Self-Adapting Execution of Pipe-and-Filter Systems

Master's Thesis

Marc Adolf

March 23, 2016

Kiel University
Department of Computer Science
Software Engineering Group

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

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Kiel, _____

Abstract

To improve the performance of *pipe-and-filter* applications several steps can be taken. One of these is the utilisation of different computational resources, like memory or CPU cores. Especially the modular design can be used to employ parallelisation techniques and distribute threads to certain filters. This resource distribution in *pipe-and-filter* systems is a difficult issue. Often it has to be handled manually or the same automatic strategy is used for every application. Especially input dependent computational efforts and a dynamic behaviour of the system that change at runtime can't be handled well with these strategies. Following the *MAPE-K* approach, we present a realisation of the required effectors and sensors in the used execution model, the push-model. These are utilised to create a control loop and several service classes similar to their, so called, autonomic manager. Our implementation can be used with different customisable strategies and monitored properties. We extended the *TeeTime* pipe-and-filter framework to implement our approach. We also evaluated the feasibility, the overhead, and the first implementations of different policies. Thereby, we show, that our extension can change the system at runtime and influence the performance. The measured overhead is small but varies between tested systems. Our implemented strategies show different behaviours but are not able to improve the performance in our first test scenarios. In a second evaluation with different benchmark scenarios the performance can be clearly improved.

Contents

Chapter 1

Motivation

In the field of stream processing the inputs of the applications consists of multiple elements [Hormati et al. 2009]. The computation applied to each of them often includes different independent steps. Hence, the structure can be divided into these steps, which each element has to pass through. Thereby, each station uses the output of the preceding one as input and applies the same operation to every element. This results in a network of separate computational steps, called filters. The connection between the single filter can range from simple function calls to complex synchronisation mechanisms. Often the *pipe-and-filter* architectural style [Taylor et al. 2009; Monroe et al. 1996] is used to model such problems in a modular way. Especially in the context of Big Data or applications with continuous data streams, like system monitoring, this is a commonly used architecture [Burtsev et al. 2014; Wulf et al. 2014]. For example the monitoring framework *Kieker* [van Hoorn et al. 2012; 2009, b] uses an underlying *pipe-and-filter* architecture to analyse the measured system states. Enabled by the low coupling of the single functions the user can easily choose which part of the analysis should be done and which algorithms should be used to treat the data.

Often some or all filters are executed separately by processing units, for example threads or processes, as soon as enough input is available. Since the single processing steps are dependent on each other, a slow filter slows down the whole system. Additionally the execution time and the computational effort of each filter may change if different inputs are given. Especially if we consider branches in the network of the *pipe-and-filter* architecture that represent if-statements or even loops, the computational effort can be further scattered. A simple example may be the handling of two CSV files. An if-statement is used to process only certain data. One only contains entries that don't trigger such an if-statement and the other one has multiple of such entries. For these reasons it is difficult to distribute the available resources to the single filters in an optimal way. Often this is done manually by the user before the execution. This can already fail in the case where different inputs create different computational efforts. If we consider continuous data streams, these computational efforts can even change during a single execution. This makes a good *static* distribution beforehand more difficult or even impossible. Additionally the behaviour of the system may change depending on the underlying hardware and operating system.

Frameworks for *pipe-and-filter* architectures like *TeeTime* [Wulf et al. 2014; Wulf and Hasselbring 2016] provide the possibility for the user to build systems in this architectural style. To support this, the executing system, pipes and basic filters are already given. Here filters are often called stages. The user only needs to connect those and may build own

stages, if needed. Currently in *TeeTime* the challenge to create a good resource distribution is mostly left to the user. Only if the execution model requires it, threads are automatically assigned to stages. *Dynamic* resource assignments that change during the execution are not supported.

1.1 Goals

There are many approaches which adapt certain *pipe-and-filter* and stream processing systems. Most of them provide a solution for their special use cases and underlying systems. Some of them use threads or CPU cores to optimise the execution. Others increase the caching or employ further methods. We will use the *TeeTime* framework as a reference implementation and extend it to reach our goals. In this work we ultimately provide the instruments to enable a general purpose *pipe-and-filter* framework to be automatically and dynamically adapted at runtime. For this purpose we define the following goals.

1. We will enable a general purpose framework to change the resources distributed to stages at runtime. Here we limit the type of resources to processes or threads that are assigned to stages. Therefore, a thread should be able to be assigned to a stage at runtime and to be withdrawn from it. Additionally, we want to be able to collect information of the behaviour from the single stages and the system as a whole during the execution. For example we want to observe if a certain stage may be the bottleneck of the system.

2. Since we want to adapt the framework at runtime and to be able to react to certain stages of the system, we combine both functionalities of the first goal. Therefore, our second goal is to use these instruments and create an extension which can dynamically adapt the used framework. Thereby, we want to be able to collect the measured observations and analyse them. With this data we want to understand how the system behaves and where we can improve it. Again, these steps should be done during the execution. After deciding where the application can be improved, we want to use our methods for resource distribution to optimise the system. The measuring, analysing, planing and execution of the desired changes will be done similar to an autonomic managing system described by [Kephart et al. 2003; Horn 2001]. For this purpose we will build an extension for *TeeTime* and only change the original code if it is necessary. The current behaviour should be preserved and might be optional in the future. Since there exist many other approaches and no one seems to be clearly the best, we also want to design our system to enable an easy replacement of the used resource distribution algorithms. In this initial work we don't expect to find the best algorithm for this problem.

3. Our last goal is to evaluate our approach. Thereby, we want to show the feasibility of our implemented methods and whether we could reach the first two goals. During the

implementations operations are added to *TeeTime*, which influence the performance of the framework even if the new control loop is not used. Therefore, we want to measure the overhead of these operations. Hereby, the original implementation and our approach will be compared. Especially monitoring may create such overhead, if it can't be turned off. In the last part of the evaluation we want to measure how our extension influences the execution in different situations. Thereby, varying distribution strategies can be applied to compare them.

1.2 Document Structure

In the following, in Chapter 2 we present the *pipe-and-filter* architectural style and give an overview how *TeeTime* implements it. Furthermore, *autonomic computing systems* are introduced. In Chapter 3 it is discussed how this autonomic behaviour can be transfered to our reference framework *TeeTime*. Thereafter, in Chapter 4 we describe how we realise the change of the used resources, and therefore the resource distribution, at runtime and which problems may occur. In the next Chapter 5 we show how we gather the monitoring data and display the design of our extension, described in the second goal. After this the results of the evaluation are presented in Chapter 6. At last in Chapter 7 we give an overview of the related work and summarise this work in Chapter 8. Additionally, we present options for future improvements of our extension.

Chapter 2

Foundations and Technologies

In this chapter we first introduce the *pipe-and-filter* architectural style, which we want to extend with an self-adaptive management. After this, we display how *TeeTime* implements this architecture to create a general purpose framework, which allows users to build applications in the *pipe-and-filter* style. We use this framework and will extend it later on. In the last part we introduce an approach to build *autonomic computing systems*.

2.1 The Pipe-and-Filter Architectural Style

In the field of stream processing data from a stream consists of several elements. Often they are manipulated element by element. This computation may require many different steps per element. Some of these steps may be optional. Each single step may provide its own special functionality. These steps are reused for every received element and the stream processing as a whole is centred around the composition of these applied manipulations. We call these steps *filters* or, like in *TeeTime*, *stages*. Each stage may receive elements from its predecessor, processes some of them and then sends the processed items to its successors, if it has some. Since the system behaves like a pipeline with different processing stages, the connectors between them are named *pipes*. This architectural style is therefore called *pipe-and-filter* [Taylor et al. 2009; Monroe et al. 1996].

Often Unix commands connected by Unix pipes are presented as a well known example. An illustrative model is displayed here:

```
history | grep "sudo" | grep "install"
```

These three connected commands already represent a simple *pipe-and-filter* architecture. The data processing steps, the Unix commands, are the stages. The connectors (`|`) are the already mentioned pipes that transfer the output of the predecessor to the successor. The stages in this example are *history*, which is the data source of this system, and two instances of *grep*. At the beginning the saved commands of the user are read by *history*. They are then transferred by the pipe to the first *grep* instance. With the parameter “sudo” *grep* outputs only lines where the *sudo* command was used. The second *grep* receives only these remaining lines. The previous data was hence filtered in the second stage. Now only lines where the word “install” was used are kept and printed. In total this system searches

for every time where the *sudo* command was used in combination with *install*. These example can be further extended. The used stages can be replaced without considering the specifications of the other connected stages. Naturally this may change the outcome of the system.

In Figure 2.1 the example from above is shown in an alternative way to visualise the architecture. Here we illustrate it like a directed graph, with boxes representing *filters* and arrows as *pipes*. Intuitively the direction of the data flow is the same as the direction given by the arrows. This representation also illustrates the advantage of the natural low coupling in *pipe-and-filter* architectures. The only connection between the single modules is the data transport through the pipes. Due to this properties building a *pipe-and-filter* architecture can remind of constructing a structure with "Lego blocks", especially if a drawn graph is used to comprehend or design the system [Taylor et al. 2009]. This style of comprehension can be similar to other modelling tools in analogous field like *Ptolemy 2* [Dept. 2016].

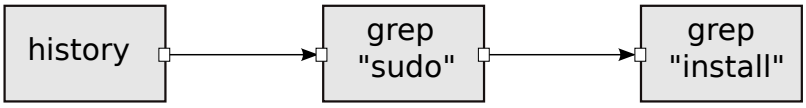


Figure 2.1. A Pipe-and-Filter configuration drawn as a graph

A *pipe-and-filter* architecture can contain any number of stages. These stages can be extended with multiple input and output pipes. In common architectures with no restrictions to the processed data elements, the connection may be limited to stages by their respective produced and received data types. Even feedback loops can be formed to represent more complex systems, like data loops [Wulf et al. 2014]. Through the strong modular design of the stages, the transport is entirely done by the pipes. The stages don't need to know more than, how to get new elements and how to send the processed ones. Therefore, it is even possible to simply distribute stages over multiple servers and just connect them with an appropriate pipe. One can say that, regarding the stages, pipes are like wormholes where items come by and where the finished products can be dumped.

