On Improving the Performance of Pipe-and-Filter Architectures by Adding Support for Self-Adaptive Task Farms

Master’s Thesis

Christian Claus Wiechmann

November 16, 2015

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,
Abstract

The focus of the Pipe & Filter architectural style in software development is the improvement of modularity in software systems. Since all functionality is implemented in filters, the system is clearly split into separate components by architectural design. Oftentimes, we can parallelize such components to significantly improve the performance of the Pipe & Filter architecture.

In this thesis, we design a self-adaptive Task Farm for Pipe & Filter architectures. The Task Farm includes a number of user-defined filters which are parallelized by creating new instances of them. The workload is then distributed evenly between the filter instances. The Task Farm automatically adapts to the underlying hardware as well as the concrete implementation of the parallelized filters. Therefore, it measures its current throughput and parallelizes its included filters until the underlying hardware is optimally utilized. Hence, we maximize the performance of the Pipe & Filter architecture. We show the structure and the behavior of the Task Farm and all its components in a detailed way.

We also perform a performance evaluation to analyze the behavior of the Task Farm and its speedup compared to single-threaded execution of the Pipe & Filter architecture. For that purpose, we employ four different multi-core processor architectures and four different workload scenarios to ensure that the evaluation is as representative as possible. For the evaluation, we provide a Task Farm implementation for the Pipe & Filter framework TeeTime. We compare the computation times between the test systems and test scenarios, showing that with suitable configuration parameters, the Task Farm provides a significant speedup in all cases. Furthermore, we show that in most cases the Task Farm does not decrease the total computation time compared to single-threaded execution, even if we use suboptimal parameters.
Contents

4.2.3 Reconfiguration Component .................................. 44
4.3 Monitoring Behavior ............................................. 45

5 Performance Evaluation ............................................. 47
  5.1 Methodology ....................................................... 47
  5.2 Experimental Setup .............................................. 48
    5.2.1 Environment .................................................. 48
    5.2.2 Test Scenarios ................................................. 50
  5.3 Results and Discussion ......................................... 54
    5.3.1 Overview of the Computation Times ...................... 54
    5.3.2 Detailed Analysis using the INTEL System ............... 64
  5.4 Threats to Validity .............................................. 74

6 Related Work ..................................................... 77

7 Conclusions and Future Work ................................... 79
  7.1 Conclusions ..................................................... 79
  7.2 Future Work .................................................... 79

Bibliography .......................................................... 81
Chapter 1

Introduction

1.1 Motivation

The Pipe & Filter architectural style in software development [Taylor et al. 2009] helps improving modularity and performance [Wulf and Hasselbring 2015] of software systems. Conforming to this style, any program is broken down into filters, which contain the application logic, and pipes, which handle the communication between these filters. Pipes transfer data and are generally unidirectional. For example, a calculation of a sum of two input elements with the sum as an output element could be a valid filter.

To provide a more efficient development using this style, the Pipe & Filter framework TeeTime [Wulf and Hasselbring 2015] for the Java programming language can be used. TeeTime enables developers to model pipes and filters (from here on: stages) using Java.

Since TeeTime focuses on the performance of the execution, the topic of parallelization becomes relevant. It would be useful if specifically labeled stages are automatically parallelized in a way that resources are optimally used. Not only would parallelization improve performance, but it would also enable legacy systems to be parallelized inside TeeTime, provided that used stages are synchronized correctly to avoid concurrency errors.

Therefore, we introduce a construct to dynamically parallelize a stage in TeeTime to improve the total computation time. Such a construct is called a Task Farm [Aldinucci et al. 2011; Cole 2004; 1991; Aldinucci and Danelutto 1999]. Another important feature of the Task Farm is to evenly distribute incoming elements from previous stages of the Pipe & Filter architecture to the parallelized stages. After all, an even distribution of the total workload is necessary for optimal parallelization. Finally, after the parallelized stages finish the execution of some elements, the elements have to be merged to a common element output stream of the Task Farm.

Furthermore, to optimally utilize the underlying hardware, a self-adaptive behavior [Suleman et al. 2010; van Hoorn 2014; Gonzalez-Velez 2006] of the Task Farm is necessary. A self-adaptive Task Farm parallelizes its stages in such a way that the computation time is minimized for the underlying hardware.

The goal of this Master’s Thesis is the implementation of a TeeTime stage following the Task Farm pattern. It parallelizes other stages without any need to manually introduce parallelization patterns to the Pipe & Filter architecture. This parallelization approach is provided in a self-adaptive way, i.e., the Task Farm assigns additional CPU resources by
1. Introduction

itself to maximize the efficiency of computations.

1.2 Goals

1.2.1 G1: Integration of the Self-Adaptive Task Farm Parallelization Pattern in TeeTime

G1.1: Design of a Self-Adaptive Task Farm Stage

One or more stages inside a Task Farm are parallelized depending on its throughput and the available CPU resources. The required monitoring infrastructure must be minimal invasive and maximal efficient. To design the Task Farm, it is necessary to first define the structure of it and then define the behavior of its components.

G1.2: Implementation of the Task Farm Stage in TeeTime

In this master’s thesis, an implementation of the Task Farm is provided for TeeTime. The constraints of this task are as follows. Firstly, there are no feedback loops included in any TeeTime Pipe & Filter architecture. Secondly, multiple stages to be parallelized inside the Task Farm are handled as a single composite stage. Lastly, we assume that the stages are correctly synchronized to avoid erroneous concurrent execution.

1.2.2 G2: Performance Evaluation

After implementing and testing the aforementioned requirements in TeeTime, we verify performance improvements by means of a benchmark. We realize several different scenarios as Pipe & Filter architectures to achieve a sophisticated evaluation. The different scenarios are described in the following list.

CPU-Intensive Workload The CPU-intensive test case focuses on the usage of CPU resources inside the Pipe & Filter architecture. We implement two types of CPU-intensive workloads: balanced and unbalanced. Using a balanced approach, the parallelized stages always compute the exact same workload for each element. In contrast, an unbalanced workload results in a different computation time for each element.

I/O-Intensive Workload The Task Farm does not assign any new I/O resources by parallelization. Therefore, it is useful to analyze the parallelization behavior of the Task Farm as well as the impact on the total computation time using a mainly I/O focused workload.

Combined CPU- and I/O-Intensive Workload Most real-world applications do not entirely depend on either I/O or CPU workload. The combined CPU and I/O test case shows the performance of the Task Farm with a combined CPU and I/O usage of the Pipe & Filter architecture.
1.3. Document Structure

Furthermore, we evaluate how the Task Farm behaves on different test systems. Therefore, we execute the aforementioned test scenarios on multiple systems with different processor architectures.

1.3 Document Structure

This thesis is structured as follows. At first, Chapter 2 introduces the foundations and technologies we use for the design and evaluation of the Task Farm. Afterwards, we design the structure of the Task Farm and its components in Chapter 3. Additionally, we introduce the configurable parameters that have to be set for the Task Farm to reach optimal performance. In Chapter 4, we then discuss the behavior of all components introduced in the previous sections. In Chapter 5, we evaluate an implementation of the Task Farm in TeeTime. Using four different test scenarios and four different test systems, we show the internal behavior of the Task Farm in detail. In Chapter 6, we discuss related work of this thesis. Lastly, Chapter 7 shows the conclusions of this thesis and discusses possible future work.
Foundations and Technologies

In this section, we describe some basic knowledge about foundations and technologies for the rest of the thesis. At first, the Pipe & Filter architectural style is introduced in Section 2.1. Afterwards, we discuss the Task Farm parallelization pattern in Section 2.2. We then introduce the Pipe & Filter framework TeeTime in Section 2.3. At last, we discuss the topic of self-adaptive thread management in Section 2.4.

2.1 The Pipe & Filter Architectural Style

Pipe & Filter architectures, as described by Taylor et al. [2009], possess generally three types of components. The first type is the filter, which includes a well-defined computational task. The second type is the pipe, which unidirectionally connects the different filters with each other. The pipes are connected to the filters via ports. Using filters, pipes, and ports, elements traverse the defined Pipe & Filter architecture and are consecutively executed by the filters. The pipes of a Pipe & Filter architecture can generally contain multiple elements in a buffered queue. The elements inside the queue were already executed by the previous filter, but not yet by the following filter.

An example for a Pipe & Filter architecture is illustrated in Figure 2.1. Rectangles represent filter with their ports, while arrows indicate pipes. In the example, fib generates a number of Fibonacci numbers. Those numbers are then duplicated by the copy filter to its two output pipes. Finally, the Fibonacci numbers are printed to a console (println filter).
2. Foundations and Technologies

and saved to the hard drive (save filter). On most systems, println and save can be executed concurrently, since they do not depend on each other at all. Furthermore, the other filters can run concurrently if, for example, copy works on numbers that fib already created before. Therefore, the execution of a filter does not need to wait until previous filters are finished themselves, providing an excellent opportunity for parallelization.

Another example for a Pipe & Filter architecture are the standard UNIX pipes. UNIX uses the character | as pipes, while filters are represented by applications communicating via standard input and output streams. For instance, let us observe the command ls | grep .log | xargs rm. ls lists all files and directories of the working directory. grep .log then selects all lines containing the string .log, i.e., all files with the .log file extension. Finally, xargs rm removes all those selected files.

We can observe certain properties of pipes and filters. The throughput of a pipe is defined by the number of elements the following filter pulled from the pipe in a certain time interval. Subsequently, we define the throughput of a filter with a single input pipe as the throughput of the input pipe.

The word filter, if used to describe a computational unit for the Pipe & Filter architectural style, is ambiguous. After all, a filter can also be understood as a component suppressing certain input elements. Therefore, we use the word stage instead of filter from now on.

2.2 The Task Farm Parallelization Pattern

Cole [1991] introduced co-called Algorithmic Skeletons. A skeleton is defined as a higher-order function describing a certain computational behavior. The Farm pattern represents such a skeleton and has been defined by, for example, Aldinucci and Danelutto [1999]. Semantically, it describes the identity function of its underlying algorithm. However, the input data stream is parallelized such that the underlying algorithms is concurrently executed. In [Aldinucci et al. 2011], the Farm pattern was, among other patterns, then implemented for the FastFlow framework for Pipe & Filter architectures for stages with one input port and one output port only [FastFlow].

Figure 2.2 shows illustrations of three varieties of the Farm pattern. In Figure 2.2a, a Farm pattern uses multiple consecutive algorithms after one another in a pipeline. We call these algorithms worker stages. In the figure, they are depicted by (W). (D) is an input stream distributor, which distributes input elements to worker stages in order to achieve parallelization. In contrast, (M) merges the computed elements to a single output stream. As we can see in Figure 2.2a, the number of different algorithms parallelized in a single Farm is not limited.

While Figure 2.2a shows the most basic use of a Farm pattern, varieties also exist. Figure 2.2b shows how the Farm pattern supports circular dependency. The merger output stream is here used as the distributor input stream. Another variation is illustrated by Figure 2.2c. We see that we can even use subordinated Farm patterns as a worker stage of another Farm pattern. Additionally, we notice that the worker stages direct their output
2.3. The Pipe & Filter Framework TeeTime

TeeTime, as introduced by Wulf and Hasselbring [2015], is a Java framework for application development utilizing the Pipe & Filter architectural style. For this purpose, it implements stages, pipes, and ports as first-class entities and enables the framework user to design own Pipe & Filter architectures. However, TeeTime currently does not support circular dependencies inside its Pipe & Filter architectures.

While the framework itself already offers a selection of preset stages, pipes, and ports, the user can customize them and add own functionality. In particular, it is very useful to implement custom stages to independently define functionality for the own use case. All
2. Foundations and Technologies

stages can be categorized into two types. The *producer* stages do not have any input ports and are generally used to generate data elements. In contrast, *consumer* stages need data from their input ports to make progress in their execution.

The communication management feature of the concurrent computation of stages is a key of *TeeTime*. It distinguishes between intra-thread and inter-thread communication. Intra-thread communication represents pipe-based communication between stages executed within the same thread. Therefore, it is not necessary to synchronize communication here. After all, only one thread accesses the connecting pipes. However, inter-thread communication is more complex. It represents the pipe-based communication of stages executed within different threads. Therefore, the pipe communication has to be suitably synchronized to avoid concurrency errors. Because of the synchronization, inter-thread pipes introduce a higher communication overhead than intra-thread pipes.

![Diagram of TeeTime Pipe & Filter architecture with mapping of stages to threads](image)

Figure 2.3. *TeeTime Pipe & Filter* architecture with mapping of stages to threads (the beginning of a new thread is represented by a gray background color)

We need to discuss some aspects of the automatic thread management of *TeeTime* to understand the design decisions of the *Task Farm* in later chapters. The automatic thread management of *TeeTime* is designed as follows. The framework user decides which stages should be executed in its own thread. Figure 2.3 illustrates the behavior of the *TeeTime* automatic thread management. Suppose that the framework user defined the stages A, C, and E as the starting point of a new threads (gray). All directly following stages that are not a starting point themselves are then also executed in the same thread. Therefore, if A starts thread 1, B and D are also executed in thread 1. C and E however are executed in their own threads since they do not have following stages that are not a starting point of a new thread.

However, *TeeTime's* automatic thread management also harbors limitations. Let us suppose that stage E is not a starting point for a new thread, as illustrated in Figure 2.4. It
2.4. Self-Adaptive Thread Management

A **Task Farm** implementation would profit immensely by including a self-adaptive thread management. After all, self-adaptive behavior would ensure that the in the **Task Farm** included stages are optimally parallelized, regardless of the underlying platform.

Self-adaptive thread management represents the automatic addition and removal of threads to optimally parallelize a software system. For example, Suleman et al. [2010] introduces an approach to generate a significant speedup using self-adaptive parallelization on a pipeline architecture. Suleman’s approach analyzes the performance of the parallelized software system by gathering runtime measurements. Afterwards, the approach decides if a thread is to be removed or another thread is added based on the runtime measurements. The goal is to parallelize as much as necessary to optimize performance while not running unnecessary threads. After all, unnecessary threads lead to performance overhead as well as a higher power consumption.

For general software systems, the **SLAstic** self-adaptation approach was introduced by van Hoorn [2014]. Figure 2.5 illustrates the different steps required by **SLAstic** [van Hoorn et al. 2009].

For creating self-adaptive behavior in software systems, we require, like Suleman et al. [2010], performance monitoring of the relevant system behavior. For example, we might
2. Foundations and Technologies

![Diagram showing the components of the SLastic self-adaptation approach](image)

**Figure 2.5.** Components of the SLastic self-adaptation approach [van Hoorn 2014]

want to monitor throughputs, reaction times, or capacities, in case they are a bottleneck we want to mitigate by parallelization. The monitoring can either be implemented specifically for the monitored software system to ensure a high efficiency, or a monitoring framework like Kieker [Kieker] can be used for more features and less implementation effort. As the second step, the measurements for the adaptable software system are analyzed. We decide the concrete adaptation behavior, for example the addition or the removal of a worker thread, in this step. Lastly, we enter the reconfiguration step. Now the state of the software system is changed according to the decisions made in the analysis step. After the reconfiguration of the software system is finished, we revert back to the monitoring step.

