summary and short description of possible states.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>The stage is doing productive work.</td>
</tr>
<tr>
<td>Blocked</td>
<td>The stage, together with the whole RunnableStage needs to wait for new input or for the disposal of finished items before productive work can continue.</td>
</tr>
<tr>
<td>Active waiting</td>
<td>The stage is waiting, but another stage on the same thread is doing productive work.</td>
</tr>
</tbody>
</table>

it is only a matter of transforming the data into a specific representation. Therefore, our approach will be more precise about how to acquire the data.

If we consider the data to be present in some sort of log, like a list of pairs of a state and a time stamp when the change of state occurs for each stage, then we have all the data present to compute either the average active time, the percentage of active time or cumulative active time, and respectively blocked or active waiting time. Also, other data can be excavated from this log. It might be interesting to know if one thread was blocked once but for a long time, or many times just for a swift moment.

Therefore, we will follow this specific approach to collect data. Then we can switch between these single representations with ease and pick the one that is best suited for the present specific situation. Furthermore, this approach can be easily extended by new representations.

Additionally, we will represent not only one kind of state at a time, but rather the apportionment of states throughout the execution of each stage. This allows us to consider different approaches at once when reasoning about the location of a bottleneck. This possibility for the use of multiple approaches is the key to our argumentation in Section 5.3.

4.2 Logging the States

In this chapter we will introduce the exact locations in the control flow of a stage when the stage changes states, as well as challenges and solutions for saving them.

4.2.1 The Locations of State Changes

Figure 4.1 shows the points at which a stage can change its state. a denotes the point when the stage starts execution. In dependence to what is returned at b, it is decided at point c whether the actual computation is done. This happens if all necessary ports have delivered a proper element. If no proper element has been received in one of the ports, the stage is considered blocked.
4.2. Logging the States

This might not be the actual state of the thread, as we already mentioned in Section 2.2.2, even though it is true for most stages in the WordCounter application. One exception is the merger (compare: Figure 1.1) which shall count as an example for other stages that need to wait busily in such a case. The merger will try to pull from each input port at least once before it cancels its current execution cycle. Other stages may need to wait for a proper element to continue their execution and it may not be adequate to actually block the thread. In that case, the thread that executes the stage is considered running in the system’s point of view, but the state of the stage is nevertheless considered blocked.

Presupposed that at point c all elements are given to continue the execution, the state of the stage is active. After the computation is done, the element is sent. Now point d denotes the crucial point mentioned in Section 4.1.1 where active waiting might occur. The state of the stage now depends on what pipe is connecting the successor stage. If the next stage is running on the same thread and is therefore connected via an intra-thread pipe, there cannot be any blocking due to the system’s architecture, as mentioned in Section 2.2.3.

Since the stage is not productive in that case either, the sending stage receives the state active waiting. If the pipe is an inter-thread pipe and the pipe can take more elements, the previous stage stays active while alternatively becoming blocked.

Finally, from point e on, the stage is waiting to be executed again and for that time considered actively waiting again. It doesn’t matter if it really waits for some previous stages on the same thread or if it just waits to be triggered by the executing thread itself again. Even in the last case, this stage change makes sense as it stores information about the time the thread, or namely RunnableStage, takes between triggering the executions. Although this information is not needed for the current case, it is easier to implement because we do
4. An Extended and Combined Approach

not need a differentiation whether the stage really has a predecessor or not, but can handle all stages the same.

4.2.2 The Storage of the Data

In Section 4.1.2 we introduced a way to store the data. So in general we save the time when a stage changes its state as well as its new state. This may not seem too difficult but we need to have a closer look here.

In the experimental setup in Section 5.2 we mention the test file with 190,692,500 words. Storing a state each time we get to one of the locations a,c,d,e for every stage of the application, would clutter up the whole memory. So as a first optimization we only save real state changes. Before creating and storing a new state, we check if the last saved state was already the same.

Additionally, as we observe in Figure 4.1, there is effectively not much work done between a and c. So switching to active at a just to be blocked again at c or even b would again be more work and loss of memory space than benefit. Therefore, we only save the time stamp of possible state change at a at first. Attaining c, we decide what state has been present the whole time, then check whether the new state is to be saved. We consider the whole time of receiving the element as productive if the stage can do its work, and unproductive if it cannot. Particularly, we consider the stage blocked if it only tries to receive a new element, but cannot do any productive work afterwards.