The processed data stream itself can be finite or even continuously. In both cases it can be advantageous to let some of the stages be computed by separate processes or threads. Therefore the execution of a *pipe-and-filter* architecture divides the stages into active and passive ones. While the passive stages don't start acting on their own, active stages do run on their own and trigger connected passive ones. Thus, each active stage has its own thread. If a passive stage is invoked the thread is also used to execute the passive one. Often single stages are also duplicated to improve the throughput of the system. During the duplication the architecture needs to deal with state variables which potentially increases the coupling. Many implementations choose to avoid these synchronisation problems by restricting the stage to be stateless, like [Welsh et al. 2001].

There exist different strategies for the execution of *pipe-and-filter* architectures. They differ mainly in the chosen direction of communication. Therefore, their functions specify

how the control flow will propagate. One strategy is called the *push-model*. In this model we start with the producer stages, which create the initial elements. In some implementations like in *TeeTime* [Wulf and Hasselbring 2016] every producer in the *push-model* has to be active. As soon as an element is produced and sent to one successor, this successor is invoked. In case the succeeding stage is passive, the thread of the sending stage will execute the receiver. This is continued, if necessary, as long as the successor is passive or the last step of the processing was done. If the following stage is active, its owning thread will process the element. After this is done the new thread behaves like the producer thread. In general, we can say that stages that are at the beginning of the data flow, invoke their successors as soon as they can provide an element. Elements are pushed from the beginning to the sink.

The second strategy is the *pull-model*. Here the data sinks at the end of the data flow, are the active parts. As soon as it needs an element, the sink will invoke a predecessor, executing it if its passive. Active stages behave analogous to the data sink, until a producer is reached. Elements are pulled out of the stages by the sinks.
Another strategy is to mix these two procedures. Active stages that need more elements may pull them from their predecessors and push the result through their passive successors [Buschmann et al. 1996].

In *pipe-and-filter* architectures some stages can create a bottleneck. "A bottleneck can be described as an area (one or more components) where the request arrival rate is higher than the outgoing rate" [Michiels et al. 2002]. The stage that has the lowest ratio is the bottleneck and thus slows down the whole system. It does not produce enough elements for its successors to run efficiently and its predecessors may be blocked due to full pipes. As a simple example in the *push-model*, assume that we have a *pipe-and-filter* architecture in which each stage is executed in a different thread and each stage has one input and one output pipe. Each stage can only process as many elements and as often as it receives them from his predecessor. Hence if a stage is noticeably slower than the others, the successors are slowed down by it. If we also consider a maximum buffer size in a pipe, even the predecessors are limited by it, waiting for the pipe to have room for more elements. This bottleneck stage is obviously slowing down the rest of the system. Since the neighbouring stages are slowed down to the same throughput, this cascades through the whole system. Assigning more threads to the bottleneck may increase the overall performance. It is not always possible to optimize the program, e.g. if the data access is limited. Additionally adding more threads than necessary can be a waste of resources and may even hurt the performance [Suleman et al. 2010; Soulé et al. 2013]. This is similar to project management where the longest path in a Gantt chart describes the critical path of a system. If this path is slowed down the whole project is delayed and vice versa.

There also exist different techniques to optimise stream processing. Two examples are the *fusion* and *fission* of stages. In the process of stage *fusion* two successive stages are fused to generate a single new stage which combines the processing steps of these two. This can be done to reduce communication overhead or to simply free resources. The opposite

technique is the *fission* of a stage. Here some of the processing steps, a stage employs for a data element, are sourced out to a new stage. Hereby the computational effort of a single stage is split into more components [Hirzel et al. 2014]. *Pipe-and-filter* architectures can be used for different data processing tasks, like streaming, compression or system monitoring [Suleman et al. 2010; Wulf et al. 2014].

2.2 TeeTime: a Pipe-and-Filter Framework in Java

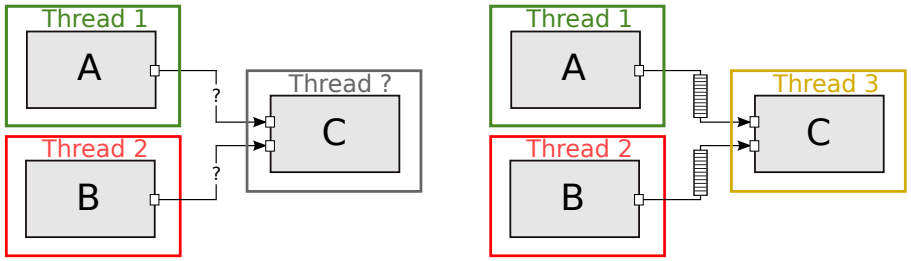
TeeTime is a *pipe-and-filter* framework for Java [Wulf and Hasselbring 2016]. The framework allows the user to build systems in the *pipe-and-filter* architectural style. The main idea is to provide a framework for stage developers and users who employ these stages. A system should be easy to build, like the "Lego blocks" mentioned before. For this, *TeeTime* already provides some predefined basic filters. For example a merger that merges two data streams by a certain merge strategy and provides this merged data stream is such a basic filter. Additionally individual filters can be easily built to meet the needs of the user. The execution itself is handled by the framework. Only has to be started by the user. In *TeeTime* filters are called *stages*. In the following we refer to the entire *pipe-and-filter* architecture as a *configuration*.

TeeTime implements the *push-model*. It restricts every producer stage to be executed by its own thread. Therefore a producer can never be passive. The framework enables the user to distribute the computational effort of the stages on different threads. For this purpose, a stage can be manually set as active, which means that it will be executed in a new thread. By default only the producers will be active at first. Due to the *push-model*, the other passive stages are executed by the same threads like their predecessors, which invoke them. Thereby the computational effort of the threads may not be evenly distributed among them.

In *TeeTime*, there exits a second condition in which a stage has to be declared as active. In Figure 2.2a a simple example scenario of this special case is displayed. **Stage A** and **Stage B** are producer stages and hence run by their corresponding threads. Now the framework has to decide, who is responsible for **Stage C**. Imagine that **Stage C** should only process elements from **Stage B** after it has received the first item from **Stage A**. The first strategy is to give only one of the producer stages the possibility to execute the third stage. If we simply follow the definition from the last section, the stage is only executed as often as one of the producers sends elements. In this situation not all elements will be processed. Even if preparations are taken to avoid this, another problem arises if the responsible producer finishes early.

The second strategy is to allow both threads the execution of **Stage C**. Since the stage may be user-defined, the synchronisation may cause even more problems and may slow down the system. The third strategy and the one used by *TeeTime* is to create a new thread and assign it to **Stage C**. Now all pipes can be frequently checked and the stage can be finished if both producer stages signal their end. The resulting situation is shown in Figure

2.2b. Here every stage has its own thread and the pipes are buffered and synchronized. In *TeeTime* every stage that has more than one predecessor thread, has to be active. In a *pull-model* one would have a similar problem with multiple succeeding threads.



(a) Unclear responsibility for stage C **(b)** Solution in TeeTime for stage C

Figure 2.2. Issues arising with different preceding threads in *TeeTime*

As indicated by the last problem, stages communicate via different types of pipes. In *TeeTime* a port connects one stage with one pipe. Both elements are initialised with fixed type parameters. Especially the ports enable the support of type safety. Each stage may have multiple input and output ports. To connect the ports of two stages and thereby the owning stages, the user only needs to use the given method to create this connections. The type of these pipes can currently be grouped into two classes. The first class of pipes is used for the intra-thread communication. It is represented by the *UnsynchedPipe*. This pipe transports the newest added element and forces the adding thread to execute the target stage of the pipe. Thereby the element is immediately removed again. The second class of pipes is used to handle the inter-thread communication. Here a bounded variant, the *BoundedSynchedPipe*, or an unbounded but slower alternative, the *UnboundedSynchedPipe*, is used.

The framework initialises all pipes with *DummyPipes* and replaces them later on. Before the execution starts, the *pipe-and-filter* architecture is traversed. Thereby, if necessary, new threads are created and assigned to stages. In a second visit the connecting *DummyPipes* are replaced by the appropriate variant. After this, the framework starts the execution and the starting signal is sent and thus propagated through the stages. Thereby *TeeTime* treats stages, pipes and the whole configuration as the core elements that are employed [Wulf et al. 2014; Wulf and Hasselbring 2016]. The execution of the configured *pipe-and-filter* architecture will than be handled automatically by the framework.

Imagine a configuration built with *TeeTime*. Again we can represent this system as a connected and directed graph. Since all producer stages and stages with multiple preceding threads cause a new thread to be created and assigned, we can partition the graph in different connected components. We create a *partition* for every thread in the system. Thereby each stage in a subgraph can be reached and it can only be part of exactly one *partition*. In the future we will often refer to the described subgraphs as *partitions* of such

a *pipe-and-filter* architecture. Naturally there exist exactly as much *partitions* as there are active stages in the system.

2.3 Autonomic Computing Systems

Often optimising a system to improve its performance can only be controlled through manual or semi-automated interfaces. It requires the user to manually gather information about the program behaviour. Consecutively the user has to find the right settings trying and measuring again. This is basically a trial-and-error approach, optimized by experience and by insight knowledge of the program. In a *pipe-and-filter* architecture especially the stages are designed to be reusable. Their behaviour may not be fully known to the user. Therefore, an optimal usage is probably not even possible without extensive simulation runs. Additional issues arise if the computational effort can change in an unpredictable way during the computation. Contrary to this issues, there exist systems in the nature that are able to automatically adapt themselves and their components to the current situation. This enables the organisms to perform unconscious reactions to internal and external influences. One famous example is the autonomous nervous system, which controls many organs, or systems, in the body and allows the regular operation and cooperation of many components.