For the analysis step, a possible approach is using the monitored data for a time series analysis [Box and Jenkins 1976]. At first, we create a prediction for the next measurement. Afterwards, after we have measured the actual value, we compare the actual value to the prediction. Therefore, we deduce the future performance behavior of the software system by analyzing older measurements. Depending on the degree of increase or decrease of performance, we can then decide to add to or remove threads from the software system. For different behavior of the software system we have to use different prediction algorithms to achieve accurate prediction: we distinguish between, for example, linear, exponential, and irregular behavior. Furthermore, some algorithms are more computationally costly than others. A selection of prediction algorithms was introduced by Bielefeld [2012].
Chapter 3

Structure of the Task Farm Stage

3.1 Overview

Our goal is to design a Task Farm Stage utilizing the Task Farm parallelization pattern coupled with self-adapting parallelization behavior, as discussed in Sections 2.2 and 2.4. To make this goal feasible, we define the following assumptions:

1. We do not allow feedback loops for the Pipe & Filter architecture utilizing the Task Farm Stage. This eliminates some types of Task Farm patterns shown in Section 2.2. For example, the output ports of any worker stage inside the Task Farm Stage cannot lead to the Distributor of the same Task Farm Stage. This restriction removes some complexity and enables us to measure the pipe throughput values more accurately for better self-adaptive behavior later.

2. We only allow exactly one input port and one output port each for the worker stages. While it is feasible to implement more ports per worker stage, at this point we restrict ourselves in this way for more clarity. This topic is discussed more thoroughly later in Section 7.2.

Based on these assumptions, we now model a Task Farm Stage. Figure 3.1 shows a corresponding overview. Our Task Farm Stage consists of a Distributor (D in the model), which passes any input element to one of its output ports, with regard to an arbitrary distribution strategy. A valid strategy could be, for instance, a Round-robin implementation [Wulf and Hasselbring 2015]. Each output port of the distributor leads to a unique instance of a Duplicable Stage, which can include any kind of computation. Regardless of which output port has been chosen by the Distributor, the computational result of the Task Farm Stage does not change. The self-adaptive behavior of the Task Farm Stage adds new instances of the Duplicable Stage to improve performance, as well as removes unused instances to reduce computational overhead. The Merger (M in the model) collects all output elements of every instance of the Duplicable Stage and passes them to its output port according to an arbitrary merging strategy. The Merger and the Distributor are introduced in Section 2.2.

Additionally, we design an Adaptation Thread, which controls the number of instances of the Duplicable Stage within the Task Farm Stage at runtime. It measures the throughput of its Task Farm Stage continuously and calculates whether an additional instance of a Duplicable Stage would increase the total performance or not. More worker stages do, however, not
3. Structure of the Task Farm Stage

necessarily improve performance. The total performance gain entirely depends on the type of computation done inside the Duplicable Stage and on as the used hardware. For example, I/O tasks might profit less from parallelization than pure computations. Then again, that might not be true for some RAID systems to a degree, which can have a much higher I/O performance.

In the following chapters, we discuss the structure of the Task Farm Stage in detail. In Section 3.2, we describe the basic components of the Task Farm Stage, including the Distributor, the Merger, and the Duplicable Stage. In Section 3.3, we discuss the Adaptation Thread and all its components. Additionally, for evaluation purposes, a general monitoring component for the Task Farm Stage as a whole is desirable. The structure of the monitoring component is discussed in Section 3.4.

3.2 Task Farm Stage

In Figure 3.2, a UML class diagram of the Task Farm Stage is shown. As explained earlier, it consists of a Distributor, Merger, Adaptation Thread, and one or more instances of a Duplicable Stage. It also provides parameters which allow to optimize the Task Farm Stage and the Adaptation Thread for the underlying hardware configuration.

Even though the Task Farm Stage consists of multiple stages itself, it is used like any other stage in the pipeline. This encapsulates the whole concept of the Task Farm Stage rather well. Moreover, the usage becomes less complicated as if the Task Farm Stage required
3.2. Task Farm Stage

![Class diagram of the Task Farm Stage as well as its configuration](image)

any special implementation effort by the framework user.

The Task Farm Stage requires three type arguments. The first one, \( I \), refers to the type of the input elements. \( I \) also infers the type of the Distributor and has to be the same as the input type of the Duplicable Stage. Similar to this, the parameter \( O \) refers to the type of output elements and to the type of the Merger. Moreover, it represents the output type of the Duplicable Stage. The third type parameter \( T \) refers to the concrete type of the Duplicable Stage.

Another class shown in Figure 3.2 is TaskFarmConfiguration. It represents a container for all user-relevant parameters controlling the behavior of the Task Farm Stage or the Adaptation Thread. The parameters have the following meaning:

- **adaptationWaitingTimeMillis**: The Adaptation Thread continuously tries to improve the performance of the Task Farm Stage by measuring its throughput. However, if the waiting time between each check is set too low and, consequently, the Adaptation Thread is called too often, the generated overhead is too high and decreases the performance of the Task Farm Stage. adaptationWaitingTimeMillis sets the waiting time between each check in milliseconds. Its default value is 50 milliseconds, which, as experience shows, seems to be a reasonable value for many applications. If the performance of the Duplicable Stage often varies, it might be useful to reduce adaptationWaitingTimeMillis to allow the Adaptation Thread to react faster to changes.
3. Structure of the Task Farm Stage

**analysisWindow**  This value defines the amount of recent measurements, which the Adaptation Thread uses to decide if it should create a new instance of the Duplicable Stage. The default value is 5. While a higher value gives a more stable behavior since it deals better with performance peaks, it also causes the Task Farm Stage to react slower to changes.

**throughputAlgorithm**  The Throughput Algorithm defines the algorithm used to analyze the measured throughput values. Possible values are Mean Algorithm, Weighted Algorithm, and Regression Algorithm. The standard configuration is Regression Algorithm. A more detailed explanation for all Throughput Algorithms is given in Section 4.2.2.

**weightedAlgorithmMethod**  This parameter is only used if Weighted Algorithm is used as the Throughput Algorithm. It can be exponential, linear, or logarithmic. The standard value is exponential. The Weighted Algorithm and its parameters are further explained in Section 4.2.2.

**maxSamplesUntilRemove**  The Adaptation Thread has to wait a couple of Adaptation Thread measurements before making any decisions regarding the addition or removal of instances of the Duplicable Stage. This accounts for the fact that, at the earliest, the measured throughput can only increase after the computation of another element is finished by each instance of the Duplicable Stage. The default value is 5. Further explanations on why this value is necessary are given in Section 3.3.3 and Section 4.2.3.

**throughputScoreBoundary**  The Throughput Score is a value between −1 and 1 describing how much the measured throughput of the Task Farm Stage differs from the expected throughput. A value closer to −1 means a performance decrease, while a value closer to 1 means a performance increase. A Throughput Score of 0 shows that the current performance is as expected. The current throughput is measured (see Section 3.3.1), while the expected value is calculated using a Throughput Algorithm (see Section 3.3.2). The Throughput Boundary determines how high the Throughput Score has to be after a new instance of the Duplicable Stage is added to the Task Farm Stage. Only if the Throughput Boundary is exceeded, the Adaptation Thread parallelizes further. The Throughput Boundary is set to a default value of 0.2, but different types of systems or Duplicable Stages might require other values for optimal performance. More of the theory behind the Throughput Boundary is explained in Section 4.2.3, while the evaluation in Chapter 5 shows the behavior of the Task Farm Stage for different boundaries.

**pipeCapacity**  The pipes between Distributor and Duplicable Stage, as well as the pipes between Duplicable Stage and Merger have a bounded buffer for elements. If the buffer is full, no further elements can be added to that pipe as long as the consuming stage does not remove any elements from it. The default value for the pipe capacity is 100. A higher pipe capacity might reduce performance overhead by lessening the amount of necessary communication between stages. It also reduces threading context switches in some cases. However, a high pipe capacity also reduces the impact of parallelization if the total number of elements is low. If all elements are already spread by the Distributor
3.2. Task Farm Stage

... to existing input pipes of different Duplicable Stages, adding another stage would be useless since no element would reach it.

**maxNumberOfCores** This value defines the maximum number of CPU cores the parallelization of the Task Farm Stage may use. Therefore, it directly translates to the maximum number of instances of Duplicable Stage. Each instance of the Duplicable Stage uses its own thread for computations. The Distributor and Merger also require one thread each. The Adaptation Thread represents another thread. However, since it waits for most of the time (see adaptationWaitingTimeMillis), it does not require a dedicated core. That means, the most reasonable value in this case is the number of (virtual) processors minus two, which is also the default value. If we, for example, use multiple Task Farm Stages in one pipeline, we might want to reduce maxNumberOfCores for each Task Farm Stage so that they do not interfere with each other.

**monitoringEnabled** Since monitoring adds additional performance overhead to the Adaptation Thread, it is normally deactivated. If monitoringEnabled is set to true, the monitoring services are enabled for this Task Farm Stage. The monitoring component, as is described in Section 3.4, acts independently to the Adaptation Thread. As such, deactivating this property does not deactivate the adaptation functionality. In fact, as it reduces the computational monitoring overhead, deactivating it improves the performance of the Task Farm Stage.

Thus, each configuration value already has a default value, which performs reasonably well for many cases. But if the user wants to have the highest possible performance, adjustments have to be made in most cases. The evaluation in Chapter 5 empirically shows the best configuration settings for some systems.

### 3.2.1 Dynamic Distributor

In the Task Farm Stage, the Distributor has the task not only to fairly distribute all elements to the instances of the Duplicable Stage. It also has to dynamically create and remove new ports and pipes to change the amount of worker stages of the Task Farm Stage. These requirements exceed those of a normal Distributor, so we call the Distributor of the Task Farm Stage the Dynamic Distributor from now on. A UML class diagram in Figure 3.3 shows an overview of the Dynamic Distributor.

As we can see in the diagram, the Dynamic Distributor inherits from the Distributor and adds its new functionality of dynamically adding and removing ports. The Distributor can only distribute elements to its preset output ports using a distribution strategy. Different distribution strategies can lead to different performance results. Our goal is to improve the performance of the Task Farm Stage as much as possible. We introduce the following distribution strategies:

**CloneStrategy** This strategy creates a unique copy of every incoming element for each output port. After a copy of an element is created, it is passed to its corresponding output port.
3. Structure of the Task Farm Stage

Figure 3.3: Class diagram of the Dynamic Distributor
3.2. Task Farm Stage

Since the goal of the Task Farm Stage is to increase performance by parallelization, this distribution strategy is useless for our purposes. After all, we would not only distribute each incoming element, but also create even more elements and thus increasing the workload, which contradicts our goals.

**CopyByReferenceStrategy**  This strategy passes the reference of every incoming element to each output port. CopyByReferenceStrategy strategy also cannot be used for the Task Farm Stage, since it would also increase the total workload.

**BlockingRoundRobinStrategy**  This strategy first selects an output port by cycling through all available output ports by using a Round-robin pattern. It then distributes an input element to the chosen output port. For the next element, this distribution strategy, again, chooses another output port, and so on. While this is useful for us in theory, it uses a blocking send for each element to transfer it to the output port. Therefore, if the pipe connected to the currently selected port is full, the BlockingRoundRobinStrategy wastes time by waiting for the pipe to accept another element. Depending on the connected pipe, waiting might result in a lower performance, since the following worker stages might have no elements to work on. After all, the following worker stages do not receive any more input elements while the distribution strategy waits for a single pipe.

**NonBlockingRoundRobinStrategy**  This strategy behaves like the BlockingRoundRobinStrategy. However, if the pipe connected to the selected output port is full, it tries another pipe (in Round-robin order) instead. For our use case, this strategy is the most useful, since it gives us the best performance of all presented distribution strategies.

The functionality to dynamically add new ports can be approached in two ways:

1. We could add a second input port to the Dynamic Distributor. If it contains a CREATE-element, we would create a port. If it contains a REMOVE-element, we would remove a port.

2. We could also use a synchronized list of actions as another attribute for the Dynamic Distributor. If another thread adds an action to this list, the action can create or remove a port before another element gets processed.

The first approach has downsides related to automatic thread management in some frameworks, for example in TeeTime (see Section 2.3). In TeeTime, a separate Distributor action port would result in a constraint for the Dynamic Distributor to only possess one input port. Otherwise, the framework would not be able to assign the Distributor to a thread. After all, we could not decide if the Distributor should be executed in the same thread as the stage connected to the input port or in the thread of the stage connected to the action port. While the Distributor of our Task Farm Stage always possesses exactly one input port, in the future we might want to extend the Task Farm Stage to support multiple
3. Structure of the Task Farm Stage

input (and output) ports (see Section 7.2). Therefore, the first approach would hinder us in the future.

Thus, we decide to take the second approach. The CREATE- or REMOVE-actions added to a list in the Dynamic Distributor are called Port Actions henceforth. In Figure 3.3, we can see all available port actions:

**CreatePortAction**  This Port Action will instruct the Dynamic Distributor to create a new port and pipe to an instance of the Duplicable Stage.

**RemovePortAction**  This Port Action will instruct the Dynamic Distributor to remove an output port.

**DoNothingPortAction**  This Port Action has no effect on the Dynamic Distributor. It is mostly used for testing purposes.

The Dynamic Distributor provides the method `addPortActionRequest(PortAction)` to add one of these Port Actions to its list. Every time an element is processed in the Dynamic Distributor, one Port Action is executed previously.

### 3.2.2 Dynamic Merger

Since the Merger, similar to the Distributor, needs to be able to dynamically create and remove ports, we call it Dynamic Merger from now on. In Figure 3.4 a UML class diagram of the Dynamic Merger and its related classes is shown.

As we can see, the structure of the Dynamic Merger is very similar to that of the Dynamic Distributor. For example, it uses certain strategies to determine how to merge elements from its input ports. The following strategies are available:

**RoundRobinStrategy**  This strategy works similar to its counterpart of the Dynamic Distributor. It passes incoming elements in Round-robin order to the output port.

**BusyWaitingRoundRobinStrategy**  This strategy behaves similar to the RoundRobinStrategy, but it also checks if the currently checked input port is already closed, which can happen if the connected instance of Duplicable Stage has been removed. Since removing input ports is part of the desired functionality of the Task Farm Stage and the Dynamic Merger, this strategy is most useful for our purposes.

In terms of adding or removing ports, the Dynamic Merger works the same way as the Dynamic Distributor discussed in Section 3.2.1. Therefore, we can also use Port Actions to implement that functionality here. As can be seen in Figure 3.4, we also implement the CreatePortAction, RemovePortAction, and DoNothingPortAction for this case. The functionality of these Port Actions, as well as the method `addPortActionRequest(PortAction)` to add them to the Dynamic Merger, is the same the one of the Dynamic Distributor.
Figure 3.4. Class diagram of the Dynamic Merger
3. Structure of the Task Farm Stage

3.2.3 Duplicable Stage

A Duplicable Stage requires to implement the interface ITaskFarmDuplicable, which is illustrated by Figure 3.5.

```
<<Interface>>
ITaskFarmDuplicable<I, O>
+duplicate() : ITaskFarmDuplicable<I, O>
+getInputPort() : InputPort<I>
+getOutputPort() : OutputPort<O>
```

Figure 3.5. The Duplicable interface, allowing to create new instances of a Duplicable Stage

Similarly to the definition of the Task Farm Stage, ITaskFarmDuplicable requires the input and output type of the elements passing it. Those types have to conform to the input and output types of the Task Farm Stage; otherwise the Distributor or the Merger would be declared with an incompatible type.