Then again the location d, where we might change the state to active waiting, turned out to be crucial. If we saved every state change there and switched back to active again at e if any work needs to be done, this would clutter the memory just too much to be constructive. Therefore, we do not save any real state change, but instead keep considering the stage active while the successor stages do their work. Additionally, we clock the elapsing time and store it cumulatively for each stage. By this, we keep the overhead, both execution time and memory, to a minimum. In spite of losing exact timestamps this is sufficient, especially for the cumulative time representation that we will use in the evaluation (see Chapter 5). With little effort all other representations can still be achieved. The same counts for the time elapsing between e and the next a which clocks the time the stage actively waits for previous stages.

4.2.3 The StateLogger

Figure 4.2 shows the structure of the StateLogger that we implemented to log collect and represent the data. The Stateble interface establishes communication with the stages. We can see that the stages themselves collect their states. Although it may appear better at first sight to have an external monitor that collects all the data without changing the stages, this would lead to complications. First and foremost, it would be inaccurate to have a program that samples the stages. Due to the high throughput and the related short term of method cycles, such sampling would be likely to miss state changes. In
4.2. Logging the States

addition, an external program collecting the data would either need a separate thread or need synchronization to be called by the other threads; in the worst case both. This would be a much greater intervention to the behaviour of the application because of the non-determinism of synchronizing threads and context switches. So we decided to accept the overhead caused by logging the states inside the stages themselves. This overhead is consistent throughout the whole execution time and for both versions. Additionally, with this approach, we can not miss state changes. So this is the most accurate and consistent way to log the stages’ state changes in TeeTime. We made sure to handle these stage change locations at the highest possible abstraction level. By this newly added stages can also be logged without imposing an individual state change handling for each new stage.

The pipes also use the interface to communicate the status of sending the items back to the stage. As we can see in Figure 4.1, the stage sends the finished item via the output port to the pipe. But the stage does not know if the sending itself succeeds or if the pipe is full and therefore the sending is busily waiting in a loop. Therefore, we need this feedback to receive knowledge about the sending behaviour of the stages. The active waiting stop watch is also handled by the single element pipes because there are some stages that overwrite the execute()-method and therefore trigger the sending themselves. So for all these stages we would have to integrate the stopwatch individually.

The StateLogger also features a strategy pattern for easy exchange and extensibility of the data representations. By this we make sure that different approaches can work together to get the most accurate and curtain results. And, as long as our data is sufficient, new not yet considered approaches can easily be integrated.
Applying our Approach to TeeTime

5.1 Scenarios

For the following evaluation we have built and executed the WordCounter application with both versions of TeeTime. The StateLogger itself, presented in Section 4.2.3, is the same in both versions. As mentioned before, we implemented the logging of the state changes inside the stages. Therefore, we needed to add a few lines of code, but we have not touched any functionality. These changes were made as identically as possible for both versions. (Compare Section 5.4 for threats to validity.) Consequently, our approach does change the execution time. Since both versions were changed in the same way, we can assume that this does not impose much inaccuracy. The execution time of both versions and also each duration of states should be elongated by nearly the same amount of time. It confirms our assumption that we were able to find a solution without struggle.

In Section 3.1 we have introduced and discussed some possible scenarios for the WordCounter application. We have already stated that the only promising scenario type is changing the number of threads and we discussed the results in Section 3.1.1. Later in Section 5.3, we observe that the blocking time coefficient will reveal the same information. Thus, we did not change the number of threads for our evaluation. We chose to run the WordCounter application with six WordCounter stages leading to eight threads altogether. This is not the optimal setup according to Figure 3.1. However, by using eight threads altogether, we ensure that each core of the processor used in the experimental setup (compare: Table 5.1) only has to handle one thread of our application. We chose this scenario to minimize random noise imposed by unpredictable context switches.