In computer systems mechanisms are often used to regularly check the behaviour of software components and to adapt them if needed. Thereby, the need for human interaction should be minimized. Such a mechanism can be utilised in many different fields. Hence a general purpose architecture for "autonomic computing systems" was introduced by [Horn 2001; Kephart et al. 2003]. They define it as "a computing system that senses its operating environment, models its behaviour in that environment, and takes action to change the environment or its behavior"[Horn 2001]. This approach is also called the *MAPE-K control loop*, or just *MAPE-K* [Bruni et al. 2012].

In general such a system tries to keep the different observed and executed program parts on course of a given policy. Every time the mechanism discovers a deviation to the pre-calculated behaviour it tries to react to it and to restore a state according to the policy. We can summarise that an autonomic, or self-adapting, system reconfigures itself to meet its own needs. The decision to adapt the system may arise by events or through frequent measurements of critical properties. There are various approaches to create flexible systems, like described in [van Hoorn 2014; van Hoorn et al. 2009a] or [Weyns and Holvoet 2007], which can also be applied in the *pipe-and-filter* context.

The main component of such an adaptive system is the control loop. Through this loop the autonomic mechanism regularly analyses the system state and initiates changes. The execution time of each loop iteration, how fast the system reacts, and how often it attempts to change some properties are influences how often and how fast the system can react. Dependent on the purpose of the adaptive program, a fast and reactive loop can be needed, but it may also be necessary to delay it. This way potential overhead can be reduced and

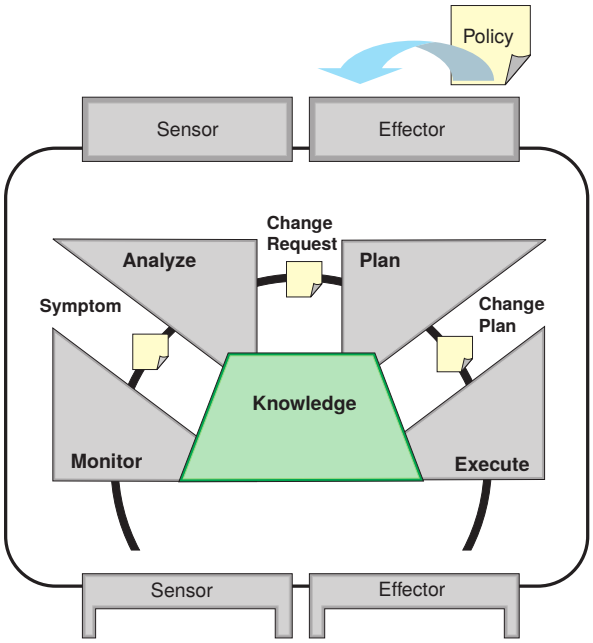


Figure 2.3. The MAPE-K Autonomic Manager [Kephart et al. 2003]

the computation may be more stable.

The computation of the loop can be split into two general phases. In the first phase we analyse the system. This analysis represents the current state of the observed program. Now this data can be used to compare the real behaviour with the expected one. In the second phase the next actions are planned and applied. If the control loop decides that intervention is needed, the planned actions are executed. After this two phases are completed, the loop begins anew with data or events from the changed system [Kephart et al. 2003].

Figure 2.3 shows a more detailed version of the described *MAPE-K* approach. Here an *autonomic manager* as a part of the architecture is shown. The analysing and changing phases are further divided. In the beginning of every iteration data about crucial properties is collected. This is done in the **Monitor** part. This data may include the status, the performance and an identifier for the measured resource. In this way the autonomic system is able to "rapidly organize and make sense to this data" [Kephart et al. 2003]. After all data is collected, the loop proceeds to the **Analyze** part. In this part the gathered data is assembled to create the system state. In the next step the state is analysed. It is decided whether some change is needed. Thereby, techniques like time-series forecasting

can be applied to the data. After the decisions are made, what part of the program is not performing well and should be changed, this information is transferred to the next phase of the iteration. During the **Plan** part the course of action is set, which allows the system to reach the goal described in the preceding part. In the last part, **Execute**, this course of action is taken and changes are made to the system [Kephart et al. 2003].

During the adaptation process data has to be gathered to create a feedback by the control loop. This feedback is ultimately returned to the observed components and changes are applied to them. Every managed component provides a *touchpoint*. Through this interface data can be gathered in the monitoring part. Moreover, changes in the execution phase can be applied directly to these components. As Figure 2.3 indicates with the *sensor* and *effector* at the top of the control loop, this *autonomic manager* can be part of bigger systems and can also provide a *touchpoint*. Analogous, the controlled components can also vary. These can be other subsystems, managers or other more or less complex parts. Naturally, even simple parts, like a single classes or threads can be managed. Furthermore, the amount and the type of the controlled system can vary. It may range from a single resource to a complex system of multiple objects. The set can be composed of heterogeneous or homogeneous pieces. At the end of this range a whole business system may be managed.

Autonomic computing systems can be found in many applications and shapes. [Kephart et al. 2003] describe four possible categories: self-configuring, self-healing, self-optimizing and self-protecting. Basically, in all these categories human intervention is reduced to a minimum and the systems adapt themselves during runtime.

The specific area of application is very broad. Amongst other examples are variable cloud applications, real time systems that may react to certain events and even games like Fl0w [Chen 2016], which adapts its difficulty at runtime to the ability of the user.

Chapter 3

Requirements for the Self-Adaptation of Pipe-and-Filter Architectures

The main goal of this work is to enable a system that employs a *pipe-and-filter* to adapt itself during execution. As an example implementation for our approach we choose *TeeTime* [Wulf and Hasselbring 2016]. *TeeTime* is a good example for tasks that can be configured with human interaction, but is not yet automated. The framework allows the user to distribute threads to stages when he builds his architecture. To address this a method called *setActive()* is given by the framework. Further influence is not possible and hence the execution has to be carefully planned beforehand.

Every *pipe-and-filter* architecture may be part of a bigger system, that itself is administered in an autonomic way. In this first attempt for an adaptive extension we focus on a single connected compilation of stages. Our work may later be extended to provide touchpoints for overlaying autonomic managers, described in the *MAPE-K* architecture. Other approaches like the *DMonA Architecture* [Michiels et al. 2002] already implemented such a recursive management for *pipe-and-filter* architectures.

There already exist different approaches to dynamically adapt *pipe-and-filter* architectures from varying use cases. All solutions are specialised on their use case. Most algorithms are designed to optimize the usage of the computational resources, increasing the throughput or saving operations through reducing overhead. Often the approaches decide how many resources are used to execute which stage. The decision which thread or core executes a stage is also called a *schedule*. Since the real schedule of the threads and processes is not changed, in terms of execution order, we will call it *thread assignment* or in short just *assignment*.

Often the thread to stage assignment of a *pipe-and-filter* architecture is done before the system is executed. This is called a *static* assignment. Some systems support changes at runtime. This is especially the case, if the computational effort or the resources may change during the execution or if a static schedule could not be computed beforehand. These changes at runtime are called *dynamic* assignment [Hormati et al. 2009]. In the following approach, our main goal is to extend *TeeTime* to enable the framework to use dynamic assignments. The approach also covers static algorithms. The design is made in the style of the *autonomic computation* approach [Horn 2001; Kephart et al. 2003].

Many of the already existing approaches for adaptive *pipe-and-filter* architectures are designed for stateless stages, which can be easily duplicated. Additionally, in most cases

there seem to be no consideration of different pipe strategies for inter- and intra-thread communication. This results in the same synchronisation mechanisms in every such case. Alternative solutions, like the ones used by *TeeTime*, are often not considered [Suleman et al. 2010; Chandrasekaran et al. 2003; Soulé et al. 2013; Burtsev et al. 2014]. Stages that would use intra-thread communication are fused in these systems. Often the internal properties of fused stages are not explained further.

These approaches also don't consider feedback loops. Without these restrictions and their consequences threads can be easily assigned to a *many-to-many* relation with stages. Especially the allocation of one thread to two stages that have other active stages between them, is not possible in *TeeTime*, but is normally allowed in other approaches. The varying, already existing, assignment algorithms claim to be the best amongst each other, and this may even be true for their special circumstances. Therefore, we want our extension to be automatically able to receive and employ different assignment algorithms. For the purpose of creating our adaptive system, we need to gather general key points of existing algorithms in the next step.

For example the Feedback-Directed Pipeline Parallelism (*FDP*) approach [Suleman et al. 2010] searches for the bottleneck in the system and tries to assign more cores to it. If no free core is available, the system changes its operational mode to a power optimising one. The power optimisation mode searches for the two stages with the highest throughput that run on two different cores and assigns them to the same core. The freed core is assigned to the bottleneck . This approach tries to optimise the processing time of the elements and the resource utilisation.

In Figure 3.1 the expected behaviour of the original *FDP* approach [Suleman et al. 2010] is displayed. In the image an example system is used. All stages are allocated to cores instead of threads. During the total execution stage three, *S3*, is the bottleneck of the system. Initially three free cores are available. The other cores have exactly one stage assigned to them. At the start of the execution the adaptive algorithm begins in the *optimization mode* and identifies stage three as the bottleneck. In each iteration it assigns an additional core to the stage until no free core is available. Then the algorithm switches to the *power saving mode*, where it fuses other stages to regain resources. If no more stages can be fused without worsening the total performance, the mode is switched again. Now all three freed cores are again assigned to *S3*. At the end the algorithm can't free new resources and can therefore not assign more cores to the bottleneck. The configuration is now optimised for the given resources. To provide more stability to this process, the approach does not allow the same assignment a second time. This contrasts the needs of computational efforts that could change during the execution. If a new assignment was used and resulted in a worse performance the previous assignment would be restored. This approach strives to create a stable stage to core assignment.