Generally, for the Task Farm Stage it is not necessary or particularly useful if the Duplicable Stage is a single stage. It can also be another Composite Stage, i.e. a number of stages combined to one (just like the Task Farm Stage). Theoretically, it is even possible to use another Task Farm Stage as a Duplicable Stage, but since the enclosed Task Farm Stage would be duplicated many times, the performance overhead and interference would be too high to make it useful.

ITaskFarmDuplicable requires three methods to be implemented. The methods getInputPort() and getOutputPort() have to return the single input and output port of the Duplicable Stage, respectively. Oftentimes, the stages implementing ITaskFarmDuplicable already implement methods to return their input and output ports themselves. In that case, the interface can be implemented by delegation. The last and most important method to be implemented is duplicate(). This method gets called if the Adaptation Thread decides to create another instance of the Duplicable Stage. The framework user may implement this method freely to achieve the desired functionality. The least complex implementation of this method is probably executing a single call to the constructor of the Duplicable Stage and then returning the created instance.

Theoretically, the specification of the duplicate() method is sufficient for the framework user to create any desired behavior when duplicating the stage. However, another approach is to define some annotations, which then could be used to declare a stage as a Duplicable Stage. This topic is discussed in more detail in Section 7.2.

3.3 Adaptation Thread

The Adaptation Thread consists of three components: the History Component, the Analysis Component, and the Reconfiguration Component. An overview of these services is given in
3.3. Adaptation Thread

As discussed in Section 3.2, for every Task Farm Stage there is exactly one Adaptation Thread. It becomes active every few milliseconds (see adaptationWaitingTimeMillis in Section 3.2). Every time the Adaptation Thread becomes active, its components are executed.

The first component is the History Component. It records the current throughput of the instances of the Duplicable Stage inside the Task Farm Stage. This data is then analyzed by the Analysis Component to determine how much of a performance improvement compared to past measurements has been achieved. At last, the Reconfiguration Component uses the results of the previous analysis. It decides if adding a new instance might improve performance or if we should remove a currently used instance to reduce threading overhead.

In the following chapters we discuss the structure of each component. They are, like in Figure 3.6, color-coded in every diagram: any object belonging to the History Component has a red background color, the Analysis Component has a green color, while the Reconfiguration Component is illustrated in orange.

### 3.3.1 History Component

The first component executed in each cycle of the Adaptation Thread is the History Component. Figure 3.7 shows a UML class diagram of the History Component as well as its related classes.

The History Component is the least complex component of the Adaptation Thread in terms of functionality. It only consists of the following three classes:
3. Structure of the Task Farm Stage

![Figure 3.7. Overview of the History Component in the context of the whole Adaptation Thread](image)

**ThroughputEntry** This is a container class for the measurement data. It consists of the total throughput of all instances of the Duplicate Stage at a certain time of measurement.

**ThroughputHistory** The ThroughputHistory contains all throughput entries recorded. It also provides methods to access the measurements for other components like the Analysis Component.

**TaskFarmHistoryService** This class is a facade to provide a less complex way to call the ThroughputHistory for every instance of the Duplicate Stage.

The behavior of the History Component is further discussed in Section 4.2.1.

### 3.3.2 Analysis Component

The Analysis Component has more complex architecture than the History Component. In Figure 3.8, a UML class diagram of the Analysis Component is shown.
3.3. Adaptation Thread

Figure 3.8. Overview of the Analysis Component in the context of the whole Adaptation Thread
3. Structure of the Task Farm Stage

As mentioned in Section 3.2, the Analysis Component contains three algorithms: the Mean Algorithm, the Weighted Algorithm, and the Regression Algorithm. The exact behavior of each of these algorithms is explained in detail in Section 4.2.2. Derived from the AbstractThroughputAlgorithm class, they compute a forecast of the last measurements of the History Component and compare it with the current measurement via the doAnalysis() method. We call this computation result the Throughput Score, which is a floating-point number between $-1$ and 1. 0 means the performance has not improved, while 1 means the performance has improved very much. Consequently, $-1$ represents a very high performance decrease. Using the Throughput Score, we can check how much the performance has increased or decreased after we added or, respectively, removed an instance of the Duplicable Stage. The window attribute corresponds to the analysisWindow defined in Section 3.2.

The TaskFarmAnalyzer class is, like the TaskFarmHistoryService of the History Component, a facade for easier access to the algorithms. The getThroughputScore() method provides access to the last analysis result for the Reconfiguration Component, while the analyze() method starts a new analysis using the given measurement history.

While the Mean Algorithm, Weighted Algorithm, and Regression Algorithm probably are a large enough selection of algorithms and can be used for most use cases, the framework user can also define own algorithms. The throughputAlgorithm configuration explained in Section 3.2 corresponds to the used algorithm such that the framework user may change this setting to her or his own implementation.

The behavior of the Analysis Component is explained in Section 4.2.2.

3.3.3 Reconfiguration Component

The function of the Reconfiguration Component is the addition and removal of instances of the Duplicable Stage based on the result of the analysis. In Figure 3.9, a UML diagram of the Reconfiguration Component and its related classes is shown.

The ReconfigurationCommandService class contains the logic on how the Reconfiguration Component behaves depending on the results of the Analysis Component: it could create another instance of the Duplicable Stage, it could remove one of the existing worker stages, or it could do nothing at all in that iteration. The decideExecutionPlan() method decides the following action (new instance, remove instance, or no action) using the given Throughput Score every time it is called (once per Adaptation Thread cycle). The samplesUntilRemove attribute corresponds with the maxSamplesUntilRemove introduced in Section 3.2. decideExecutionPlan() returns one value of the enumeration TaskFarmReconfigurationCommand, which corresponds to the addition and removal of instances of the Duplicable Stage.

At last, the TaskFarmController is called. Using addStageToTaskFarm() and removeStageFromTaskFarm(), the TaskFarmReconfigurationService executes the previously decided TaskFarmReconfigurationCommand.

The History Component and the Analysis Component both only depend on throughput measurements and, thus, do not depend on the type of the processed data elements of the
3.3. Adaptation Thread

Figure 3.9. Overview of the Reconfiguration Component in the context of the whole Adaptation Thread
3. Structure of the Task Farm Stage

Task Farm Stage. However, the Reconfiguration Component is different. The TaskFarmController directly manipulates the Task Farm Stage and its Dynamic Distributor and Dynamic Merger. Thus, it does need to know type parameters of the Distributor and Merger as well as the type of the Duplicable Stage the TaskFarmController duplicates. Without this information, much less type safety would be achieved.

More details on how the TaskFarmReconfigurationCommand is decided in each cycle of the Adaptation Thread are given in Section 4.2.3.

3.4 Monitoring Components

3.4.1 Monitoring Services

Since the Task Farm Stage possesses many different configuration settings (for more details see Section 3.2), it is useful for the framework user to be able to debug the Task Farm Stage and its behavior. For this purpose, two different Monitoring Services are designed. A UML class diagram of the Monitoring Services is shown in Figure 3.10.

![Diagram of Monitoring Services](image)

**Figure 3.10.** Overview of the Monitoring Services

We use two Monitoring Services, which each serve different purposes. The PipeMonitoringService observes the pipes between the Distributor and the instances of the Duplicable Stage. Every time the doMeasurement() method is called, a new monitoring measurement instance of PipeMonitoringData is created. This instance contains for a single pipe at a single point in time the following data fields:

- **time** This is the time of the moment when this data was recorded.

```
3.4. Monitoring Components

- **numPushes** This is the number of elements which have entered the pipe.
- **numPulls** This is the number of elements which have left the pipe.
- **size** This is the current number of buffered elements inside the pipe.
- **capacity** This is the maximal number of buffered elements inside the pipe. It corresponds to the pipeCapacity configuration setting introduced in Section 3.2.
- **pushThroughput** This is the number of elements which have entered this pipe since the last measurement.
- **pullThroughput** This is the number of elements which have left this pipe since the last measurement by an instance of the Duplicable Stage.
- **numWaits** This is the number of times an element could not yet be added to the pipe because it was full (i.e., size equals capacity).
- **uniquePipeId** This attribute is used for bookkeeping purposes to be able to later combine the PipeMonitoringData instances for each pipe.

Using these measurements, the exact behavior of the pipes themselves can easily be observed. However, if we want to have information about the Task Farm Stage itself, we need to use the SingleTaskFarmMonitoringService. Similar to the PipeMonitoringService, it creates instances of the TaskFarmMonitoringData class, which contains the following data:

- **time** This is the time of the moment when this data was recorded.
- **stages** This is the number of instances of the Duplicable Stage currently used in the Task Farm Stage.
- **meanPullThroughput** This is the current mean value of all the pullThroughput values of the pipes.
- **meanPushThroughput** This is the current mean value of all the pushThroughput values of the pipes.
- **sumOfPullThroughput** This is the current sum of all the pullThroughput values of the pipes.
- **sumOfPushThroughput** This is the current sum of all the pushThroughput values of the pipes.
- **throughputBoundary** This is the current Throughput Boundary as defined in Section 3.2.

The mean values as well as the sums could also be manually obtained by analyzing the PipeMonitoringData instances. However, if we only, for example, need to know how well the Task Farm Stage performs, we generally only need aggregated data like mean values or sums. For this purpose, the SingleTaskFarmMonitoringService is useful.
3. Structure of the Task Farm Stage

Since all Monitoring Services are derived from IMonitoringService, the same methods are used to access them. doMeasurement() is used to create a new data instance and getData() returns the collected data. addMonitoredItem(K) adds an item (i.e. a pipe or a Task Farm Stage) to be monitored if doMeasurement() is called.

These monitoring classes are integrated in the Task Farm Stage as well as the Adaptation Thread. Thus, they can be used by setting monitoringEnabled to true (see Section 3.2). However, the PipeMonitoringService also works for more universal use cases. It is able to monitor any pipe in the Pipe & Filter architecture as long as the specific pipe implementation supports monitoring.

3.4.2 Data Export Components

Besides the collection of data for performance optimization purposes, we also implement a mechanism to export available monitoring data to a CSV file. For this purpose, the Data Export classes (from now on: Exporters) are available. In Figure 3.11, a UML class diagram of the Data Export classes is given.

The AbstractMonitoringDataExporter class implements the basic functionality to extract the monitored data to CSV files. It requires instances of PipeMonitoringService and SingleTaskFarmMonitoringService (they should already contain some monitoring data). Deriving from this abstract class, there are two type of Exporters:

1. AbstractGeneralCSVExporter describes an Exporter which will export the data of all TaskFarmMonitoringData instances to a CSV file.

2. AbstractStackedCSVExporter provides methods to export all PipeMonitoringData instances to a CSV file. It is called “stacked” because for every time value, all pipe data are exported in the same line. Using the stacked data format, it is easy to, for example, create stacked diagrams of the pullThroughput.

For both types of Exporters, several concrete classes are available for use. The included data can be deduced from the specific class names. For example, TimeBoundaryStages3D creates a CSV file with the time, Throughput Boundary, and stages data columns. In contrast, StackedTimePullThroughput2D creates a CSV file using the time column as well as one column for each pipe containing the corresponding value of pullThroughput.

Besides our ready-to-use Exporters, the user is free to define custom Exporters. However, the data given by the PipeMonitoringService and SingleTaskFarmMonitoringService currently cannot be extended. Therefore, the given data has to suffice for any new Exporter.
3.4. Monitoring Components

Figure 3.11. Overview of the Data Export components
After introducing the structure of the Task Farm Stage and its Adaptation Thread, we can now direct our attention to their behavior.

Before further discussion, a crucial aspect to understand is the initialization of the Task Farm Stage. At creation time of the Task Farm Stage, the Dynamic Distributor, the Dynamic Merger, and the Adaptation Thread are also created. Since the Dynamic Merger is the last element to be started at the beginning and to be terminated at the end (see Section 3.2), it is also used to start and terminate the Adaptation Thread. Before the Merger starts, any throughput values measured by the Adaptation Thread would not be useful. After all, in that moment, elements might already traverse the Distributor and the (for now) sole instance of the Duplicable Stage. That means, the pipes connecting the instance of the Duplicable Stage with the Merger and the Distributor might already be full. The worker stage would not process any more elements, since its output port does not yet accept new elements, and the throughput would be artificially low. Therefore, the Adaptation Thread is started right after the Merger is started. After the termination of the Merger, no elements or instances of the Duplicable Stage are remaining to be measured. Therefore, we can also terminate the Adaptation Thread after the Merger is terminated.

At first we create exactly one instance of the Duplicable Stage. Since we want to retain the basic correctness of the Pipe & Filter architecture, this basic instance never gets deleted by the Adaptation Thread. As long as the basic instance exists, all elements are guaranteed to be correctly executed by the Task Farm Stage.

For the basic operation of the Task Farm Stage, the general traversal of elements through it, and the addition and removal of instances of the Duplicable Stage, are described in Section 4.1. The Adaptation Thread’s components are discussed in Section 4.2. At last, the specific behavior of the Monitoring Components is described in Section 4.3.

4.1 Basic Behavior

4.1.1 Basic Traversal of Elements

A UML activity diagram illustrating the basic behavior of the Task Farm Stage in regard to the traversing elements is shown in Figure 4.1.

Elements passing through the Task Farm Stage have to take similar paths. At first, any element passes the Dynamic Distributor (see Section 3.2.1). As the name suggests, it
4. Behavior of the Task Farm Stage

Figure 4.1. Basic traversal of elements through the Task Farm Stage

distributes all incoming elements to any currently available output port. As explained in Section 3.1, every output port of the Dynamic Distributor is connected with an instance of the Duplicable Stage. Therefore, the Distributor acts as a way to distribute the incoming element load to multiple worker stages. Since in the case of the Task Farm Stage we use the NonBlockingRoundRobinStrategy, the elements passing through the Dynamic Distributor are evenly distributed to its output ports.

As the next step, elements pass exactly one instance of the Duplicable Stage. The NonBlockingRoundRobinStrategy distributes elements to all worker instances evenly. Thus, each instance has to be executed in a separate thread to achieve parallelization. Using an individual thread for each worker stage, the performance improves for tasks with CPU-intensive workloads in particular. After all, we do not assign any additional I/O
resources by parallelization.

At last, after passing through an instance of a Duplicable Stage, the element passes the Dynamic Merger (see Section 3.2.2). Similar to the Dynamic Distributor, the Merger takes available elements from the input ports and passes them to its output port. Since we use the BusyWaitingRoundRobinStrategy, the merging happens evenly over all currently available input ports. Therefore, the pipes between the instances of the Duplicable Stage and the Dynamic Merger are being consumed as fast as possible. If those pipes were not emptied fast enough, the instances of the Duplicable Stage might not be able to process any more elements because of full output pipes, which would greatly degrade performance.

However, there is still a downside to this strategy of basic traversal of elements. If the capacity of the pipes between worker stages and Distributor is too large while the total number of elements is rather low, it can have a negative impact on the efficiency of the parallelization. For example, let us assume a total of a hundred elements and a pipe capacity of a thousand elements. After the Task Farm Stage is started and the first few elements traverse is, the Adaptation Thread has not yet added any worker stages. Thus, the NonBlockingRoundRobinStrategy of the Dynamic Distributor now fills its first, and sole, output port with all hundred elements. Therefore, that initial instance of the Duplicable Stage now has to finish the whole workload by itself. Any parallelization of the Task Farm Stage after the elements are already distributed does not have any effect. After all, there is no remaining element the Dynamic Distributor could pass to any newly created instance of the Duplicable Stage. As such, we have to be certain that the pipe capacity is always much lower than the maximum number of elements.

We have already discussed why a single worker thread per instance of the Duplicable Stage is useful. However, the Dynamic Distributor and the Dynamic Merger are also executed in a separate thread each. Using separate threads is more efficient, since the alternative would be adding their workload to a worker thread. However, that might, depending on the concrete design of the Duplicable Stage, take CPU time from the worker stage, which would lessen the throughput. Other stages connected to the Task Farm Stage from the outside may use the same thread as the Dynamic Distributor or the Dynamic Merger though. However, if the Distributor or the Merger do not have enough CPU time to pass their elements, the performance of the whole Task Farm Stage decreases. Thus, not too many stages should use the same thread as the Distributor or the Merger.

4.1.2 Addition of New Worker Stage

Another basic feature of the Task Farm Stage is the ability to dynamically add new threads to further parallelize all computations. To achieve this, a new instance of the Duplicable Stage is created as a new worker thread by the Adaptation Thread and connected to the Dynamic Distributor and the Dynamic Merger at runtime. A UML activity diagram in Figure 4.2 illustrates how and in which order these steps are done.

Figure 4.3 shows a model of a Task Farm Stage creating a new stage. In the following, we refer to the components of this model with their assigned numbers in the figure.
4. Behavior of the Task Farm Stage

Figure 4.2. Activity diagram showing the creation of a new instance of the Duplicable Stage

Figure 4.3. Task Farm pattern showing the components related to the creation of a new worker stage
4.1. Basic Behavior

At first we need to create the new instance of the Duplicable Stage (1). Otherwise we would not have a stage we could connect to the Distributor (2) or the Merger (3) in the first place. As mentioned in Section 3.2.3, the creation of a new worker stage can easily be done using the duplicate() method of the ITaskFarmDuplicable interface. However, the new instance is not yet executed by the framework, but only created.

The next step is to create pipes (4) between our new instance and the Distributor and Merger. As explained in Section 3.2.1 and Section 3.2.2, we use the concept of Port Actions to achieve this. Specifically, we first create an instance of CreatePortAction for the Dynamic Distributor and add the PortAction using the addPortActionRequest(PortAction) method. The CreatePortAction later creates a new port for the Distributor (2) as well as the new pipe (4). Since the newly created instance of the Duplicable Stage (1) is not executing at the moment, the Dynamic Distributor can only fill the pipe buffer (4) at this point. However, no element can be processed yet.

Similar to the previous step, another CreatePortAction has to be created for the Dynamic Merger (3). The addition of the PortAction to the Merger happens in the same way as with the Dynamic Distributor. However, the buffer of the pipe (5) between our new instance and the Merger does not get filled yet. After all, the new worker stage (1) was never started.

Finally, as the next and last step, the new instance of the Duplicable Stage (1) is started within its own worker thread. Now it starts processing elements passed to it by the Dynamic Distributor (2).

There are some intricate particularities concerning the handling of Port Actions by the Dynamic Distributor and the Dynamic Merger. As such, we discuss the Port Actions in more detail. In Figure 4.4, a UML sequence diagram of the behavior of the CreatePortAction is shown.

In the diagram, we see the instance of the TaskFarmController, the Dynamic Distributor, and the Dynamic Merger. Following the lifelines of the Distributor, new elements are constantly processed. Each time an element is processed, the Distributor checks for any Port Actions (method calls 2, 5, 11, 15, 19). If no PortAction is found, the current elements gets distributed and the method terminated (gray-colored method-call boxes).

Now we assume that the TaskFarmController creates a new instance of a CreatePortAction for the Distributor (method call 1). In the diagram, all objects directly related to that CreatePortAction have a yellow background. Just after the new PortAction is created, it is added to the Dynamic Distributor by the addPortActionRequest(PortAction) method. Such a method might be called, as can be seen here, between the normal activities of the Distributor, but also during those activities. Therefore, the call has to be suitably synchronized to avoid any concurrency errors. Since the TaskFarmController has to ensure that the new instance of the Duplicable Stage has been correctly initialized and started, it now waits until the CreatePortAction has been executed (method call 6).

That means that, for the TaskFarmController to stop waiting, we need to wait for a new element in the Dynamic Distributor. While checking for Port Actions (method call 5), the CreatePortAction is executed and the new Distributor output port as well as the pipe between
4. Behavior of the Task Farm Stage

Distributor and the new worker instance is created. As the Port Action has been completed, the waitForCompletion() method returns (see return message 8) and the TaskFarmController continues.

The next step of the TaskFarmController is, as already shown in the activity diagram (Figure 4.2), creating another instance of CreatePortAction for the Dynamic Merger. As with the Distributor, it is added to the Dynamic Merger in a concurrent manner before waiting for completion of the Port Action. In the instant the Merger processes an element, the new CreatePortAction is found and executed, creating a new input port for Dynamic Merger as well as a new pipe between the new instance of the Duplicable Stage and the Merger. Now the TaskFarmController stops waiting (see return message 17) and starts the new worker thread (not shown in Figure 4.4).
4.1. Basic Behavior

We have seen that the Dynamic Distributor and the Dynamic Merger need to process an element before executing Port Action. This behavior makes the concurrent implementation much less complex in terms of implementation. The Port Actions directly change the structure of the Distributor or Merger. Also, the TaskFarmController is always executed in a different thread. Thus, the complexity of correctly synchronizing all changes as well as the pipe creation outweights the downsides of using these Port Actions and waiting for their execution by far. As long as the potentially waiting TaskFarmController is interrupted at the end of execution, the port action approach does not cause any infinite waiting. The high frequency of elements processed by the Distributor and the Merger also leads to a very similar runtime behavior to if we directly added ports to the Distributor and the Merger, even if port actions are not executed instantly, as we have seen.

4.1.3 Removal of Worker Stages

Since our Task Farm Stage should be able to remove instances of the Duplicable Stage, we also need to discuss the removal of resources. A UML activity diagram shown in Figure 4.5 illustrates its behavior.

At first we need to remember that we do not want to compromise the correctness of the Pipe & Filter architecture. Therefore, we must not remove the basic first instance of the Duplicable Stage at any time. That means that our first step is to check whether there is more than one instance remaining. If this is not the case, we do not remove any existing stages and the removal process ends.

The next step, is to choose which instance should be removed. There are several possible strategies on how to choose a candidate for removal. In the following, a non-exhaustive list of them is given.

1. We choose the first available instance. This would probably be the most intuitive strategy, but it does not give us any further advantage.

2. We choose the stage with the longest mean computing time. However, for our general approach, this is not suitable. After all, we have to assume that different elements have different computation times if executed in a worker stage. If a worker stage needs a longer time for a specific element, it is, therefore, not always a sign for inefficiency.

3. Another way is to choose the worker stage with the least number of elements in its input pipe buffer. For this approach, the waiting time until the actual instance termination is shorter. After all, there are the least remaining elements we have to wait for until stage termination (elements still in the pipe buffer must not be discarded). Thus, we achieve a shorter time until a worker thread is actually terminated.

Since we want to take a general approach, we use the third strategy. After we have identified the stage to be removed, we can derive the Distributor output port we want to delete.
4. Behavior of the Task Farm Stage

For the next steps, the behavior is similar to the addition of resources in Section 4.1.2. We create a new instance of RemovePortAction and add it to the Dynamic Distributor. When the RemovePortAction is executed, the corresponding Distributor output port is closed, not allowing any new elements to enter the selected instance of the Duplicable Stage.

Contrary to the addition of resources (see Section 4.1.2), we do not change the Dynamic Merger. Thus, we have the advantage that any element remaining in the worker stage input pipe can safely be processed. The BusyWaitingRoundRobinStrategy of the Merger (see Section 3.2.2) is able to deal with any input ports never emitting any ports. After the last buffered element has been processed by the selected instance, worker stage is terminated and the corresponding input port of the Merger is closed. After the Merger waits for all input ports to close, it can then terminate itself. That means, that the removal of instances of the Duplicable Stage does not forcefully deletes any worker stages but allows them to
4.2 Adaptation Behavior

naturally terminate by not passing any new elements to them.

In Figure 4.6, a UML sequence diagram shows the removal of a stage with focus on the *RemovePortAction*.

![Sequence Diagram](image)

*Figure 4.6. Sequence diagram showing the removal of an output port in the Dynamic Distributor (yellow)*

The removal a worker stage works just the same as for the addition of a new stage in Section 4.1.2, only without affecting the *Dynamic Merger*. At first, we create a new instance of *RemovePortAction* (method call 1) and add it to the *Dynamic Distributor* (method call 3). Afterwards, the *TaskFarmController* waits for the completion of the *Port Action* (method call 5). When the *Distributor* processes a new element, the *RemovePortAction* is executed (method call 4.1) and removes the *Distributor* output port. Finally, the waiting of the *TaskFarmController* stops (return message 6) and the stage removal process is complete.

4.2 Adaptation Behavior

The *Adaptation Thread* consists of a main loop: it executes its underlying components, adds monitoring information, and waits for a specified time such that it does not require too much CPU time (parameter *adaptationWaitingTimeMillis*, see Section 3.2). Afterwards, the
4. Behavior of the Task Farm Stage

loop begins anew. In the following, we define an iteration of the Adaptation Thread as an iteration of its main loop.

The main functionality of the Adaptation Thread is located inside its components. Their behavior is illustrated in a UML activity diagram in Figure 4.7.

In the activity diagram, the three different components are painted in different colors. All objects related to the History Component have a red background and are explained in detail in Section 4.2.1. The Analysis Component uses a green background and is explained in Section 4.2.2. At last, the Reconfiguration Component is colored with an orange background and is discussed in Section 4.2.3.

4.2.1 History Component

In the following, the behavior of the History Component is described. As can be seen in Figure 4.7, at first the throughput of all pipes between the Dynamic Distributor and the instances of the Duplicable Stage is measured.

We later want decide if we add new resources. Therefore, we need to know whether the throughput of the whole Task Farm Stage (the sum of the throughput values of the pipes between Distributor and worker stages) has improved since the last few measurements. Therefore, we now calculate the sum of all throughput measurements.

The last step of the History Component is to save the sum of throughput measurement as well as its time of measurement. After all, the Analysis Component (see Section 4.2.2) might need the exact measurement time to make more exact calculations. To limit the memory usage, it is not necessary to record the entire history of the whole execution. In Section 3.2, we introduce the analysisWindow, which is the number of history measurements used by the Analysis Component. That means that we can remove older measurements whenever we add a new one.

4.2.2 Analysis Component

The Analysis Component calculates a Throughput Score, which shows just how much the current throughput of a given Task Farm Stage differs from its predicted value. For the calculation of the predicted values many different algorithms can be used. Generally, a certain amount of previous throughput measurements are used to extrapolate the future runtime behavior of the enclosed stages.

As a first step, we need a prediction for the current Task Farm Stage throughput based on the recorded measurements of the History Component. We can then compare that prediction with the actual measurement to see how much the performance deviates from the predicted value. In the following, a non-exhaustive list of algorithms to calculate this prediction is shown. Each algorithm is briefly introduced and then analyzed in regard to applicability to our use case.

Mean Algorithm This algorithm uses a previous throughput measurements of the Task Farm
4.2. Adaptation Behavior

Figure 4.7. Activity diagram showing the behavior of the Adaptation Thread
4. Behavior of the Task Farm Stage

Stage and calculates the average value of it. This value will then be interpreted as the expected value of the current point in time. The Mean Algorithm was used, for example, by Bielefeld [2012], but since it has a low complexity it has also been used in many other related work, like, for example, in Box and Jenkins [1976], Herbst et al. [2014], and Box et al. [2011].