5.2 Experimental Setup

The benchmark was executed on a multi-core system. The hardware and software details are shown in Table 5.1. Both versions of the WordCounter application get their test data in the form of a huge text file containing 190,692,500 words. We use the Oracle Java Development Kit (JDK) in the version 1.8.0_60, together with the eclipse IDE in the version 4.5.1.

In order to smooth out variations in the measurement, we set the number of warm-up iterations to three and the number of measurement iterations to five. To gain a better
5. Applying our Approach to TeeTime

Table 5.1. Setup used for the evaluation

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel Core i7-4710 HQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>x86-64</td>
</tr>
<tr>
<td>Clock/Core</td>
<td>2.5GHz</td>
</tr>
<tr>
<td>Cores (# of threads)</td>
<td>4 (8)</td>
</tr>
<tr>
<td>RAM</td>
<td>16 GB</td>
</tr>
<tr>
<td>OS</td>
<td>Windows 10</td>
</tr>
</tbody>
</table>

stability we do three Virtual Machine runs. So the presented results are each the average of 15 application runs. Additionally, the six WordCounter stages are averaged for a clearer arrangement. Thus, we gain a very good stability, especially for the three stages that are of the highest interest.

5.3 Results & Discussion

Figure 5.1 shows the apportionment of execution time of the WordCounter application. In figure a we see the results of the test runs of the application built and executed with TeeTime 1.1.2 and in b the data of the one built and executed with TeeTime 2.0. The total time in blue shows the execution time of the whole application, while the stages are each apportioned into active time in green, active waiting time in yellow and blocked time in red. The six single WordCounter stages, each containing a Tokenizer, ToLowerCase and MappingCounter stage, are summarized and averaged into one representative each, as we mentioned in Section 5.2. The stages are ordered on the axis in such a way that the chronological data flow through the stages is bottom to top.

To gain better clarity, figure c shows the differences of the other two charts. This is not requested by any approach, but it is an obvious step as we compare two versions of the software to find the new bottleneck. This figure again shows the total execution time in blue at the top, and then the apportioned differences in each stage staying in a particular state. Positive numbers denote an increased amount of time for TeeTime 2.0 in respect to version 1.1.2, while negative ones mean a decreased amount of time.

We focus on the greatest difference for each stage because small differences could be caused by inevitable disparities in the data collection due to differences in the two TeeTime versions (compare Section 5.4).

The nearly non-existent runtime of the CountingMapMerger arises from the fact that the stage is only triggered when all WordCounter stages have finished counting and the Merger receives their results. The overwhelming blocking time of the Merger is caused by this behaviour, too. According to approaches considering the active time, these stages cannot be the bottleneck. Additionally, the approaches considering the blocking time can give a better clue. As the Merger is the first stage of the collective RunnableStage, we can conclude that the blocking is caused by an empty pipe. All approaches based on blocking and starvation
5.3. Results & Discussion

(a) Cumulative active, active waiting and blocked time of WordCounter application run with TeeTime v1.1.2
(b) Cumulative active, active waiting and blocked time of WordCounter application run with TeeTime v2.0
(c) Difference of the above presented data.

Figure 5.1. Apportionment of execution time of the WordCounter application run on 6 threads with one test file.
5. Applying our Approach to TeeTime

conclude that the bottleneck needs to be searched upstream.

Starting chronologically at the bottom of the charts, we see the first three stages running together on one thread (compare Figure 1.1). We observe the paramount increase of active waiting time for the InitialElementProducer and the File2SeqOfWords. This means that both stages need to wait longer for the Distributer to do its work in TeeTime 2.0 than in TeeTime 1.1.2.

The essentially higher blocked time of the Distributer shows what they are all waiting for. This time, the stage that has a high cumulative blocking time is the last stage of the collective RunnableStage. Consequently the blocking must be caused by pushes to full pipes. Again, we can draw a conclusion from the blocking-based approaches as all of them state that the bottleneck is to be searched downstream. Thus, the WordCounter stages seem to contain the bottleneck. We need to investigate them further to localize the bottleneck.

Similar to the conclusion above, the elongated active waiting time of the Tokenizer and the ToLowerCase stages are hints to look even further rearwards, leaving only the Mappingcounter, which also shows the highest increase in active time. By all we have deduced so far, we need to consider the MappingCounter to be the system’s bottleneck.