In [Guggi and Rinner 2013] another approach is described. Here it is assumed that every stage is already executed with their maximal throughput. More resources can not be given to them or would not improve their behaviour. An example for this case may be a

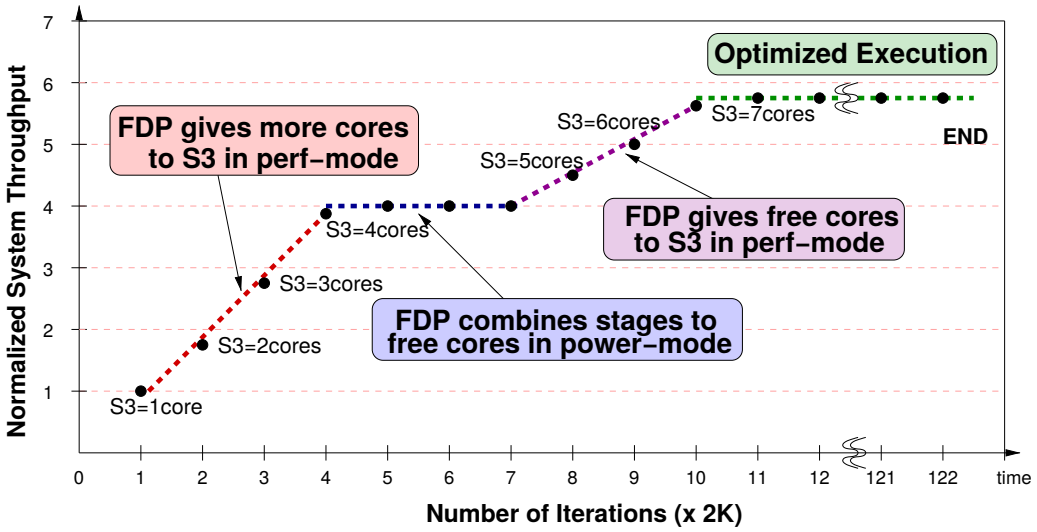


Figure 3.1. The expected behaviour of the FDP approach [Suleman et al. 2010]

configuration that is executed in a distributed system with every stage on its own node. In this case [Guggi and Rinner 2013] choose to save resources and to slow down every other stage to the speed of the limiting one, with the lowest throughput. This is ultimately done by reducing or increasing the interval in which the producers create elements at runtime.

While the two previous techniques initialise the configurations with one thread per stage, [Hormati et al. 2009] propose *Flextream* which tries to optimise the assignment beforehand through metrics. In their case the amount of input and output pipes was measured. A higher number implies a higher need for computation and therefore an own thread should be assigned to the owning stage. The partitions created by this initialisation are used as a hint for the runtime adaptation. Often the use cases differ for distinct approaches, so finding the best for a general purpose framework may not be possible. Therefore, our extension will offer a modular and configurable approach to vary the used assignment strategy and even the used metric.

All outlined assignment algorithms have three different things in common: first of all they measure a certain property of all stages. Often this is the throughput or the execution time of a stage per element being processed by it. The second trait is that they try to achieve their goal through interactions with arbitrary stages. The applied changes are mostly of the same style. A stage gets threads assigned to or the threads are withdrawn from a stage. The commonly agreed view seems to be that the assignment of an additional thread may not mandatorily improve the performance, but is unlikely to worsen the execution. Lastly they all want their systems to behave in a certain way described by the algorithms.

So, all algorithms gather data on the single components of the system and need a way

to influence them. The single stages can be seen as the resources managed by all of them. Peripheral variables, like the number of assigned or free cores, are secondary resources that are implicitly managed through manipulating the single stages. Also in the monitoring phase the point, where the measurements are taken, are the stages themselves or the pipes used by them. Therefore, we have to implement touchpoints for the monitoring and execution parts, as described by the *MAPE-K* approach [Horn 2001; Kephart et al. 2003]. Three key points are the *sensors*, the *effectors*, and the used *policy* or strategy.

Sensor We decided to put our first sensor implementation into the pipes. Here we can gather multiple pieces of information, like throughput or the remaining items in the pipe. For that, an extension of the current pipe implementation is necessary. The throughput of the pipes is a crucial element in the configuration and our extensions to gather and provide information will add more overhead. Hence, after the implementation an evaluation and a comparison of the new and the current version is important. In the *push-model* some information, like the number of remaining items, is related to the target stage. So, we decide to collect data of the input pipes of each stage to represent its state information.

Effector When we talk about resource distribution in the *pipe-and-filter* architecture we ultimately talk about the number of threads directly associated to the single stages. As mentioned earlier, we distinguish between active and passive stages. This can be generalised to stages which have a thread assigned to them and the ones that are executed by foreign threads. Distributing resources can be done by activating the stage which performance we want to increase. On the other hand reducing the used resources can be done by setting a certain stage as passive. Later on, we can extend this effector to apply multiple threads to one stage. This step needs the possibility to duplicate stages. Through these effectors most of the algorithms can be translated, but it needs other methods to use strategies that employ special behaviour like [Guggi and Rinner 2013] which reduce the producer speed.

Policy The *MAPE-K* approach describes a point, where the user can define the behaviour of the system and the rules and goals that should be applied to them, the so called policy. We've already discussed this in the field of stream processing. With the *pipe-and-filter* architecture multiple solutions already exist and none of them can be labelled as the best. We've decided to give the user some predefined assignment algorithms at hand, through which he can choose how the adaptive system will behave. This basically represents the rules or the policy for our execution.

As a conclusion we can summarise that we have a certain system that provides us a set of homogeneous resources in form of the stages. We need to alter the stages accordingly to our needs and implement effectors to activate or deactivate them at runtime. At the same time we need to gather data about the behaviour of the single stages and hence the whole system. This data is gathered at the incoming pipes of each stage. The gathering and the

interfaces to provide the information also have to be implemented. These two expansions directly influence the code of *TeeTime*.

In the next step we have to implement the four stages of the *MAPE-K* approach. Thereby, the sensors and effectors are used. The control loop frequently gathers the data from the stages and analyses them. The analysed information is used to feed the planning phase. The choice of the goals and the policy of the adaptive system is done by the choice of the assignment algorithm. In the original proposal for autonomic computation systems the analysing part is also responsible for defining the goal for the further course of the program [Horn 2001]. In our approach we gave this responsibility to the assignment algorithms, since they may have different objectives. Besides, we don't want to split these two parts and create a system more complicated to use than necessary. In the last step we have to define a method that takes the results of any assignment algorithm and transfers them into changes of the system. Our vision is to create an easy orchestration of the active and passive stages. The users may choose an existing policy or even implement their own.

The following chapters provide insights of our adaptive approach. In Chapter 4 we discuss the implementation of the effectors and the problems that may occur during the execution. Than in Chapter 5 an overview about the management system is given.

Chapter 4

An Approach to Change Executing Threads of Stages at Runtime

We discussed in Chapter 3 the need for effectors that can initiate change to the system at runtime. These effectors have to change the thread to stage assignment of a *pipe-and-filter* architecture. Hence it is necessary to enable *TeeTime*, the used framework, to activate stages at runtime. Also threads should be able to be removed from stages. This chapter describes how the effectors are implemented in *TeeTime* and which problems may arise from dynamically changing the state of a stage at runtime.

Naturally, it should also be possible to deactivate a stage and to set it passive. The freed thread can be terminated or returned to a thread pool. In both cases we have to exclude concurrency problems, like race conditions. *TeeTime* uses the *push-model* to execute its configuration. Therefore our approach is designed for the *push-model*. This decision causes some restrictions, especially which stages have to be active. For example every producer stage has to be executed by its own thread. A second restriction occurs if a stage has predecessors which are processed by different threads.

For example, assume that we have two such predecessors. If both can execute the stage, there has to be a synchronisation mechanism, which may block the other thread by an undetermined time. This is especially important if there are more passive successors. If we would decide on one of the threads to be responsible for this stage, we need to ensure that the stage is executed often enough to process every element delivered by all of the incoming pipes. At the end of the execution no element should remain that could be processed by the stage. Since the behaviours of the stages are user defined, this may not even be sufficient to be fault free. In *TeeTime* it was chosen that every such stage has to be active to avoid this problems.

In the following part of this chapter, we discuss at first the developed method of activating a stage, which problems can occur, and how they were avoided. After this the reversed case, the deactivation of stages, is presented. Afterwards we explain shortly how our approach can be converted to other execution models. At last we give an idea how to assign multiple threads to a single stage through stage multiplication. This can be implemented by using the task farm approach [Wiechmann 2015].

4.1 Activating a Stage at Runtime

In *TeeTime* we can assume that every stage that is passive can also be set as active. Since we already explained some restrictions for the configuration, we can reduce the possible scenarios. Therefore, we only need to consider passive stages with one predecessor thread. Stages with zero or more than one predecessors can't be passive and thus are already active. So this sole predecessor thread is also the only one currently executing the stage we want to activate. Additionally, the framework uses different pipes for inter-thread and intra-thread communication.

Figure 4.1 displays a simple configuration, which is used as an example in the following description. We have only two stages, which are connected by a single pipe, and one thread is executing both of them. The pipe is the regular one used by *TeeTime* for intra-thread communication, the *UnsynchedPipe*. Stage A is the producer stage of this configuration and can not be passive.