This is one of the least complex algorithms that can be used to predict values. While it is very fast due to its few necessary computations, for our use case it does not produce acceptable results. For Task Farm Stage configurations where the throughput does not change over time, this algorithm yields good results. After all, if the throughput is constant, the average of the last few measurements is always a correct prediction. However, the algorithm loses this advantage for every other runtime behavior and cannot accurately predict future throughput measurements. In those cases, the Mean Algorithm is as accurate as than using the last measurement as the prediction value.

Weighted Algorithm The Weighted Algorithm is a variation of the Mean Algorithm. It still creates an average over a certain number of measurements, but it also adds weights in a way such that more recent measurements have more impact for the prediction. This weighting can be configured as exponential, linear, and logarithmic. Using exponential weighting, more recent measurements are exponentially more important in the calculation than less recent ones. The same applies to linear and logarithmic weighting. If exponential weighting is used, it is similar to the Single Exponential Smoothing Algorithm in Herbst et al. [2014].

For our use case, this algorithm is more usable than the Mean Algorithm. By weighting more recent measurements higher, the progression of the runtime behavior of the enclosed stages itself is analyzed. Thus, it reacts better to changes of the throughput measurements over time. Otherwise, the algorithm does not behave well for linear or exponential growth in the runtime of the enclosed stages, since it does not extrapolate any behavior of the throughput measurements. However, irregular runtime behavior yields relatively good results since extrapolation is not possible in those cases at all.

Regression Algorithm The Regression Algorithm uses a statistical regression algorithm to predict the throughput at the current point in time. It uses at least two previous measurements to construct a straight line \( y = ax + b \), where \( x \) is a point in time, \( y \) the throughput at that time, and \( a, b \in \mathbb{R} \). If there are more than two measurements used, a straight line is found that corresponds best to all provided data points. The prediction can be obtained by solving the equation by setting \( x \) to the current point in time. A variant of this algorithm has been introduced as the Cubic Smoothing Splines Algorithm in Herbst et al. [2014].

This algorithm behaves very well for nearly linear runtime behavior of the Task Farm Stage. Exponential and other relatively regular behavior can also be accurately predicted by using a lower amount of data points for the regression algorithm. This can be explained by the fact that those functions are mathematically nearly linear for a small
4.2. Adaptation Behavior

enough interval. Since the Regression Algorithm assumes a linear function in the runtime behavior, it can yield very imprecise predictions for irregular behavior.

ARIMA Algorithm The ARIMA Algorithm and its underlying mathematics is introduced by Box et al. [2011]. This algorithm is used for making a prediction for measurements without stability over a period of time [Bielefeld 2012]. It analyzed the provided measurements and chooses a computational model that fits best [Herbst et al. 2014]. Using this model, an accurate prediction can be made for almost all possible use cases.

As long as enough measurements are provided such that the ARIMA Algorithm can accurately choose the correct computational model, it is able to fulfill all our requirements. The only downside of this algorithm is a comparatively high computational overhead.

Croston’s Algorithm A specific variation of ARIMA Algorithm is used by the Croston’s Algorithm, but it is additionally able to handle zero values as throughput measurements. While creating the prediction, it removes the zero values from its analysis and separately analyzes the duration of the zero values as well as the remaining measurements. Both analyses are then unified to create the prediction [Herbst et al. 2014].

Since we do not retrieve zero values as throughput measurements for the Task Farm Stage, we also do not need to handle them separately. Thus, this algorithm is not necessary for our use case.

To summarize our findings, the ARIMA Algorithm would be most suited to be used in the Adaptation Thread of the Task Farm Stage. Unfortunately, the ARIMA Algorithm, as well as Croston’s Algorithm, are currently not natively implemented in Java to our knowledge. However, the statistical computing language R\(^1\) does include them. Since there is, according to our investigation, no native Java implementation of R methods available yet, these algorithms are not available to choose from. Nevertheless, we cover all use cases with the Regression Algorithm and Weighted Algorithm. However, in the future, it would definitely be worthwhile to find a way to implement a full ARIMA Algorithm for the Adaptation Thread.

Now that we have selected an algorithm we can calculate a prediction value with, we can now compute how much it deviates from the most recent measurement. Let \( p \in \mathbb{N} \) be the calculated predicted throughput value based on a number of most recent history measurements given by analysisWindow. Let \( v \in \mathbb{N} \) be the most recent history measurement which we calculated a prediction for. Then the Throughput Score \( ts \) is defined by

\[
ts = \frac{p - v}{p + v}.
\]

For example, let \( p = 3 \) and \( v = 1 \), i.e., the performance has increased since the last measurement. Then we can calculate \( ts = \frac{p-v}{p+v} = \frac{3-1}{3+1} = 0.5 \). For another example, let 1\http://www.r-project.org (visited on 10/22/2015)
4. Behavior of the Task Farm Stage

$p = 1$ and $v = 3$; the performance has decreased. Then the result is $ts = -0.5$. We see that a negative Throughput Score means the performance has decreased, while a positive score means that it has increased. The higher $|ts|$ is, the more definitive is the change. A Throughput Score of 0.5 means a greater performance improvement than a Throughput Score of 0.1.

All of these calculations are done by the AbstractThroughputAlgorithm as introduced in Section 3.3.2. They are independent from the Throughput Algorithms so that a framework user implementing a new algorithm only has to implement the algorithm for the prediction $p$ itself.

4.2.3 Reconfiguration Component

The last component of the Adaptation Thread, the Reconfiguration Component, is the most complex one. It includes the entire decision tree on how to react to certain Throughput Scores.

The decision tree is illustrated in Figure 4.7. At first we check which mode the Reconfiguration Component is in. There are two modes: ADDING and REMOVING. In the following, we consider the ADDING mode, which is also the initial mode.

Now we can check if this is the first iteration of the Adaptation Thread. If this is the first iteration, we add a new instance of the Duplicable Stage since we want to observe how much the addition of a stage improves performance.

Otherwise, the behavior of the Reconfiguration Component depends on the value of sampleUntilRemove. sampleUntilRemove decreases with the number of executions of the Reconfiguration Component. Every time the Reconfiguration Component executes, sampleUntilRemove decrements. After all, the addition or removal of instances of the Duplicable Stage takes some time to affect the throughput of the Task Farm Stage.

In the case that sampleUntilRemove reaches 0, we declare that the latest addition of an instance has apparently not satisfied our wishes for performance. Therefore, that stage is removed again to avoid performance overhead. Finally, we set our mode to the REMOVING mode.

If sampleUntilRemove is not yet 0, we check the Throughput Score, which was calculated in the Analysis Component previously. Therefore, we use the throughputScoreBoundary as introduced in Section 3.2. If the Throughput Score is higher than the throughputScoreBoundary, we accept the performance improvement as high enough. Therefore, we create a new Duplicable Stage instance. Additionally, we set sampleUntilRemove to maxSampleUntilRemove (as defined in Section 3.2) to reset our waiting time for a performance increase.

However, if the Throughput Score is not high enough (lower or equal than throughputScoreBoundary), we decrement our waiting time sampleUntilRemove and wait for the next iteration of the Adaptation Thread.

Now that we have introduced the ADDING mode, the REMOVING mode remains. As explained earlier, it is chosen if the Task Farm Stage cannot be parallelized anymore with a significant performance increase at the moment. That means, the Task Farm Stage is now at
4.3 Monitoring Behavior

its optimal configuration for its platform and concrete implementation of the Duplicable Stage according to the analysis. However, we can still try to remove overhead. Every time the Reconfiguration Component is executed, we check each pipe between the Dynamic Distributor and the instances of the Duplicable Stage and remove those stages which do not have input elements anymore.

For some Pipe & Filter architectures, not all elements are produced directly after one another. Sometimes, after the first few elements have passed the architecture, new elements are produced after a time of idleness. The Task Farm Stage can also handle this more irregular behavior while being in the REMOVING mode. At some point, the initial amount of input elements has been completely processed by the Task Farm Stage. If after some time new elements arrive, the Adaptation Thread handles these as follows. After removing unused Duplicable Stage instances, we check, similarly to the ADDING mode, if the latest Throughput Score exceeds the throughputScoreBoundary. If that is the case, new elements have arrived and the performance has increased. After we detect the performance increase, the mode changes to ADDING again and the Reconfiguration Component behaves like in the first iteration.

The above behavior is included in the ReconfigurationCommandService as introduced in Section 3.3.3. Another part of the Reconfiguration Component is the TaskFarmController. It provides the functionality to add and remove duplicate() instances to and, respectively, from the Task Farm Stage. The detailed behavior of this class is explained in Section 4.1.2 and in Section 4.1.3.

4.3 Monitoring Behavior

The behavior and usage of the Monitoring Components is illustrated by a UML activity diagram in Figure 4.8.

![Figure 4.8. Activity diagram showing the behavior of the Monitoring Components](image)

For the Monitoring Components, the first step is collecting all required data while the Adaptation Thread executes. The data collection is done via the SingleTaskFarmMonitoringSer-
4. Behavior of the Task Farm Stage

*vice* and the *PipeMonitoringService* introduced in Section 3.4.1. These services then have to be, after *Adaptation Thread* execution finishes, retrieved from the *Task Farm Stage*.

The last step is passing the *Monitoring Services* to any desired *Exporter* implementation. A few of these implementations are defined in Section 3.4.2, but the user can define own *Exporters* to export user-defined data. Since *Exporters* only have read-access to the monitoring data, the *Exporters* can be called in any order and even concurrently.
Chapter 5

Performance Evaluation

5.1 Methodology

Now that we have discussed the structure and design of the Task Farm Stage and its Adaptation Thread, we evaluate the runtime behavior of the Task Farm Stage in detail. For this purpose, we implemented the Task Farm Stage in TeeTime (see Section 2.3).

To create an evaluation as meaningful as possible, we test the Task Farm Stage implementation with the help of multiple test scenarios on different test systems. The employed test systems are listed in Section 5.2.1, while the scenarios are explained in Section 5.2.2.

By using the test scenarios and systems, we aim to achieve multiple goals with the evaluation:

1. The first goal is to assess if the Task Farm Stage has a varying runtime behavior on different systems. In the event that the runtime behavior differs, we want to identify which configuration parameters (see Section 3.2) have to be changed to achieve optimal performance of the Task Farm Stage.

2. In Section 4.2.2, we have discussed the importance of the Throughput Boundary for an optimal parallelization behavior of the Task Farm Stage. Therefore, it is useful to evaluate the computation time of the test scenarios on the different systems dependent on the Throughput Boundary used by the Adaptation Thread. Therefore, we achieve an overview of the efficiency of the Task Farm Stage.

3. The third goal is to analyze the runtime behavior of the Task Farm Stage by examining the throughput of the instances of the Duplicable Stage, again, dependent on the Throughput Boundary. Therefore, we observe not only the efficiency of the Task Farm Stage, but also the efficiency of the worker stages themselves.

Since we want to achieve reliable evaluation results, we calculate median values over all test results. That way, statistical spikes in our measurements do not have a high impact. However, using the median in a meaningful way means that we have to make more than one measurement per test case and system.

Figure 5.1 illustrates the different test runs we use for the evaluation. For a high statistical quality of the results, we execute three warmup runs, followed by five real runs for measurements, per test case. Furthermore, TeeTime and our test implementation are
5. Performance Evaluation

![Test Scenario](image)

**Figure 5.1.** Overview of the runs of a single test case execution

written in Java. Since the used Java Runtime Environment (JRE) uses varying optimization strategies on different runs, we execute the warmup and real runs thrice each. The JRE restarts on each repeat to obtain different optimization strategies. Therefore, our evaluation data also include the measurements over multiple JRE instances. Before analyzing the resulting data, we calculate the median over all these runs.

For each test case, we also execute the scenario five times completely without the help of a Task Farm Stage. Again, we calculate the median value of the performance measurements. Therefore, we can observe how much computation time is saved by the parallelization of the Task Farm Stage compared to single-threaded execution.

The resulting data and an analysis of them is given in Section 5.3. Possible threats to the validity of the evaluation are discussed in Section 5.4.

5.2 Experimental Setup

Before discussing the evaluation results themselves, we first address the test environment as well as the test scenarios. In Section 5.2.1, the basic hardware and software used for the evaluation is listed and described. We then discuss the test scenarios in Section 5.2.2.

5.2.1 Environment

In Table 5.1, an overview of the test systems and their configuration is shown. For our evaluation, we use four test systems: SUN, AMD-I, INTEL, and AMD-II. The systems are part of the Software Performance Engineering Lab\(^1\) of the University of Kiel. All systems possess two processors except AMD-II, which possesses only one. AMD-I, INTEL, and

\(^1\)http://www.se.informatik.uni-kiel.de/en/research/software-performance-engineering-lab-spel (visited on 10/22/2015)

48
5.2. Experimental Setup

AMD-II run Unix systems on x86-64 AMD or Intel architectures. In contrast, SUN operates using a SPARC V9 architecture with a Solaris operating system.

<table>
<thead>
<tr>
<th>System</th>
<th>SUN</th>
<th>AMD-I</th>
<th>INTEL</th>
<th>AMD-II</th>
</tr>
</thead>
<tbody>
<tr>
<td># Processors</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Processor</td>
<td>UltraSPARC T2+</td>
<td>AMD Opteron 2384</td>
<td>Intel E5-2650</td>
<td>AMD Opteron 2356</td>
</tr>
<tr>
<td>Architecture</td>
<td>SPARC V9 (64 Bit)</td>
<td>x86-64</td>
<td>x86-64</td>
<td>x86-64</td>
</tr>
<tr>
<td>Clock/Core</td>
<td>1.4 GHz</td>
<td>2.7 GHz</td>
<td>2.8 GHz</td>
<td>2.3 GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>16</td>
<td>8</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>Threads</td>
<td>128</td>
<td>8</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>RAM</td>
<td>64 GB</td>
<td>16 GB</td>
<td>128 GB</td>
<td>4 GB</td>
</tr>
<tr>
<td>Disk Space</td>
<td>134 GB</td>
<td>131 GB</td>
<td>437 GB</td>
<td>4 TB</td>
</tr>
<tr>
<td>Disk Contr.</td>
<td>RAID1/SAS</td>
<td>RAID1/SATA</td>
<td>SATA</td>
<td>RAID1/SATA</td>
</tr>
<tr>
<td>OS</td>
<td>Solaris 10</td>
<td>Debian 8</td>
<td>Debian 8</td>
<td>Debian 7</td>
</tr>
</tbody>
</table>

The two processors SUN system together possess 16 cores and support the concurrent execution of 128 threads, i.e., the number of virtual processors is 128. However, the clock rate of 1.4 GHz is slow compared to the other systems. The RAM and disk space are high enough to not limit us for our evaluation. After all, our test cases are not about high memory usage and there is not much disk space required either (see Section 5.2.2 for more information).

The two AMD-I processors consist of eight cores, which can concurrently execute one thread at a time each. However, the clock rate of 2.7 GHz is far higher than that of SUN. RAM and disk space are, again, high enough for our purposes.

The more recent processors of INTEL have a clock rate of 2.8 GHz. They provide combined 16 cores, which support 32 threads altogether. RAM and disk space are large enough for the execution of our test cases.

Lastly, the AMD-II system possesses the lowest computing power. The single AMD processor possesses a clock rate of 2.3 GHz and consists of four cores. Each core can only execute one thread concurrently. The RAM of 4 GB is not very high, but it is high enough not to negatively affect our test case execution. The disk space of 4 TB is, comparatively, extraordinarily high, since AMD-II is actually used as a storage server.

On all test systems, we use the Oracle Java Runtime Environment\(^2\) in the version 1.8.0_60-b27. As AMD-I, INTEL, and AMD-II are all 64 Bit systems running on Debian, they use the

\(^2\)http://www.oracle.com/technetwork/java/javase/overview/index.html (visited on 10/22/2015)
5. Performance Evaluation

Linux x64 build of the JRE. SUN uses the Solaris SPARC x64 build since it runs on a SPARC architecture.

5.2.2 Test Scenarios

In addition to a description of our test systems we also discuss our test scenarios. For that purpose, we define scenarios with different use cases: a test case mainly using CPU resources, an I/O-focused scenario, and a test case with a mix of a CPU- and I/O-intensive workload.

**CPUBAL/CPUUNBAL: CPU-Intensive Balanced and Unbalanced Workload**

The test case we are using for the CPU-intensive benchmark is illustrated by Figure 5.2.

![Figure 5.2. Pipe & Filter architecture of the CPU-intensive scenario with its parallelized stage (red border)](image)

The Pipe & Filter architecture consists of a producer stage (1) emitting a predefined amount of configurable integer values. The stage md5 (2) generates an MD5 hash value of any input number. The MD5 hashes then enters the Task Farm Stage. Inside the Task Farm Stage and after the Distributor (3), the MD5 Bruteforce stage (4) is located, which generates MD5 hashes for every number until it matches the input value. Therefore, it creates a CPU-intensive workload. It then outputs the number which generated the hash. Therefore, the output value of the MD5 Bruteforce stage (4) equals its corresponding output value of the producer (1). The MD5 Bruteforce stage is, in this case, also a concrete implementation of the Duplicable Stage and as such is parallelized (identified by the red border in Figure 5.2).

Then, after the element passes the Merger (5), it leaves the Task Farm Stage. At last, the number is discarded in a sink (6).

We use the model for the following two types of CPU-intensive workloads:

**Balanced Workload** For a CPU-intensive balanced workload, we generate a load such that every execution of the MD5 Bruteforce stage (4) will require roughly the same computation time. For this purpose, the producer (1) generates a configurable amount of the same predefined number. Now, any instance of the MD5 Bruteforce stage uses the same problem size for every input element. Therefore, each element has a similar
5.2. Experimental Setup

computation time, regardless of the concrete worker stage. From now on, we call this scenario CPUBAL.

CPUBAL has two configurable parameters. \emph{count} represents the number of elements the producer (1) generates. It directly affects the computation time. Furthermore, the \emph{problemSize} represents the number of each element produced. It affects the time each instance of the MD5 Bruteforce stage needs to process each element.

\textbf{Unbalanced Workload} For this scenario, we generate a load such that every use of the MD5 Bruteforce stage requires a varying computation time. For that purpose, instead of constantly producing the same numbers, we generate numbers 0, 1, ..., \(c\) with \(c\) as the total amount of output elements of the producer (1). Every execution of an instance of the MD5 Bruteforce (4) now has a different and linearly increasing computation time. After all, as described above, the complexity of the MD5 Bruteforce stage increases linearly with the output value of the producer (1). From now on, we call this scenario CPUUNBAL.

In contrast to CPUBAL, CPUUNBAL only needs one parameter. \emph{count} is, again, the count of elements generated by the producer (1). Since the value of the generated elements is defined by the current count of elements, it does not need to be specified.

\textbf{IO: I/O-Intensive Workload}

The I/O-intensive test case is illustrated by Figure 5.3.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure5.3.png}
\caption{Pipe & Filter architecture of the I/O-intensive scenario with its parallelized stage (red border)}
\end{figure}

For the I/O-intensive scenario, the producer (1) generates a configurable amount of the same predefined integer value. These elements then enter the Task Farm Stage and pass the Distributor (2). Afterwards, they enter the I/O Generator stage (3), which is also the concrete implementation of the Duplicable Stage in this scenario. Therefore, it is later parallelized by the Task Farm Stage (as signified by the red border in Figure 5.3). The I/O Generator stage creates a temporary file and fills it with characters. The total number of characters is determined by its input element, i.e., the integer value generated by the producer (1). Finally, the I/O Generator deletes the temporary file. Therefore, much I/O workload is
5. Performance Evaluation

created. The output value of the I/O Generator stage is a boolean flag if each I/O operation was successful. The output element then leaves the Task Farm Stage after it passes the Merger (4). At last, the sink (5) discards that element. From now on, we call this scenario IO.

Therefore, two parameters exist for IO. As for CPUBAL, we let count define the amount of elements generated by the producer (1). Furthermore, we define numChars as the number of characters written in the temporary files by the I/O Generator stage (3). count affects the computation time of the test case, while numChars affects the time a worker stage requires to process a single element.

**CPUIO: Combined CPU- and I/O-Intensive Workload**

A test scenario providing a CPU- as well as I/O-intensive workload is illustrated by Figure 5.4.

![Figure 5.4. Pipe & Filter architecture of the scenario providing both CPU- and I/O-intensive workload with its parallelized stages (red border)](image)

For the combined CPU- and I/O-intensive scenario, we first let the producer (1) send a path to an XML file for a configurable amount of times. The file path then enters the Task Farm Stage and passes the Distributor (2). Contrary to the previous scenarios, multiple stages are parallelized for the CPU- and I/O-intensive scenario by the Task Farm Stage. Consequently, multiple stages (3-5) are included in the concrete implementation of the Duplicable Stage (signified by the red border in Figure 5.4). The XML file path is then loaded into memory and parsed (3). Afterwards, the XSL-Transformer stage (4) transforms the XML document using an XSL document. The XSL transformation output is then saved to a temporary file, followed by a deletion of that file (5). The output of the saving stage (5) is a boolean flag showing if its I/O operations were successful. The flags then pass the Merger (6) and exit the Task Farm Stage. Finally, they are discarded by the sink (7).

The amount of I/O and CPU workload greatly depends on the XML input file and the XSL file used. Listing 5.1 shows an example XML input file. It contains, besides the root element, for each line a number, beginning at 0 and ending at the total number of lines. Therefore, we make the number of lines configurable to be able to dynamically create a suitable input XML file.

In Listing 5.2, the used XSL Transformation is shown. The transformation requires as
5.2. Experimental Setup

Listing 5.1. Example XML input file