By investigating Figure 5.1, we can identify the situations in which single approaches used alone could have trouble to detect the actual bottleneck. Considering the worst case, they could fail completely to identify the actual bottleneck. In sub-figure (a) we can see that the blocking-time-based approaches would identify the WordCounter stages as the bottleneck-containing component. However, they fail to identify a single stage by only being able to point out a whole subsystem.

The active-time-based approaches, on the other hand, would conclude that the File2SeqOfWords stage is the bottleneck, as it has the highest overall active time. This totally neglects the conclusions drawn by the blocking-based approaches. Increasing the performance of the File2SeqOfWords stage would most likely, first and foremost, result in an even greater blocked time of the Distributer and therefore would not have a positive impact on the system’s overall performance.

We need the blocking-time-based approaches to identify the subsystem that has the highest potential of containing the system’s major bottleneck. And we also need the active-time-based approaches to identify the actual restrictive stage within that subsystem. Thus, only by combining both approaches we are able to draw the right conclusion to find the system’s major bottleneck.

So we have seen that our approach allows to use different approaches and combine the results in order to identify the major bottleneck. However, knowing that the MappingCounter is the bottleneck is not yet enough to solve it. Therefore, we will use a static approach to find the exact location of the bottleneck within the stage. The results and a solution are presented in the next Chapter 6.
5.4 Threats to Validity

Internal

Small differences in the two versions of TecTime could impose some inaccuracy on the data. To filter out this inaccuracy, more specific measurements and observation would be needed. In the next chapter we introduce a solution. The data measured for the evaluation of this solution seems to confirm this inaccuracy (compare: Figure 6.1 and Section 6.2.3). However, this could also be a real difference in behaviour. Neither can be concluded from the data we collected so far.

External

The benchmark was only evaluated for a very coarse grained scenario. Diversifying the number and sort of scenarios would increase the external validity. Moreover, we evaluated our task farm on one specific multi-core system and a particular JVM and OS only. Other systems and different software might perform differently. To get stable results it might also be necessary to change the values for VM, warm up and measurement runs.
Chapter 6

A Solution for the TeeTime Bottleneck

6.1 The Culpable Code and a Solution

The last chapter revealed the MappingCounters as the system’s bottleneck. Unfortunately, this is not sufficient for a solution. To find the effective change from 1.1.2 to 2.0 that is responsible for this, we needed to take a look at the program code. This is included in the static approaches. We operated a manual code diff on both versions of the CountingMap. The result was that there was no difference apart from the absence of an explicitly written default constructor.

A more accurate review showed that there was only one method call within the execute(e) method. Therefore, the conclusion obtrudes that we need to search within that method call. It was a call to the CountingMap. By operating a manual diff on the two versions of this class we found one single difference. To store the count of each word a third-party data structure is used, namely an implementation of the ObjectIntMap interface provided by Carrot-Search-Labs [2015]. While TeeTime 1.1.2 uses the ObjectIntOpenHashMap from hppc version 0.6.1, TeeTime 2.0 uses the newer ObjectIntHashMap from hppc version 0.7.1.

As this was the only notable difference, it seems likely that this data structure performs slower than the older one. In the following chapter we will evaluate the impact on TeeTime’s performance when changing TeeTime 2.0 to re-use the older data structure.

So far we have seen that the coefficient-based approaches we combined before were not accurate enough to find a solution for the bottleneck. Therefore, we also needed to perform the static analyses above. As argued by Roser et al. [2001], performing such a task for the whole application would be a humongous task, even for a small and only slightly complex pipeline like WordCounter. By narrowing down the possible component capable of containing the bottleneck, our coefficient-based approach from the previous chapter literally enabled us to use such a static approach. We record that, again, a combination of approaches is needed to locate and solve a bottleneck.
6. A Solution for the TeeTime Bottleneck

6.2 Evaluation of the Solution

6.2.1 Scenarios and Experimental Setup

In order to achieve meaningful results, we will consider the same scenarios as explained in Section 5.1. The setup is the same as in Section 5.2. The hardware and software details can be found in Table 5.1.