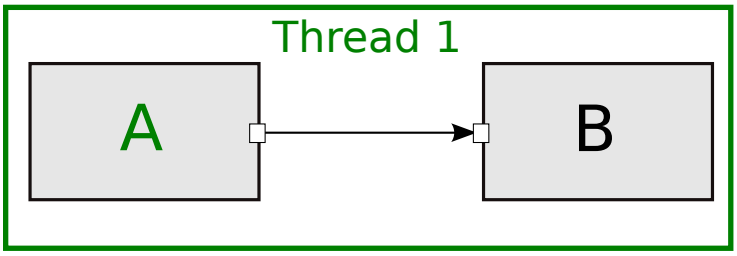


Figure 4.1. Simple configuration with a single active stage A

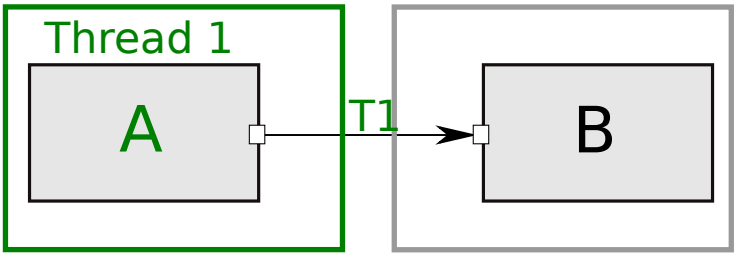
Now we take a look at **Stage B**. This stage is still passive and we want to activate it at runtime. This means the producer is running, pushing elements through the pipe and ultimately also using the following stages. To reduce complexity we first assume we have only this one single pipe between this stage and its predecessor. Later on we extend this example for multiple pipes. The behaviour and the interaction of the current stage and its successors will stay the same.

Since **Stage A** is active in the beginning and **Stage B** is passive, **Thread 1** will execute the producer stage. The pipe is an *UnsynchedPipe*, represented by a plain arrow. As soon as an element is added to the pipe, its implementation manages the workflow of the thread. The *UnsynchedPipe* forces the thread to execute its target stage, if an element is added. On the other hand one of the *SynchedPipes*, used for inter-thread communication, will just ensure that the element is added to a buffer and the execution of stage is handled by an other thread. The behaviour of the two types of pipes already gives us a hint how our starting position and our desired result differ. Therefore, just changing the pipes and starting a new thread would be sufficient at first glance.

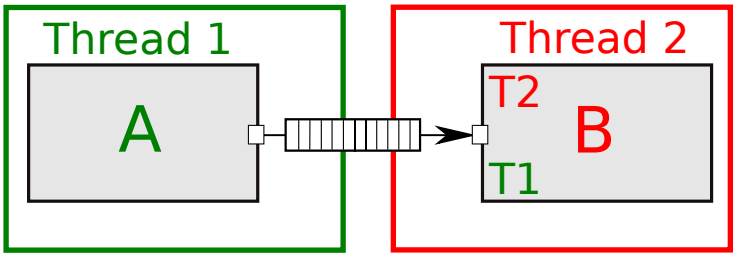
Figure 4.2a and 4.2b show a faulty situation that can occur if we just follow the naive approach. In this scenario we just assign a new thread to the stage, replace the pipe

and immediately start the thread. The described issue can occur regardless of the order of these three instructions. In Figure 4.2a the new thread is already replaced but not started. The pipe is still the same *UnsynchedPipe*. The **T1** symbol indicates which part of the configuration **Thread 1** is currently executing. In our scenario the thread is adding a new element to the pipe. The next step would be to execute the second stage. Before this happens, the stage is replaced and the new thread is started in parallel. Since **Thread 1** is still using the old pipe, it also attempts to execute the next stage, as intended by the used pipe.

Figure 4.2b shows the resulting state. Both stages have their own thread assigned and the connecting pipe is one used for the inter-thread communication. Therefore the state of the configuration is fine, but, through concurrent access and change of the pipe, we created a situation where both threads are possible executing the same code. This can result in race conditions.



(a) State of the thread just before replacing the pipe



(b) Possible state of the threads after starting the new thread

Figure 4.2. Possible faulty state during the activation at runtime

Especially if **Thread 2** obtains the only element in the pipe, a deadlock is created. **Thread 1** may not be able to progress any further and therefore can't produce new elements. In *TeeTime* a thread is not expected to execute a passive successor without having an element to process and hence an exception is thrown that results in the termination of the system. If both threads would read the same element and process it, we would also get a faulty result. Another arising issue is that elements, possibly saved in the old pipe, are lost. This

creates a similar deadlock as before. Therefore, in our solution we have to implement some sort of synchronisation that only starts the new thread if it is guaranteed that **Thread 1** is not working in the part of the partition that will be separated. Additionally while replacing the pipe, all elements and information, like if it is closed, have to be copied.

By now we have already learned that changing the pipe at runtime is necessary for the activation of a stage. Furthermore, the pipes can block a thread from accessing the following stage. We can use this property to create partitions by using the appropriate pipe. So we start by replacing the *UnsynchedPipe*. Creating and assigning a new thread also causes no problems. Now we only need to know when it is safe to start the new thread.

To solve this problem we could observe the thread, the stages of the partition or the predecessor stages. This would involve changes and gathering information of multiple parts that may again be changed in parallel. For example, another stage that is related to the one we want to activate is also activated by an other thread. So this could be less reliable, may need more synchronisation and may require numerous changes with potential overhead to the framework.

In our approach we use one part of the configuration that we have to change in any case: the pipe. Since we also don't want to change the code of the given pipes and reduce their performance during their usual execution, we implemented a new type of pipes, the *WaitingPipes*. The *ActivatingPipe* is a subclass of the *WaitingPipe* and is used during the activation of a stage. The procedure is the following:

First the old pipe is replaced by the *ActivatingPipe*. Thereby, the references in the corresponding ports are updated, beginning by the target port. The pointers to the pipes are set as *volatile* in all ports to avoid problems during the concurrent access and replacement of the pipe. Then all states of the old pipes are copied to the new pipe to save them. Since all states that might be set are never toggled again, this can be done without synchronisation considerations. In view of the mechanisms of the old pipe, no elements have to be saved now. After these preparations are done, the new pipe waits for the next element, respectively for the call to the *add()*-function. As soon as this function is used, we know for sure where the preceding thread is: In the pipe where it wants to add an element. Now this thread is used to execute the remaining parts of the activation. Since it is now safe to do so, it will first replace the *ActivatingPipe* by a *SynchedPipe*. In our case we use a bounded version. Then it adds the given element to the new pipe and copies again the state variables. After a legal state is recreated it starts the new thread.

The activated stage is used as the synchronisation object, since all involved participants have access to it. Creating a new pipe automatically overwrites old references in the given input and output port. The old pipe is not changed and hence can be used without worry to complete pending operations. To ensure a fault free usage, it is important to replace the pipe in the output port first. In the used *push model* the configuration graph remains able to navigate through. Thus, while changing or even accessing the pipe, a more complex synchronisation is not necessary. If we would not ensure that a pipe, entered through the input port, has already set an output port, it may cause race conditions and null pointer

exceptions. To avoid this, *TeeTime* was modified in a way that the target port is changed first and the input port directly afterwards.

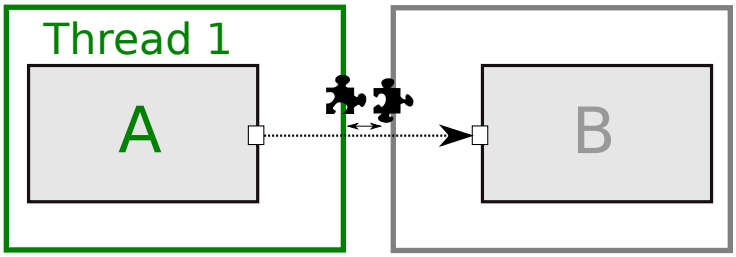


Figure 4.3. Intermediate state of the configuration during the activation

Figure 4.3 shows an intermediate state of our activation approach. By activating **Stage B** the configuration will be split into two partitions. The first part is the partition with **Thread 1**, represented by the green box. The new partition is indicated by the grey box. An associated thread is created and assigned but not started. Getting to this situation requires several steps. At the beginning of the activation **Thread 2** is created and assigned to **Stage B**. Then the *UnsynchedPipe*, connecting both stages, is replaced by an *ActivatingPipe*. In Figure 4.3, the new pipe is distinguished from the others by a different symbol. We start with an *ActivatingPipe*, but sever the connection between the stages. Therefore, the normal arrow, which represents the *UnsynchedPipe*, is altered. Additionally the dotted arrow the split puzzle pieces indicate the division of the partition.

In this configuration state the thread will eventually execute **Stage A** and afterwards the produced element has to be given to the pipe. As soon as the *add()*-function is called, we got the information about what the first thread is currently doing. We know especially that it is not executing something of the new second partition. With this information the synchronisation is finished. The intermediate pipe can now be replaced by the *SynchedPipe*. The added element is given to the new pipe and all states are copied again. This procedure restores a legal configuration of the original *TeeTime*. It is now safe to start the second thread. Both stages are now active and the configuration is the same as if they both would have been active from the beginning.

The actions taken after we got information of the thread can be handled in two different ways. The first way would be to let the thread that initiates the changes wait for the synchronisation. After this, replacing the pipe and starting the thread can be done by the initiator. The advantage of this procedure is that the changes are guaranteed to be done after the initiating thread finishes executing the method to change the stage state. The disadvantage is that we are dependent on other threads to advance in our algorithms. For example, if we wanted to activate multiple stages, we would have to wait an undetermined time for every stage, instead of using our resources in parallel.

In the second way we would let one of the threads, executing the *pipe-and-filter* archi-

tecture, do these procedures to finish the changes. Since one of our goals is to create a self-adapting extension for *TeeTime*, which may require multiple changes each time we want to adapt the system, we choose the second approach. Until the configuration in Figure 4.3 is reached, the initiator is in charge of handling all changes. Afterwards, the preceding, already existing thread handles the rest of the partitions separation.

In Figure 4.4 the resulting configuration is shown. Both stages have been activated with their own thread. Therefore the system is divided into the partitions of the two threads. The connecting pipe is the one used for inter-thread communication with a finite buffer. Hence the pictogram of the pipe indicates this used buffer and represents this type of pipes. A big difference between the *UnsynchedPipes* and the *SynchedPipes* is that the unsynchronised type forces the thread that adds an element to execute the following stage. Since this is done by a direct function calls with low overhead, a partition of multiple stages that are executed by the same thread can be interpreted similar to *fused stages*. Therefore, the activation of one stage resembles *stage fission*.