```xml
<?xml version="1.0"?>
<root>
    <line>0</line>
    <line>1</line>
    <line>2</line>
    ...
    <line>xmlLines</line>
</root>
```

Listing 5.2. XSLT file used to reverse and concatenate the values within the `line` tags of an XML file as shown in Listing 5.1

```xml
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="text"/>
    <xsl:template match="/">
        <xsl:apply-templates select="root/line">
            <xsl:sort select="position()" data-type="number" order="descending"/>
        </xsl:apply-templates>
    </xsl:template>
    <xsl:template match="root">
        <xsl:value-of select="."/>
    </xsl:template>
</xsl:stylesheet>
```

its input an XML file formatted like the example in Listing 5.1. If applied, it then reverses the elements of the input XML file and concatenates the numbers of each line. Thus, the transformation returns a single string containing the reversed sequence of line numbers. For example, let us assume an XML input file with three line tags. The output, after the XSL Transformation is applied to the XML file, would be ‘210’.

For the reversal of the XML data, the whole file has to be read from disk and all elements have to be processed in memory. Furthermore, the result is then saved and deleted again. Therefore, this scenario provides a CPU- as well as I/O-intensive workload. From now on, we call this scenario CPUIO.

For CPUIO, we require three parameters. The first parameter `xsl` represents the path to the XSL Transformation we want to use. As explained above, we use the reversal transformation as shown in Listing 5.2. The second parameter is `numXmlLines`, which represents the number of `line` tags for the XML input file. Before the execution of the CPUIO test scenario starts, we dynamically create a suitable XML input file as defined by `numXmlLines`. 

53
5. Performance Evaluation

Regarding I/O workload, *numXmlLines* affects the duration per element of the loading (3) and saving (5) stages. Furthermore, a higher value of *numXmlLines* also increases the CPU workload generated by the XSL-Transformer (4), since a more extensive XML document is processed. The last parameter is, again, *count*. *count* represents the number of times the producer stage (1) sends an element containing the XML input file path. The parameter *count* affects the computation time of the whole test case.

5.3 Results and Discussion

Now that we have described the evaluation methodology as well as the experimental setup, we can discuss the evaluation results. In Section 5.3.1, we present the computation time each system took for each scenario dependent on certain parameters. In Section 5.3.2 we then closely analyze the behavior of the Task Farm Stage and its Adaptation Thread for one selected system.

5.3.1 Overview of the Computation Times

For the evaluation, we use a value of 50 ms for *adaptationWaitingTimeMillis* (see Section 3.2), i.e., the Adaptation Thread does one iteration every 50 ms. Furthermore, we let the pipe capacity (see also Section 3.2) be 100 for a low performance overhead of the Task Farm Stage. The remaining configuration parameters of the Task Farm Stage are kept to their default values as defined in Section 3.2. We note that the *maxNumOfCores* parameter is, therefore, set to the number of virtual cores (threads) minus two. As discussed in Section 3.2, the Distributor and the Merger are executed in their own threads. Therefore, we reserve them a thread each, resulting in a maximum number of virtual cores minus two worker stages.

The Adaptation Thread performs 40 iterations in two seconds. Therefore, in the event of a computation time of two seconds or above, we have enough time to observe the behavior of the Task Farm Stage, at least regarding the computation time. Therefore, our goal is to choose the parameters of the test scenarios (as defined in Section 5.2.2) in a way for the scenario to at least require two seconds of computation time.

**CPUBAL Scenario**

The values of the parameters *count* and *problemSize* for CPUBAL as defined in Section 5.2.2 are shown in Table 5.2. **SUN** possesses, as we have seen in Section 5.2.1, the lowest computing power per virtual processor core. Thus, *count* and *problemSize* need to be set much lower for that system. The other systems require higher parameters to satisfy our previously defined computation time goal.

Figure 5.5 shows the diagrams for the CPUBAL scenario. We consider the results of the **SUN** system first, which is shown in Figure 5.5a. The behavior of the **Mean Algorithm** (blue) is very irregular. The computation time is always high for a Throughput Boundary
Table 5.2. Used parameters of scenario CPUBAL on the different test systems

<table>
<thead>
<tr>
<th>count</th>
<th>problemSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUN</td>
<td>1000</td>
</tr>
<tr>
<td>AMD-I</td>
<td>1500</td>
</tr>
<tr>
<td>INTEL</td>
<td>2000</td>
</tr>
<tr>
<td>AMD-II</td>
<td>2000</td>
</tr>
</tbody>
</table>

Figure 5.5. Computation times for the CPUBAL scenario
5. Performance Evaluation

greater than 0.8. No real dependency between the computation time and the Throughput Boundary can be seen. Furthermore, the standard deviation is for a Throughput Boundary lower than 0.8 always around 6 seconds. That standard derivation is exceedingly high for this scenario, since the maximal computation time using this algorithm lies around 19 seconds. Additionally, the other algorithms lead to a standard derivation of about one second, making the Mean Algorithm comparatively unreliable.

The Weighted Algorithm (green) provides a much more predictable behavior than the Mean Algorithm. However, it only provides good performance for very low Throughput Boundaries. However, a Throughput Boundary of around 0.0 results in further parallelization even if we did not measure any performance increase (see Section 4.2.3). Therefore, the Task Farm Stage might create more worker threads than necessary for good performance, leading to overly busy systems, higher power consumption, and a worse runtime behavior.

The Regression Algorithm (brown) behaves similar to the Weighted Algorithm. However, it is most efficient for a Throughput Boundary of 0.05, leading to better thread usage than the Weighted Algorithm (as discussed above).

Without a Task Farm Stage, the CPUBAL scenario needs about 21 seconds to complete on SUN (see the red graph in Figure 5.5a). Therefore, if we use the Regression Algorithm with a Throughput Boundary of 0.05, we achieve a speedup of about 4. Compared to the 16 available cores and 128 concurrent threads, this is a low value. Thus, the Task Farm Stage could not fully benefit the many available virtual processor cores available in SUN. After all, we expect a far higher speedup than 4 with the 16 physical cores of SUN. However, the performance is still significantly increased compared to single-threaded execution.

Compared to the SUN system, the Throughput Algorithms are more reliable in AMD-I, as shown in Figure 5.5b. The Mean Algorithm does not provide any performance increase at all, regardless of the Throughput Boundary. Thus, it is not a useful algorithm in this case.

The Weighted Algorithm behaves similar to the SUN system with the same disadvantages. However, the best possible computation time of 5 seconds is still not optimal.

The Regression Algorithm provides the lowest possible computation time for the CPUBAL test case on AMD-I. It behaves, as for SUN, optimally for a Throughput Boundary of 0.05.

Overall, using the Regression Algorithm with a Throughput Boundary of 0.05 achieves a speedup of 4 compared to single-threaded execution. Since AMD-I only has 8 cores, AMD-I is much better utilized than SUN.

Now let us discuss the more recent system of INTEL in Figure 5.5c. Again, the Mean Algorithm provides no improvement in computation time, regardless of the Throughput Boundary. Furthermore, the Weighted Algorithm provides good results only for a Throughput Boundary of 0.0. The Regression Algorithm achieves the highest performance for Throughput Boundaries up to 0.05.

Using the Regression Algorithm with a Throughput Boundary of 0.05, we achieve a speedup of around 6 compared to single-threaded execution. With 16 cores and 32 threads, this is no optimal utilization, but much better than with SUN.

The last evaluation system is AMD-II, for which the results are shown in Figure 5.5d.
5.3. Results and Discussion

AMD-II only provides 4 cores, of which two are needed for the Distributor and the Merger. Thus, as discussed earlier, the Task Farm Stage only parallelizes up to a maximum of two worker stages. We see this limitation clearly in the diagram. No matter the algorithm or Throughput Boundary, the two worker stages are used to create a speedup of 2. We cannot realistically expect more speedup under these circumstances.

At last, it is noteworthy that even using its least efficient Throughput Algorithms and Throughput Boundaries, the performance of the Task Farm Stage is never lower than that of the single-threaded execution. The reason for the small unconditional performance increase is that even without parallelization, each scenario that uses a Task Farm Stage employs at least three threads in TecTime (see Section 2.3). The first thread reaches from the producer stage to the Distributor. The second thread consists of the first instance of the Duplicable Stage. The last thread reaches from the Merger to the sink. Therefore, even without parallelization by the Adaptation Thread, the architecture of the Task Farm Stage provides a small amount of concurrency. This concurrency leads to a performance increase.

CPUUNBAL Scenario

The values of count and CPUBAL as defined in Section 5.2.2 are shown for the different test systems in Table 5.3. Again, SUN provides a much lower number of elements because of its lower computing power per CPU core. AMD-I, INTEL, and AMD-II require much higher values for count.

<table>
<thead>
<tr>
<th>System</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUN</td>
<td>1700</td>
</tr>
<tr>
<td>AMD-I</td>
<td>10000</td>
</tr>
<tr>
<td>INTEL</td>
<td>10000</td>
</tr>
<tr>
<td>AMD-II</td>
<td>7000</td>
</tr>
</tbody>
</table>

Table 5.3. Used parameters of scenario CPUUNBAL on the different test systems

Figure 5.6 shows the diagrams of the CPUUNBAL scenario for each test system. Let us start with the discussion of SUN. The results of SUN are shown in Figure 5.6a. All Throughput Algorithms reach their lowest computation time at a Throughput Boundary of 0.0. However, the Regression Algorithm (brown) performs better than the other algorithm for Throughput Boundaries between 0.0 and 0.1.

While the runtime behavior might not be ideal at low Throughput Boundaries (as discussed earlier), the most reliable result with the optimal computation time requires the Regression Algorithm with a Throughput Boundary of 0.0, reaching a speedup of 4 compared to single-threaded execution (red). That speedup does, again, not show a good utilization of SUN.

The results of the AMD-I system are shown in Figure 5.6b. The Mean Algorithm (blue)
5. Performance Evaluation

![Graphs showing computation times for CPUUNBAL scenario](image)

(a) SUN system  
(b) AMD-I system  
(c) INTEL system  
(d) AMD-II system

**Figure 5.6.** Computation times for the CPUUNBAL scenario
and the Regression Algorithm (brown) both reach optimal computation times at a Throughput Boundary of 0.05. The Weighted Algorithm (green) provides, again, only good results with a Throughput Boundary of 0.0.

Using the Regression Algorithm and a Throughput Boundary of 0.05, AMD-I achieves a speedup of around 6 compared to single-threaded execution. Since AMD-I provides 8 cores, this is a very good utilization of the system.

Regarding the INTEL system, the results are shown in Figure 5.6c. Every Throughput Algorithm shows the same behavior for CPUUNBAL on INTEL. They are very efficient for a Throughput Boundary of 0.0, while leading to a much worse performance for higher Throughput Boundaries.

If we use the Regression Algorithm with a Throughput Boundary of 0.0, we achieve a performance speedup of around 7.5 compared to single-threaded execution. With the 16 cores in mind, this is a good utilization of INTEL.

At last, we show the behavior of the AMD-II system in Figure 5.6d. Since we, again, have only two CPU cores available for parallelization, both cores are fully utilized no matter the Throughput Algorithm or the Throughput Boundary. Therefore, we nearly achieve a speedup of 2 compared to single-threaded execution.

**IO Scenario**

The values of the parameters count and numChars of IO as defined in Section 5.2.2 are shown for the different test systems in Table 5.4. As INTEL is a more recent system, the I/O performance is also better. Therefore, count and numChars are set higher on INTEL. However, on AMD-II the I/O performance is lower than on the other systems. Thus, we reduce count on AMD-II. On SUN, AMD-I, and AMD-II, the numChars is set to 3000000, resulting in a file size of 3 MB. In contrast, INTEL uses a numChars of 5000000, resulting in a file size of 5 MB.

<table>
<thead>
<tr>
<th></th>
<th>count</th>
<th>numChars</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUN</td>
<td>1000</td>
<td>3000000</td>
</tr>
<tr>
<td>AMD-I</td>
<td>1000</td>
<td>3000000</td>
</tr>
<tr>
<td>INTEL</td>
<td>1500</td>
<td>5000000</td>
</tr>
<tr>
<td>AMD-II</td>
<td>500</td>
<td>3000000</td>
</tr>
</tbody>
</table>

In Figure 5.7, the computation time diagrams for the IO scenario are shown. Let us first discuss the SUN system in Figure 5.7a. The Mean Algorithm (blue) does not provide good performance, regardless of the Throughput Boundary. The Weighted Algorithm (green), while being more efficient than the Mean Algorithm, does not show optimal performance either. However, the Regression Algorithm (brown) is the most efficient Throughput Algorithm at a
5. Performance Evaluation

![Graphs showing throughput boundary over time for different systems: SUN system, AMD-I system, INTEL system, and AMD-II system.](image)

**Figure 5.7.** Computation times for the IO scenario
5.3. Results and Discussion

Throughput Boundary of 0.05. Using the Regression Algorithm with a Throughput Boundary of 0.05, we reach a performance speedup of around 3 compared to single-threaded execution (red).

The results of the AMD-I system are shown in Figure 5.7b. On AMD-I, every algorithm reaches the optimal computation time for the IO scenario. However, the difference lies in the Throughput Boundaries. The Mean Algorithm is most efficient at a Throughput Boundary of 0.05. The Weighted Algorithm performs best at a Throughput Boundary of 0.175. However, the Regression Algorithm achieves the best computation time at a Throughput Boundary of 0.225. Since, as discussed earlier, we want to avoid parallelization without performance improvement, we choose the algorithm with the highest possible Throughput Boundary at low computation time: the Regression Algorithm. Using the Regression Algorithm at a Throughput Boundary of 0.225, we achieve a performance speedup of 3 compared to single-threaded execution.

The INTEL system, as shown in Figure 5.7c, behaves similar to the SUN system in the IO scenario. The Mean Algorithm provides nearly no performance increase. The Weighted Algorithm performs better, but does not reach the lowest possible computation time at its optimum. However, the Regression Algorithm performs the most efficient at a Throughput Boundary of 0.025. Using the Regression Algorithm at a Throughput Boundary of 0.025, we achieve a performance speedup of 3.5 compared to single-threaded execution.

The results of the AMD-II system are shown in Figure 5.7d. Independently from the used Throughput Algorithm or Throughput Boundary, the two worker stages achieve a speedup of around 1.5 compared to single-threaded execution.

**CPUIO Scenario**

The values of the parameters count and numXmlLines of CPUIO as defined in Section 5.2.2 are shown for the different test systems in Table 5.5. As INTEL provides higher CPU and I/O performance, count and numXmlLines are higher for INTEL than for the other systems. Since SUN provides lower computing power per core, we reduce numXmlLines for that system.

<table>
<thead>
<tr>
<th>System</th>
<th>count</th>