6.2.2 Results & Discussion

Figure 6.1 shows the execution time apportionment of the WordCounter application. In figure a we see the data of the application built and run with TeeTime 1.1.2. Figure b shows the data of the one built and run with TeeTime 2.0, but this time using the older data structure. The total time is the execution time of the whole application, while the stages are each apportioned into active time in green, active waiting time in yellow and blocked time in red. The six WordCounter stages, containing a Tokenizer, a ToLowerCase and a MappingCounter stage each, are summarized and averaged in one representative. The stages are ordered on the axis in such a way that the chronological flow through the stages is bottom to top.

For a better focus on the differences figure c shows the differences of the other two charts. This again shows the total execution time overhead at the top and then the apportioned differences in each stage staying in a particular state. Positive numbers denote an extension in time for TeeTime 2.0 in respect to version 1.1.2 while negative ones mean a smaller amount of time.

We can still see some differences, but the main goal – to improve the overall execution time – is achieved. In Figure 6.1 c we can see that now the newer version of TeeTime runs almost as fast as the old one. The blue bar right at the top showing the execution time overhead is barely existent.

6.2.3 Threats to Validity

Internal

Differences of the stages between the two versions are now often a redistribution of active and active waiting time. This might be real behaviour, but could also be caused by imprecision of the timestamps used by Java or by small disparities in the data collection due to differences in the two versions of TeeTime.

External

As already stated in Section 5.4, the validity is threatened by the narrow selection of used systems, software and scenarios. This could be improved by diversifying those aspects.
6.2. Evaluation of the Solution

(a) Cumulative active, active waiting and blocked time of WordCounter application run with TeeTime v1.1.2

(b) Cumulative active, active waiting and blocked time of WordCounter application run with TeeTime v1.1.2

(c) Difference of the above presented data.

**Figure 6.1.** Apportionment of execution time of the WordCounter application run on 6 threads with one test file.
Chapter 7

Conclusions and Future Work

7.1 Conclusions

We have introduced general classes of bottleneck detection approaches and discussed examples for most of them. By doing so, we have outlined their feasibility for the specific needs of TeeTime. Furthermore, we have observed similarities and common prerequisites in most of the approaches. These led to our own extended and combined approach which gave good insight into the application’s behaviour and enabled us to detect the stage that carried the bottleneck. The actual change that led to the performance loss, then was further localized by a static code analysis.

Thus, we see that sometimes more then one approach is needed to find the exact location of bottlenecks. The coefficient-based approaches combined in our approach can provide a rather coarse grain hint for the location of the system’s major bottleneck, as well as pointing out the actual stage that acts as the bottleneck within a subsystem. But an additional static analysis of the culpable stage might be necessary to find the exact location. It is likely that any individual approach instead of a combination would have had trouble pointing out the exact location of the specific bottleneck.

We have found that the third party data structure, that was used to store the count for each word, was culpable for the longer execution time of the WordCounter application that was build with TeeTime 2.0. Simply using the older structure eliminated the problem completely.

7.2 Future Work

The StateLogger can be used as a regression benchmark incorporated into the continuous integration of TeeTime. Thereby, it could help to keep the standard as high as possible and to detect bottleneck-related anomalies as early as possible. Therefore, significant test applications would be needed. Until now, we only searched a specific bottleneck responsible for a specific elongation of execution time. The StateLogger should also be used to find general bottlenecks in the applications build with the TeeTime framework to further optimize its performance.

So far, the overhead of logging the states is always included when running any applications containing our StateLogger. Therefore, a way is needed to effectively switch the state
7. Conclusions and Future Work

logging on or off. Further optimization to minimize the overhead on execution time and memory could also be identified.

Additional representations or new approaches can be integrated into the StateLogger’s formatting strategies. By this, the StateLogger might become even more accurate in finding the bottlenecks. So far, the shifting of bottlenecks is not included in the StateLogger. It works with averaged data of the whole execution and thereby could miss the actual bottleneck (compare: Roser and Nakano [2015]). A method to keep the intervention of the StateLogger with the System to a minimum while attaining intermediary results, would enable the StateLogger to also detect shifting bottlenecks.
Bibliography


Bibliography