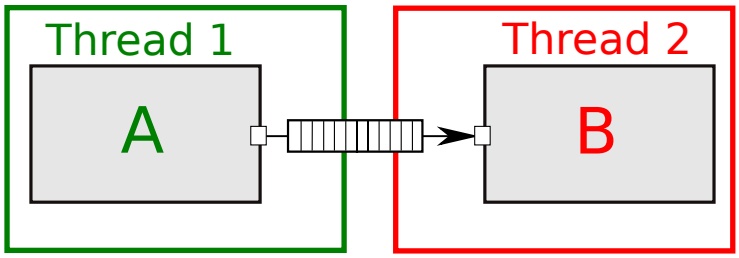


Figure 4.4. Result of the activation

TeeTime allows for a stage to have multiple input pipes. Hence a scenario, where the stage to be activated has multiple pipes, is possible. These pipes can even origin from different stages in the same partition. Figure 4.5 displays the starting point in such an example scenario. A similar situation would occur if there are only two stages with multiple pipes between them. In the given example there exist two consumer stages that are connected to the single producer **Stage A**. Lastly **Stage D** is connected to **B** and **C**, which provide the elements for the last stage. This configuration only needs one thread to be executed. Imagine this system is running and processing elements. Now we intend to declare **Stage D** as active. To enable our approach to handle multiple input ports, a simple extension is sufficient. Like before, we start with creating a new thread for the new partition. Again we need information about the already existing thread to synchronise it with the new one. All pipes have a certain probability to be used, but we don't know which is used next. Therefore all pipes have to be replaced like in the single pipe scenario. Every pipe is substituted with one of our *ActivatingPipes*.

The remaining process is shown in Figure 4.6. Figure 4.6a presents the system stage after the activation and all corresponding changes are initiated. All pipes are replaced by

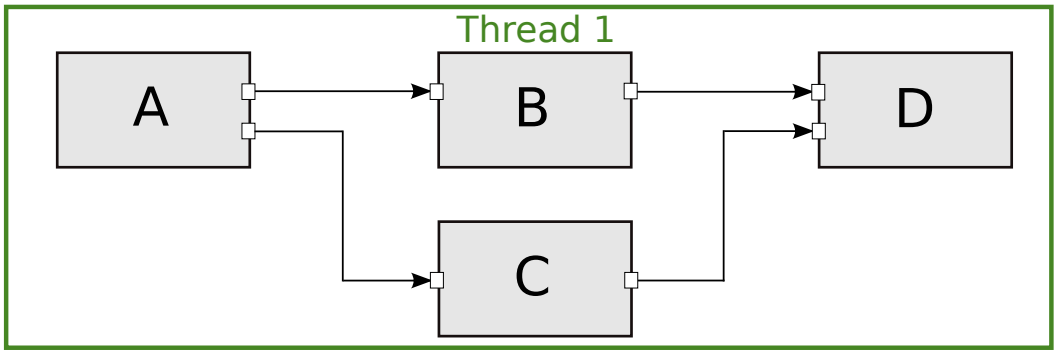


Figure 4.5. Configuration with multiple input pipes but the same thread

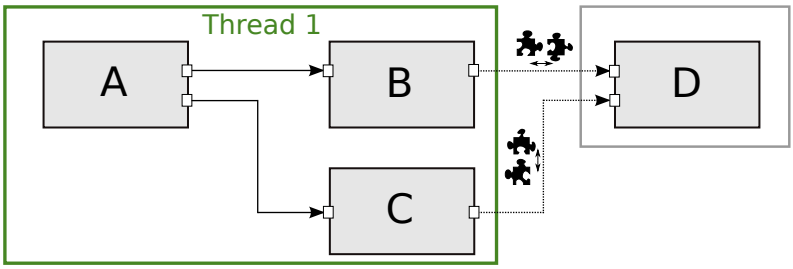
ActivatingPipe and a new thread is created. Since it is not determined which pipe is used next, we had to replace all incoming pipes. The task of the thread initiating the change is now done and it is not further involved. All remaining steps are executed by **Thread 1**.

Like before the next step requires the acquisition of information about the running thread. Again we use the fact that it has to provide its elements to its successor. Therefore, we can use the *add()*-function as synchronisation point here as well. Figure 4.6b visualises this step. The pipe between **Stage B** and **Stage C**, marked in blue, is eventually used in our example. Since we only have one preceding thread, the other *ActivatingPipes* will not be used any further and can be replaced by now. To enable our implementation on doing this, all neighbouring incoming *ActivatingPipes* of a stage know each other.

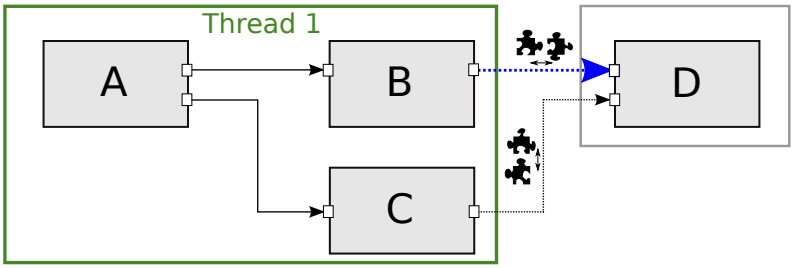
In this way we can reduce this scenario to one in which we only have one remaining *ActivatingPipe*. This case is shown in Figure 4.6c. The setting is very similar to the activation with one single pipe after the *UnsynchedPipe* was initially replaced. Now all steps are the same as before. The pipe is replaced, the new element is given to it and all states are copied. At the end **Thread 2** is started.

The result is presented in Figure 4.6d. All pipes have been replaced and the fission is complete. This example reveals a possible disadvantage in our approach. If we chose to activate a rarely used branch of the *pipe-and-filter* architecture, the *ActivatingPipes* may only be replaced after a long time or even at no time. Since the normal state is restored as soon as a pipe is visited, this issue does not disrupt the execution of the system. It may have to be considered and handled in cases where such a state or original pipes are required. For example, if we want to deactivate this activated state later on, we require that stages are not in such an intermediate state.

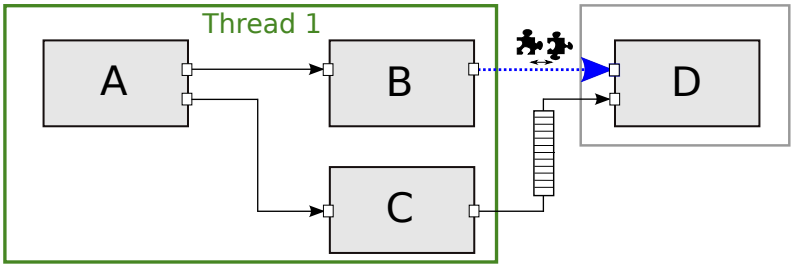
During the implementation of our approach we discovered two issues that can occur during the activation. The first one is a race condition, caused by execution details from *TeeTime*. If the initiating thread changes the *unsynchedPipe* to a *ActivatingPipe*, while **Thread 1** is currently adding an element to the pipe, the system temporarily looses track of this item. An example where this may happen is if the thread calls the *add()*-function in a situation



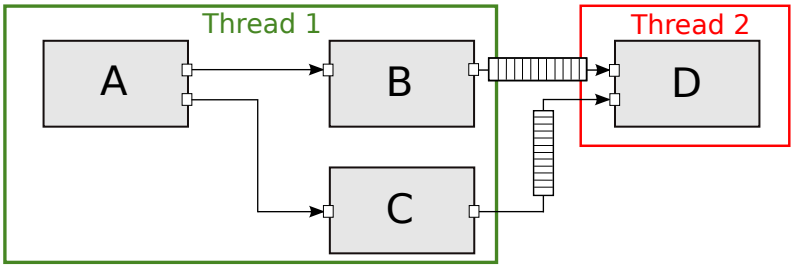
(a) At first all pipes are replaced



(b) Waiting for the next element



(c) Replace one after another



(d) Result of the activation

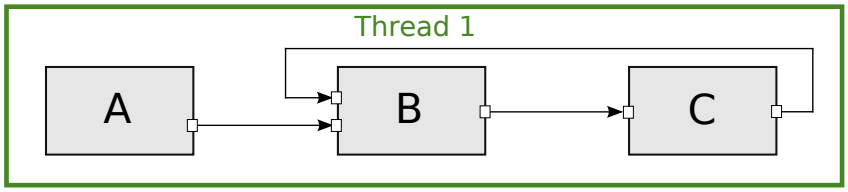
Figure 4.6. Process of the activation with multiple pipes

like in Figure 4.1 and immediately after the element is added the pipe gets replaced. Now the state displayed in Figure 4.3 is reached, but the element is not yet transferred to the new pipe. At this point the thread resumes the execution and tries to get the item it put in the pipe beforehand. As already mentioned, the semantic of *TeeTime* ensures that the element would normally be available. Due to our interaction this assumption is not met this time and an *NotEnoughInputException* will be thrown that causes the execution to abort. To avoid this, we added an additional synchronisation point that catches the exception and forces the thread to wait for the pipe to have copied all values.