<th>numXmlLines</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUN</td>
<td>1000</td>
<td>400</td>
</tr>
<tr>
<td>AMD-I</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>INTEL</td>
<td>1500</td>
<td>2000</td>
</tr>
<tr>
<td>AMD-II</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

In Figure 5.8, the results of the CPUIO scenario are shown. Again, let us begin with the SUN system, whose results are shown in Figure 5.8a. The Mean Algorithm (blue) performs
5. Performance Evaluation

Figure 5.8. Computation times for the CPUIO scenario
5.3. Results and Discussion

well for very low Throughput Boundaries, but does not reach the lowest possible computation time. The Weighted Algorithm (green) does reach the lowest possible computation time for a Throughput Boundary of 0.025. However, the Regression Algorithm (brown) reaches that optimum for a Throughput Boundary of 0.1. Since we want to maximize the Throughput Boundary (as discussed above), the Regression Algorithm is the most useful Throughput Algorithm here. Using the Regression Algorithm at a Throughput Boundary of 0.1, we achieve a speedup of around 5.5 compared to single-threaded execution (red).

The results of the AMD-I system are shown in Figure 5.8b. The Mean Algorithm does not improve performance at all. In contrast, the Weighted Algorithm and the Regression Algorithm reach the lowest possible computation time for a Throughput Boundary of 0.00 and 0.025 respectively. As such, the Regression Algorithm is the most useful Throughput Algorithm in this case. Using the Regression Algorithm at a Throughput Boundary of 0.025, we achieve a speedup of around 3.5 compared to single-threaded execution.

For the INTEL system, the results are shown in Figure 5.8c. The Mean Algorithm does not provide any performance increase at all. Similarly, the Weighted Algorithm also does not decrease the computation time by much. However, the Regression Algorithm provides the lowest possible computation time at a Throughput Boundary of 0.05. Using the Regression Algorithm at a Throughput Boundary of 0.05, we achieve a speedup of around 5 compared to single-threaded execution.

The last system is AMD-II, whose results are shown in Figure 5.8d. Again, the performance does not change with the used Throughput Algorithm or Throughput Boundary. By using its two worker stages, a speedup of about 2.5 is achieved compared to single-threaded execution.

**Insights from the Analysis of the Computation Times**

We can gather some insights from the analysis of the diagrams shown in this chapter. For one, we see much lower speedups for the IO scenario than for the CPUBAL and CPUUNBAL scenarios. The reason is that, even though we add worker threads during parallelization, every worker thread utilizes the same underlying I/O resources. At some point, the available I/O resources are fully utilized. Therefore, further parallelization does not affect computation times anymore if the focus lies on an I/O-intensive workload.

While the speedup of the IO scenario is generally low, we can see in the CPUIO scenario that the Task Farm Stage can handle I/O workload combined with CPU workload. As long as the implementation of the Duplicable Stage does not predominantly employ I/O workload, the speedup reached because of the Task Farm Stage does not suffer.

In all scenarios on all systems, we have seen that the Regression Algorithm is either superior or equal in performance to the Mean Algorithm and the Weighted Algorithm. Therefore, the Regression Algorithm has shown the best results for our evaluation. In contrast, the Mean Algorithm is sometimes unreliable and oftentimes leads to a bad performance.

Generally, we have seen that we should choose low values for the Throughput Boundary. Throughput Boundaries greater than 0.3 oftentimes lead to a high computation time.
5. Performance Evaluation

Sometimes, even very low values such as 0.0 or 0.025 should be chosen. The most efficient Throughput Boundary highly depends on the test system, the scenario, and the Throughput Algorithm.

For all test cases, we have measured a significant speedup. However, for systems with a high amount of CPU cores and threads, the speedup is not as high as we might have expected. For example, SUN employs CPUs with 128 threads combined. The Task Farm Stage does however never reach a speedup even approaching 128. As we have discussed in Section 4.1.1, this behavior is a downside of a high pipe capacity. For our test scenarios on the SUN system, we use 1000 to 1700 elements while employing a pipe capacity of 1000. Therefore, even if we were to parallelize ideally and instantly, the Distributor could only provide the first 10 to 17 worker stages with elements until all elements are distributed. Even if new worker stages were to be added, they Distributor would not pass them any elements. Thus, the amount of parallelization is limited in this case.

Therefore, we now test if we could reach a higher performance with different parameters on SUN for the CPUBAL scenario. Let us set the pipe capacity to 10, while changing the values of count and problemSize to 100000 and 30, respectively. Therefore, there are enough elements and a low enough pipe capacity to guarantee a high amount of parallelization. Now we run the test using all three available Throughput Algorithms. However, even at the lowest possible computing time, we do not reach a higher speedup than 3 compared to single-threaded execution. This is not due to a lack of parallelization, since all 128 threads were used. Therefore, the overhead due to stage communication and due to distributing and merging elements to and from many stages is high enough to lower the performance significantly. As such, we can deduce that the Task Farm Stage is currently not able to fully utilize the SUN system. However, we see in Section 5.3.2 that a better speedup is possible with the INTEL system.

5.3.2 Detailed Analysis using the INTEL System

Now that we have observed the general performance of the Task Farm Stage, we choose one system to analyze in detail. For that purpose, we pick the INTEL system, which is the most recent one of all available test systems. In Section 5.3.1, we found out that the Regression Algorithm generally shows better performance all other Throughput Algorithms in our tests. Therefore, we employ the Regression Algorithm for our detailed analysis.

In Section 5.3.1, our results show that, using the Regression Algorithm, the computation time of some scenarios is shorter than five seconds. This happens, for example, for the IO scenario (see Figure 5.7c). This is not desirable since the parallelization might not be fully completed after only a short computation time. After all, not in every Adaptation Thread iteration are new stages added because of maxSamplesUntilRemove (see Section 3.2). Since we want to show the behavior of the Task Farm Stage in detail in this chapter, we have to be certain that the parallelization during runs is fully completed before the test run is finished. Therefore, to create a longer computation time, we increase the count parameter of each test case. Additionally, to ensure a high amount of parallelization (after all, we want to
analyze the parallelization behavior of the Task Farm Stage, we use a pipe capacity of 10 (see Section 3.2).

For the detailed analysis, we observe the behavior of the Task Farm Stage in several diagrams for each test scenario. The first diagram shows the computation time dependent on the Throughput Boundary. Since the parameters are different than in Section 5.3.1, the computation times have changed, too. Secondly, we provide a 3D diagram showing the number of stages for a specific Throughput Boundary at any point in time during the execution of the Task Farm Stage. Therefore, we can observe how many instances of the Duplicable Stage are created by the Adaptation Thread. Thirdly, we look at another 3D diagram showing the total throughput of the Task Farm Stage, again dependent on the Throughput Boundary and the specific point in time. Using the total throughput, we can infer how efficient the Task Farm Stage is, especially combined with the number of stages shown in the previous diagram. After all, a high number of worker stages does not necessarily ensure a high throughput. The fourth and last 3D diagram shows the mean throughput of each instance of the Duplicable Stage, dependent, again, on the Throughput Boundary and the specific point in time. The diagram shows if the individual worker stages have a different throughput, i.e., if the efficiency varies depending on the current amount of stages.

**CPUBAL Scenario**

In Table 5.6, the parameters used for the CPUBAL scenario on the INTEL system for the detailed analysis are listed.

<table>
<thead>
<tr>
<th>count</th>
<th>problemSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEL</td>
<td>10000</td>
</tr>
<tr>
<td></td>
<td>15000</td>
</tr>
</tbody>
</table>

Figure 5.9 shows the evaluation results for the CPUBAL scenario. In Figure 5.9a, we see that the Regression Algorithm (blue) is most efficient at a Throughput Boundary of 0.025. At that point, the Task Farm Stage achieves a speedup of about 18 compared to single-threaded execution (red). That speedup is a very good result for INTEL, especially compared to the speedups shown in Section 5.3.1.

Let us now look at the number of stages during execution in Figure 5.9b. For a Throughput Boundary up to 0.025, the Adaptation Thread parallelizes up to the theoretical limit of 30 worker stages (32 threads of INTEL minus Distributor and Merger), as illustrated by the red color. For higher Throughput Boundaries, the amount of parallelization rapidly decreases. For example, at a Throughput Boundary of 0.05, the Adaptation Thread only creates around 10 stages (yellow).

Additionally, we see that the computation time for a Throughput Boundary up to 0.025 is the same as the optimal computation time in Figure 5.9: the diagram stops at around 6 seconds. Otherwise, the computation time is much longer, which is, however, not shown.
5. Performance Evaluation

(a) Computation time

(b) Analysis of the number of stages

(c) Analysis of the total throughput

(d) Analysis of the mean throughput

Figure 5.9. Detailed behavior of the CPUBAL scenario
in the diagram for more clearness in the diagram. An additional point is that for higher Throughput Boundaries, the Task Farm Stage at first creates, like always, new worker stages. However, these worker stages are then, after some time, removed again. Therefore, only one worker stage executes all remaining elements, which is not efficient.

Figure 5.9c shows the total element throughput of the Task Farm Stage per 50 ms (which is the adaptationWaitingTimeMillis). As we can see, Figure 5.9c looks similar to Figure 5.9b. Therefore, we can conclude that for the CPUBAL scenario, a higher amount of stages always provides a positive impact on the throughput of the Task Farm Stage. The highest throughput is reached whenever the highest number of stages are in use.

The last diagram for the CPUBAL scenario is shown in Figure 5.9d. It shows the mean throughput of the worker stages per 50 ms. Whenever comparing this diagram to the other three diagrams of the CPUBAL scenario, we have to pay attention to the axes. To be able to view all the data of this diagram without overlapping surfaces, we exchange the axes for the Throughput Boundary and the computation time.

Looking at Figure 5.9d, we see that the mean throughput is high for Throughput Boundaries greater than 0.025. Remembering the other diagrams of CPUBAL, we see that the individual throughput of each stage becomes higher the less worker stages exist. For our optimal Throughput Boundary of 0.025, the mean throughput lies around 3 (yellow), while the throughput for higher Throughput Boundaries, after an initial period, lies around 6 (red). However, as we can see in Figure 5.9c, the mean throughput of the worker stages is not lowered so much that parallelization becomes inefficient. After all, the total throughput still increases by adding more worker stages.

**CPUUNBAL Scenario**

In Table 5.7, the parameters used for the detailed analysis of the CPUUNBAL scenario on the INTEL system are listed.

| count | INTEL | 25000 |

In Figure 5.10a, the computation times for the CPUUNBAL scenario are shown. Like the behavior of CPUUNBAL observed in Section 5.3.1, the Regression Algorithm (blue) provides the best performance at a Throughput Boundary of 0.0. At that point, the speedup compared to single-threaded execution (red) lies around 14, which is a good result for INTEL.

Figure 5.10b shows the behavior of the Adaptation Thread concerning the number of stages. For a Throughput Boundary of 0.0, the maximum amount of stages are used for the whole computation (red). For Throughput Boundaries up to 0.05, the Regression Algorithm still recognizes the need for parallelization, but the worker stages are removed after some time again. At higher Throughput Boundaries, hardly any parallelization occurs (blue).
5. Performance Evaluation

Figure 5.10. Detailed behavior of the CPUINBAL scenario
5.3. Results and Discussion

Figure 5.10c shows the total throughput of the Task Farm Stage per 50 ms. Since the CPUUNBAL scenario requires more computation time per element the more time has elapsed (see Section 5.2.2), the total throughput decreases the longer the execution goes on. However, we can still see that for a lower Throughput Boundary, the total throughput is generally higher compared to higher Throughput Boundaries.

The mean throughput per 50 ms as shown in Figure 5.10d behaves analog to the total throughput. The higher the elapsed computation time, the lower the mean throughput becomes because of the higher computation complexity of CPUUNBAL. However, we also see that the mean throughput for lower Throughput Boundaries is generally lower (blue) than for higher Throughput Boundaries. But as we can see in Figure 5.10c, this lower mean throughput is still not low enough to decrease the total Task Farm Stage throughput whenever the number of worker stages is high.

**IO Scenario**

In Table 5.8, the parameters used for the IO scenario on the INTEL system for the detailed analysis are listed.

<table>
<thead>
<tr>
<th>count</th>
<th>numChars</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEL</td>
<td>20000</td>
</tr>
<tr>
<td></td>
<td>5000000</td>
</tr>
</tbody>
</table>

As we can see in Figure 5.11a, the Regression Algorithm (blue) provides the most efficient behavior for the IO scenario and a Throughput Boundary of 0.04. At that Throughput Boundary, we achieve a speedup of 16 compared to single-threaded execution (red). Especially for the IO scenario, this is a very good result.

Figure 5.11b shows the number of worker stages in the Task Farm Stage. Up to a Throughput Boundary of 0.05, the maximum number of worker stages is used (red). For higher Throughput Boundaries, the amount of parallelization decreases rapidly (orange and blue).

The diagram of the total throughput per 50 ms shown in Figure 5.11c looks, like in the CPUBAL scenario, similar to the worker stage diagram (Figure 5.11b). This similarity shows, again, that the parallelization always improves the total throughput of the Task Farm Stage in the IO scenario.

The mean worker stage throughput (Figure 5.11d) shows, as in CPUBAL, that, after an initial amount of computation time, the mean throughput is always higher if less worker stages exist. For the least amount of worker stages at a Throughput Boundary of 0.075 or higher, the mean worker stage throughput lies around 5 elements per 50 ms. In contrast, for a Throughput Boundary of 0.05 or less, the mean throughput only lies around 3. However, like in the previous scenarios, the decrease in mean throughput is not grave enough as to make parallelization inefficient.
5. Performance Evaluation

![Performance Evaluation Diagrams](image)

(a) Computation time  
(b) Analysis of the number of stages  
(c) Analysis of the total throughput  
(d) Analysis of the mean throughput

**Figure 5.11.** Detailed behavior of the IO scenario
5.3. Results and Discussion

**CPUIO Scenario**

In Table 5.9, the parameters used for the last scenario IO are listed.

**Table 5.9.** Used parameters of scenario CPUIO on INTEL for the detailed analysis

<table>
<thead>
<tr>
<th></th>
<th>count</th>
<th>numXmlLines</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEL</td>
<td>10000</td>
<td>2000</td>
</tr>
</tbody>
</table>

At last, we discuss the detailed behavior of CPUIO. In Figure 5.12a, we see that the
5. Performance Evaluation

Regression Algorithm (blue) is most efficient at a Throughput Boundary of 0.02. At that Throughput Boundary, we achieve a speedup of around 8.

In Figure 5.12b, the maximal number of worker stages (red) is used for a Throughput Boundary up to 0.025. For higher Throughput Boundaries, the amount of parallelization quickly decreases (yellow and blue).

Like in CPUBAL and IO, the diagram for the total throughput per 50 ms (Figure 5.12c) is similar to the diagram showing the amount of existing worker stages (Figure 5.12b). Therefore, the CPUIO scenario profits from parallelization up to the maximal number of worker stages. After all, the total throughput reaches its maximum for the Throughput Boundaries which induce the highest number of worker stages.

The mean worker stage throughput per 50 ms for the IO scenario is shown in Figure 5.12d. In the diagram, we, again, observe that the mean throughput suffers from a high number of worker stages (blue), leading to a throughput of around 2. In contrast, for Throughput Boundaries higher than 0.05, the mean throughput lies at around 5. However, this lower mean throughput is not significant enough to decrease the total throughput enough to make parallelization inefficient.

Insights from the Detailed Analysis

In the detailed analysis, the parallelization behavior of the Task Farm Stage depends heavily on the Throughput Boundary. The behavior can divided in two cases.

For the first case, the Throughput Boundary is low enough to ensure an optimal amount of parallelization for the whole computation time, as illustrated in Figure 5.13. In the diagram we see the throughput per 50 ms of each added worker stage at a Throughput Boundary of 0.0. As we can observe, excluding some irregularities, each worker stage provides a consistent throughput from its creation to the end of execution. Since the input elements are all correctly distributed to all existing stages, the resulting computation time lies around 7 seconds.

In contrast, Figure 5.14 shows an example for the second case with the same workload as in the first case. However, this time we use a Throughput Boundary of 0.5. At the start, some parallelization occurs up to 5 additional worker stages. Since the Throughput Boundary is too high for this scenario, the Adaptation Thread decides not to parallelize further. However, afterwards, the stages are removed again until there is just the basic worker stage left at about 4 seconds. Consequently, the basic worker stage has to execute all remaining input elements on its own, raising the computation time to ca. 80 seconds.

In Section 4.2.3, we explained that the Adaptation Thread, after finishing with the parallelization, continuously removes worker stages whose input pipes are empty. Otherwise, worker stages are not removed. Therefore, we can conclude that the Dynamic Distributor with its NonBlockingRoundRobinStrategy fails to distribute elements to all worker stages such that empty pipes are avoided. If the pipes were not to become empty, the 5 created worker stages would exist until the whole execution is completed. While the usage of 5 worker stages does not provide an optimal parallelization (after all, the Throughput Boundary is too
5.3. Results and Discussion

![Stacked diagram showing the throughput of each added worker stage for CPUBAL at a Throughput Boundary of 0.0](image)

Figure 5.13. Stacked diagram showing the throughput of each added worker stage for CPUBAL at a Throughput Boundary of 0.0

high), retaining the stages until the end of execution would still highly decrease the total computation time. In Section 7.2, we discuss a distribution strategy which ensures a better runtime behavior of the Adaptation Thread.

Overall, there are a few more insights we can infer from the detailed analysis. For one, we have seen that a Throughput Boundary of 0.0 does not always provide the best performance for all scenarios. The reason is that, as we have discussed in Section 3.2, the computational overhead increases with higher Throughput Boundaries. After all, the Task Farm Stage then adds new stages, even if the performance is not improved because of it.

Furthermore, we could observe a much greater speedup than in the examples in Section 5.3.1. Using a well-performing Throughput Algorithm like the Regression Algorithm on a modern system shows that the Task Farm Stage is able to provide good performance results with its parallelization. However, the resulting computation time highly depends on the
Figure 5.14. Stacked diagram showing the throughput of each added worker stage for CPUBAL at a Throughput Boundary of 0.5

correct configuration for the underlying system. Additionally, lowering the pipe capacity and count parameters resulted in a evener distribution of input elements to the instances of the Duplicable Stage. Those changes also have a positive impact on the computation time, since a good performance of the Task Farm Stage requires an even and efficient element distribution.

5.4 Threats to Validity

As in most evaluations, there are some threats to validity we have consider. For one, only four test systems are available for our evaluation. There is a chance that there exist some systems where the Task Farm Stage behaves completely different from the behavior shown
Another point is that we might not have used enough test scenarios. While we tried to cover every possible case (CPU-intensive, IO-intensive, and mixed workload), there might still be scenarios where the behavior of the Task Farm Stage differs. For example, to generate an I/O workload, we read from and write to a hard drive. Another example of I/O workload would be networking access. While nothing in the architecture of the Task Farm Stage points to different behavior by using networking as I/O, we cannot completely discount the possibility of different behavior.

Similarly, we used certain parameters for the test scenarios and for the configuration of the Task Farm Stage. While the chosen parameters all gave plausible results, we also cannot discount the possibility that a change in the configuration of the Task Farm Stage would greatly affect the behavior. That being said, the evaluation results generally conform to our expectations discussed in Chapter 4. Therefore, while parameter changes might affect the resulting performance, we are unlikely to observe a completely different behavior as seen in this evaluation.
There are multiple Pipe & Filter frameworks available besides TeeTime. FastFlow [FastFlow; Aldinucci et al. 2011] is a Pipe & Filter framework for the C++ language. Like TeeTime, FastFlow also uses pipes and stages as high-level elements. It also supports the Task Farm pattern. However, the Task Farm in FastFlow does not support any self-adaptive behavior like our approach. A fixed number of worker stages has to be specified at Task Farm creation. The Task Farm then utilizes exactly the specified amount of worker stages for the whole execution\(^1\).

Pipes is another Pipe & Filter framework\(^2\). It is, like TeeTime, a framework written for Java. However, in contrast to TeeTime, it does not support automatic thread management, therefore making it inappropriate for the implementation of the Task Farm pattern. In fact, it currently does not support the Task Farm pattern.

Another Pipe & Filter framework is the Apache Commons Processing Pipeline\(^3\). The Apache Commons Processing Pipeline is written for Java and does support several multithreading methods. However, it has not been updated since 2009 and is, therefore, outdated. It currently does not support the Task Farm pattern in any way.

There are also other approaches we can use similar to the Task Farm pattern. For example, MapReduce [Dean and Ghemawat 2008] is a programming model. It first requires a map function, mapping arbitrary key-value pairs to intermediate key-value pairs. Lastly, it requires a reduce function, merging the intermediate key-value pairs to a possibly smaller set of values. An example would be a word count of documents. The map function splits document name/content pairs to word/occurrence pairs. The reduce function would add all occurrences of a word together. Therefore, we gain the functionality to count the number of occurrences of words over multiple documents. If we use multiple threads as map-worker and reduce-worker, we can achieve concurrency. The MapReduce approach is more general than the Task Farm pattern, since MapReduce does not require Pipe & Filter architectures. However, MapReduce is more complex to efficiently parallelize since it possesses two types of worker stages in contrast to the Task Farm: the map-worker and the reduce-worker.

Furthermore, it is also possible to implement the Task Farm pattern in a self-adaptive manner using the Message Passing Interface (MPI) [González-Vélez 2006]. However, MPI generally serves the purpose of distributed computing. A Pipe & Filter framework like

\(^1\)http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:tutorial (visited on 10/22/2015)
\(^2\)https://github.com/tinkerpop/pipes/wiki (visited on 10/22/2015)
\(^3\)http://commons.apache.org/sandbox/commons-pipeline (visited on 10/22/2015)
6. Related Work

_TeeTime_ specializes in computing on a single computing node.

There are also other computation models than _Pipe & Filter_. The _Actor Model_ [Agha 1986] describes _actors_ connected by communication lines to create an _actor system_. An implementation and development tool of the _Actor Model_ is _Ptolemy II_\(^4\).

van Hoorn [2014] discusses approaches for self-adaptive capacity management for component-based software systems in his dissertation. As discussed in Section 2.4, we use part of his _SLaStic_ meta-model to structure our self-adaptation approach.

Suleman et al. [2010] uses the self-adaptation approach to add and remove worker stages on demand. To determine if worker stages should be added or removed, he uses a static training approach. Therefore, the worker stages are measured over some time to observe if performance can be gained by adding or removing worker stages. However, the measurements are static and do not try to predict the future performance of the worker stages. Therefore, the reconfiguration based on the measurements is sometimes inaccurate. In contrast, our approach does not need an additional training phase. Using prediction algorithms and monitoring approaches, we accurately decide our reconfiguration plan during normal execution of the worker stages.

\(^4\)http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm (visited on 10/22/2015)
7.1 Conclusions

In this thesis, we presented an approach to improve the performance of Pipe & Filter architectures by adding support for self-adaptive Task Farms. At first, we discussed the structure of the Task Farm and its components (Chapter 3). We designed the Task Farm and all its related components (including the Task Farm Stage itself, the Adaptation Thread, and the Monitoring Components) on class level. Furthermore, we established and described all configurable parameters of the Task Farm Stage.

In Chapter 4, we explained the behavior of the Task Farm Stage. We discussed the basis traversal of elements through the Task Farm Stage as well as the basic self-adaptive behavior. Additionally, we explained in detail how the components of the Adaptation Thread as well as the Monitoring Services operate.

At last, we implemented the Task Farm Stage in the Java-based Pipe & Filter framework TeeTime and evaluated the performance of the implementation (Chapter 5). We used four different systems and four different scenarios (balanced/unbalanced CPU-intensive, I/O-intensive, and combined CPU- & I/O-intensive workload) to gain an overview of the computation times if the Task Farm Stage is used. Additionally, we did a detailed analysis using the four scenarios on a single system, resulting in a detailed view inside the functionality of the Task Farm Stage at runtime.

In our evaluation, we showed that we gain a significant speedup by using the Task Farm Stage as opposing to single-threaded execution, as long as the configurable parameters of the Task Farm Stage are appropriately set. In most cases however, the Task Farm Stage never decreases the performance compared to single-threaded execution, regardless of the parameters. Additionally, we also show that the Distributor cannot optimally distribute elements to the worker stages in all cases, resulting in suboptimal computation times for the Task Farm Stage.

7.2 Future Work

For future work, it is useful to implement a more suitable distribution strategy for the Task Farm Stage. In Section 5.3.2, we discussed that the current distribution strategy does not always optimally distribute the elements to the worker stages. Ideally, we would need a
distribution strategy which distributes elements always to the output pipe with the least buffered elements.

Additionally, we see potential to automate the configuration of some parameters of the Task Farm Stage. The `adaptationWaitingTimeMillis` (see Section 3.2) does only need to by as high as the computation time of one element in the worker stages of the Task Farm Stage. Therefore, it is possible to compute the parameter by using static analysis of the Java byte code (for example, via Soot\(^1\) [Vallée-Rai et al. 1999]) and by analyzing the underlying hardware.

Furthermore, it might be possible to automatically detect a suitable Throughput Boundary by analyzing the throughput of the worker stages. Then we change the Throughput Boundary in a self-adaptive way depending on the throughput measurements, similar to the amount of worker stages in the Task Farm Stage.

It is also possible to improve the functionality of the Task Farm Stage. One way is to add multiple input and output ports to the worker stages. Therefore, we also need as many Distributors and Mergers as there are input and output ports, respectively. Each Distributor and each Merger can then have their own strategy, resulting in an improvement of applicability of the Task Farm Stage.

Currently, it is only possible to use multiple Task Farm Stages concurrently by reducing the maximum amount of cores they may use. It is useful to implement a coordinator component, providing a turn-based mechanism to coordinate the Adaptation Threads. For example, we might use a strategy that alternates the parallelization of multiple Adaptation Threads. Another strategy would be to ensure that only one Adaptation Thread is in ADDING mode at once (see Section 4.2.3).

We used an own implementation of the Monitoring Components for the Task Farm Stage. However, using Kieker [Kieker] is useful to provide more features, a better presentation of measurements, and compatibility to other project also using Kieker.

In regard to the Duplicable Stage, we currently use an interface which the framework user has to implement for Task Farm Stage usage. A clearer approach might be a more declarative implementation of the Duplicable Stage. A class annotation can be used to mark a stage that is to be parallelized by a Task Farm Stage. Furthermore, we can introduce further annotations for the class attributes to determine duplication behavior. For example, one annotation could mean that the value of the attribute is to be copied on duplication. Another annotation could result in a copy by reference or a new instantiation of the object. Therefore, the framework user would not need to directly implement the duplication behavior of the Duplicable Stage.

At the moment, the Task Farm Stage always employs exactly one worker stage before the self-adaptation process begins. It would be useful to be able to choose how many worker stages are available at the beginning of execution. As a result, the time until the optimal number of worker stages is found is reduced if the starting amount of worker stages was chosen appropriately.

\(^1\)http://sable.github.io/soot/ (visited on 10/22/2015)
Bibliography


Bibliography