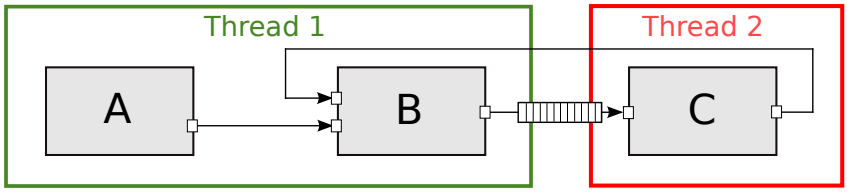
The second issue that arises comes with the potential existence of feedback loops in the system. Figure 4.7 visualises the starting situation and the problem that is created with our previous approach. It represents our currently chosen solution. We start in Figure 4.7a with a single partition that includes a feedback loop between two of the stages. It is essential that the feedback stage is not the first in this partition. In our example the loop itself is also created by a stage in this partition. The general issue can be extended to cases with several already existing partitions. Since *TeeTime* is a general purpose framework, this configuration is possible and legal. In the given scenario we now want to activate **Stage C**. This is done like before and results in the system shown in Figure 4.7b. Usually the whole activation would be finished by now and we assume that our configuration can run undisrupted again. In fact the feedback loop creates a situation where **Stage B** has two preceding threads. The first is the old one, executing **Stage A** and so far also responsible for all following passive stages. The second one is the one created while activating **Stage C**. Again our procedure would leave following passive stages in charge of the new thread. But since we have to deal with a feedback loop, both execution assumptions are in conflict with each other and a stage with two different predecessor threads is made.

Figure 4.7c displays the chosen solution in our approach. Like *TeeTime* would handle this situation during the initialisation, the stage in question is also set as active. So to repair our configuration, we have to visit every succeeding, passive stage and check if one of them has multiple preceding threads. In our example it may also be possible to reduce these three partitions into two again. But in bigger configurations this might lead to more complex changes than intended. The advantages of our solution are twofold. First the changes are clear and may be better comprehensible. Second the actions are compliant to the mode of operation of the used framework. The disadvantage is that we may produce additional overhead and use more threads than needed. Please notice that through these additional activations the number of threads required and used can be more than one per activation. This may not reflect the first impression of the intended behaviour.

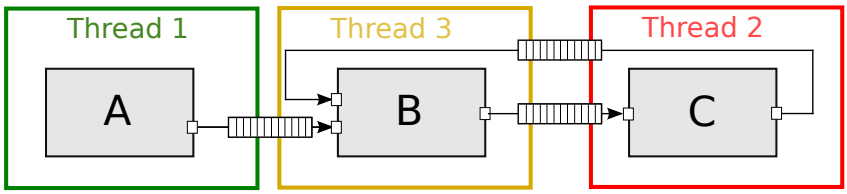
The last type of problematic configurations is similar to the issue caused by the feedback loops. We consider again a minimal example. The starting configuration, the occurring problems and our solution are illustrated in Figure 4.8. We start with the system displayed in Figure 4.8a. Again we have a single producer stage, which is directly connected to the two other stages. The whole system is executed by a single thread. An example for this configuration would be the representation of an if-statement with an empty else branch.



(a) Activation of a stage with a feedback loop



(b) Result and issue of our first approach



(c) Result of our solution

Figure 4.7. Process of the activation with a feedback loop

Now we want to activate **Stage B**. This is done as described before and the resulting configuration is shown in Figure 4.8b. As indicated in this figure, it is not defined how **Stage C** should be executed. Similar to the feedback loop issue we create a situation in which we need an additional thread to re-establish a legal state. This example can be continued endlessly by appending stages that have input pipes coming from last stage and its direct predecessor. An interesting fact is that every stage would have to be activated if we try to activate the first passive one, but if we try to activate the last one, no additional thread has to be used. This behaviour is most likely not intended by the user and needs some restrictions. Our general approach to recognise these situations is to check the successors if one has multiple direct predecessor stages. This implies the need for more than one threads, but it is not a guarantee.

We have now two options to handle these cases. Both are the extreme opposite of each other. First we can just forbid this behaviour and hence enforce that the last stage has to be activated first. The second option would be to allow these cascading activations and risk the overuse of resources. Since we want to avoid both extrema but don't want to forbid

each behaviour, we implemented a mechanism to adjust the used option.

We added a variable to determine the maximum depth in a cascading activation. If this depth is exceeded we decline the activation of the stage. Therefore setting a stage as active with a depths of zero would behave like the first option and only enable activation without cascading. A very high value, like the maximum of integers, corresponds to the second approach. Intermediate values can be used to limit the cascading and the additional used threads. The default depths is set to allow all cascading effects. This is chosen because a large cascade is unlikely to happen in most *pipe-and-filter* architectures. The added depth also has an effect to the feedback loop issue. Since we restrict the actual number of used extra threads, we also limit the activation of stages by these loops. The value chosen by the user or the algorithms that use this method may be dependent on the available resources.

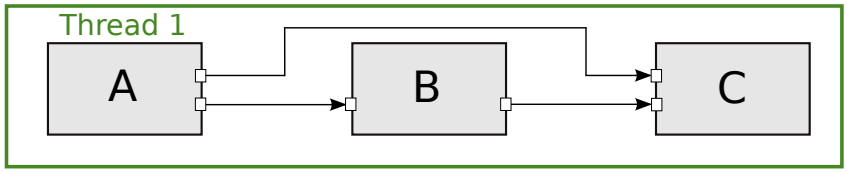
In Figure 4.8c the default result of our approach is shown. We have now three partitions and can only reduce them if we set **Stage B** as passive. Please notice that a cascade is not automatically reversed if the stage in question is set as passive again. Therefore activating **Stage B** will use two threads but setting it as passive only frees one. This issue adds to the cause that the user may not be sure how many threads he just created, while activating a single stage.

Unfortunately, our presented and implemented solution for cascading activation operations creates more issues. Imagine a configuration where the data stream is split and later reunited. An example is given in Figure 4.9a. We have one producer stage and one data sink. In **Stage B** the data is either distributed to **Stage C**, **Stage D** or both. Later **Stage E** collects the processed elements from these two stages. At first all stages are run by the same thread. Again this is a legal situation and it can be executed.

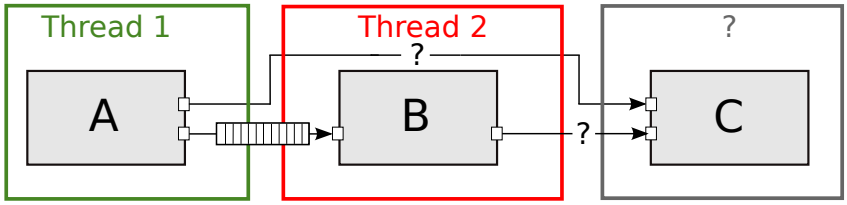
Now we want to set **Stage B** as active. Our presented mechanism recursively collects all needed threads until it meets the next active stage or the end of the graph representing the *pipe-and-filter* architecture. Remember a stage should also be activated, if the stage in question has two different predecessor stages. In this example the algorithm would eventually visit **Stage E** and classifies it as a stage that has to be active, because with **Stage C** and **Stage D** it has two different predecessors. The result is a thread assignment like Figure 4.9a. We need two more threads despite the fact that the partitions of **Thread 2** and **Thread 3** could theoretically be fused. In our approach we did not find a solution for this issue. Since this assignment still represents a legal state it can be used. On the other hand if we would deactivate our cascading mechanism, changes can result in illegal configurations. In case we would not allow cascading behaviour **Stage B** can't be activated.

4.2 Deactivating a Stage at Runtime

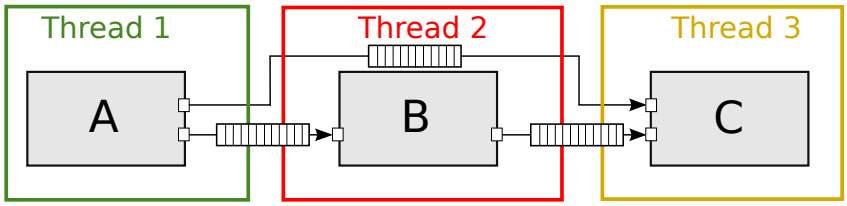
Deactivating a stage, or setting it as passive, is also necessary for an adaptive thread management in a *pipe-and-filter* architecture. As seen before, there exist special cases where a stage has to be active. These cases can not be set as passive and remain active, even if one tries to deactivate them. To provide some transparency we implemented a method



(a) Activation of a stage with a potential cascade



(b) Result and issue of our approach



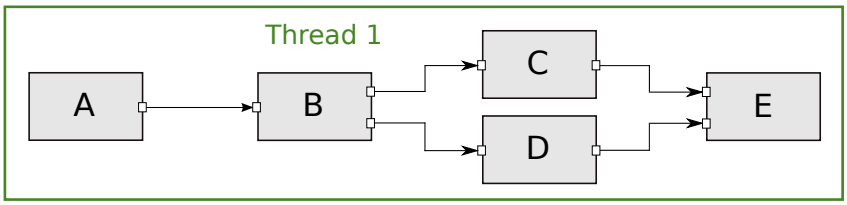
(c) Result of our solution

Figure 4.8. Process of the activation with cascading effects

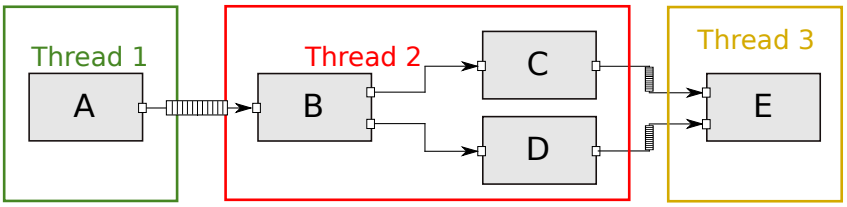
canBePassive() to the stages. It returns whether our algorithm can change this stage or not.

Figure 4.4 also shows the starting state of a simple example configuration. Here, we begin with a situation that is the same as the one after the activation. Our configuration is divided into two partitions. The first one is executed by **Thread 1** and the second one by **Thread 2**. Now we want to join both partitions and let **Thread 1** execute all stages. Since *TeeTime* employs a *push-model*, we can only deactivate **Stage B**. It is possible to take the second thread and use it as the only thread in the resulting configuration. But since we already have one handling the first stage, it would cause more overhead to swap these threads and would be impractical.

In our approach to set stages as passive we employ again the strategy of pipe changing. We start with replacing the pipe connecting both stages. As indicated by the pictogram, the pipe between these two stages is one used for inter-thread communication and therefore has a buffer. Since our desired *UnsynchedPipe* has no buffer but the current pipe has one, there may be remaining elements in the pipe that need to be taken care of. Additionally we have to avoid a situation where **Thread 1** is already finished and we deactivate **Stage**



(a) Configuration including a distributer and a merger



(b) Result of the activation of stage B

Figure 4.9. Issues arising from our cascading solution

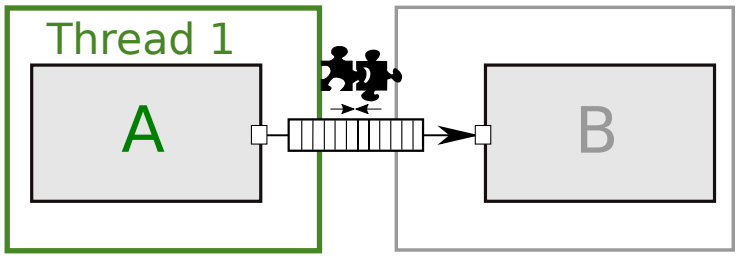


Figure 4.10. The partitions will be fused, remaining items need processing

B, thus leaving an arbitrary number of elements in the pipe. These items would never be processed. This leads to faulty results. Therefore, we have to wait again for the first thread to enter the *WaitingPipe*. Now we know that it is still alive and can handle the rest. The next step is to signal the thread of **Stage B** to die. To avoid race conditions we have to wait for its termination before we can go on. After the stage is threadless, we move on.

Figure 4.10 presents the system after we replaced the buffered *SynchedPype* by a *DeActivatingPipe*. The new pipe is initialised with the old buffered one to gain access to all remaining elements, without copying them. Also the state variables are copied. After the replacement the initiating thread is done and the remaining changes are done by the predecessor thread. The grey box also indicates that we already terminated the second thread. Now the remaining elements have to be processed.

There are three options, which thread could be responsible for processing the remaining

items. The first is the thread that calls the function and initiates the changes, to set the stage as passive. This would lead to a blocking behaviour and other changes to other stages may have to wait some time. Also the resources are used inefficiently, since two other threads are available, but aren't doing anything. The second approach would be to let the currently active thread finish the remaining elements. In this case we can't guarantee that the second thread will have finished soon after the function call. So the system may contain more remaining threads than expected.

The third approach is to let **Thread 1** do the remaining work. In this way we don't have to wait, but we enforce handling the elements by a single thread, which could have been executed in parallel. Our implementation applies the third approach. The thread processes as much elements as possible until the pipe is empty or the amount can't be reduced any further. The buffer is emptied element by element until the state in Figure 4.11 is reached. Since the implementation of the stages may vary and is user defined, it may be possible that no element is taken from the pipe in one execution. Especially if a stage has multiple pipes this may happen, for example if there exists a clock that triggers the execution.

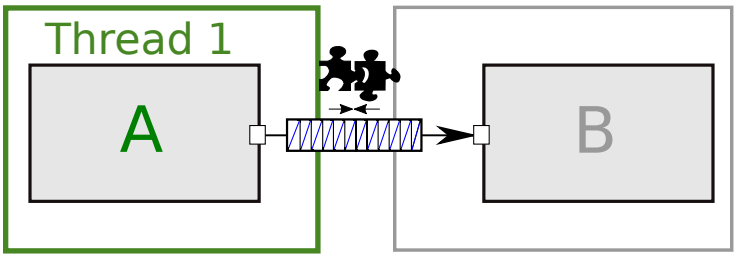


Figure 4.11. In the progress of fusion, the buffer is now empty

Thereby, an issue could arise again if there are still remaining elements at the end of the execution. For example if we can't reduce the number of elements any further and there are still some remaining in the pipe. Assume this was the last iteration of the used thread and it finishes regularly. Now we have remaining items but no thread to process them. In contrary to the first impression this is the normal and desired behaviour. If the stage would have remained active, the number of elements couldn't be reduced either. Additionally since there will be no more new items produced by the predecessor thread this situation can't change anymore. The only difference in this case is that the *DeActivatingPipe* remains in the configuration. To avoid a similar issue, created by setting a stage as passive, which has a terminated predecessor, it is important to terminate **Thread 2** only after we got hold of **Thread 1**. When the pipe is empty, we can switch it again and replace it with an *UnsynchedPipe*. In Figure 4.12 a regular *TeeTime* state is restored with one active producer. The execution can continue or we can activate **Stage B** again

Now we want to consider multiple input pipes. Again all pipes are replaced like the single pipe before, creating a situation similar to 4.6c.

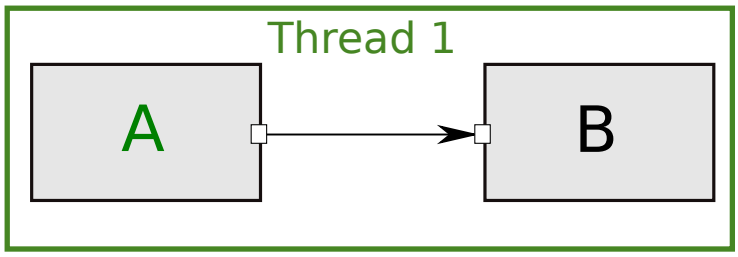


Figure 4.12. Fusion of the two partitions was completed

The used trigger is also the *add()*-function of one of the pipes. Therefore, every pipe knows their neighbouring pipes and replaces them one after another as soon as they get empty. Progress is measured if the sum of remaining elements in all *DeActivatingPipes* is reduced. As long as progress is made, the executing thread will try to reduce more of the elements. If no more progress can be made in this iteration or all pipes are empty, this step is finished. Again it may take several tries to empty all pipes and regain a system without *DeActivatingPipes*.

The feedback loop issue is also present in this approach. While setting stages at passive, we may not be able to recognise stages that don't have to be active. Figure 4.13 shows such a situation. In the configuration exist three different stages. **Stage A** is the sole producer stage in this case. **Stage B** is also active and has two input pipes. Two of the stages form a feedback loop coming from **Stage C**. The other one is a *SynchedPipe* connecting the threads. So the whole configuration is divided into two partitions. In Figure 4.7a we have already seen that it is possible to have a single thread executing the whole system. If we want to set **Stage B** as passive and fuse the partitions to create this configuration, our algorithm may decline it. This is caused by the simple fact that the stage has two different preceding threads and hence has to be active. We can avoid this false assumption by an additional test. We simply exclude the owning thread of the partition we want to eliminate during the comparison. Ultimately our approach iterates through all input pipes of the stage in question and if we find two or more different threads, we know for sure that the stage can currently not be set as passive. The result equals the configuration shown in Figure 4.7a.

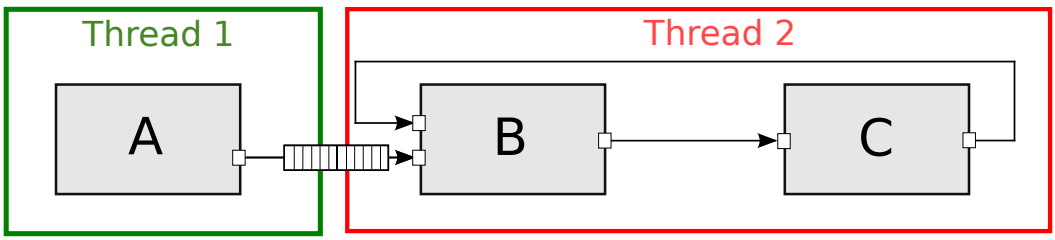


Figure 4.13. A feedback loop hindering the stage to be passive

We can summarise the course of action to set an active stage as passive as follows: Check if the stage can be passive. If this is true, we first replace all incoming pipes with *DeActivatingPipes*, which contain the replaced pipes and hence the remaining items in the buffer. Then we terminate the owning thread of this stage and wait for its termination, but only if the other thread is still alive and well doing. As soon as an element is added, we try to reduce the remaining elements as far as possible. If the pipes are empty, we replace the temporary *WaitingPipes* with *UnsynchedPipes* and regain a normal and operational configuration.

Please notice that the chosen buffer size in the pipes influences the duration for switching pipes directly. Small buffers and hence a little item count enable the algorithm to process the remaining items quickly and to re-establish a configuration with the original behaviour. Since later on algorithms may tend to deactivate fast stages, the item count will often be low due to the given circumstances. A general solution should be to take the minimum buffer size needed for an optimal execution.

On the other hand the size of the pipe influences the system as a whole. This is still true for the execution without the adaptive approach. A low value will indeed result in fast changes from the active to the passive state, but it will reduce the throughput of the pipe due to synchronisation mechanisms. A value too high will slow down the deactivation process, since we need to empty the pipe before it can replaced. In an adaptive system with the goal to increase the performance of the execution, it is more unlikely to set a stage as passive if it has many items left in the pipe. Furthermore this case is much rarer than the transfer of an element. The parameter study of *TeeTime* [Wulf et al. 2016] suggests a value of 1024 allowed elements to sustain a good throughput. Therefore in all settings the size of the *BoundedSynchedPipe* is set to 1024. Since the pipe size plays an important role in the performance of the system, an extensive parameter study may further improve our approach in the future.

4.3 Conversion to Other Execution Models

It is easily possible to adapt the presented approach to other execution models. In a *pull-model* the control flow and the restrictions would be reversed. All data sinks have to be active and stages with more than one successor thread can't be passive. In this situation we just need to change the outgoing pipes, instead of the incoming ones, of a stage accordingly. Since all active stages execute their passive predecessors, the fission and fusion are always done between the changed stage and its successor. The solutions for feedback loops and cascading activations can also be reversed.

In a mixed execution model both modifications can be applied and synchronised accordingly. If it is possible to limit the model at runtime to one of them, only the corresponding choice has to be taken. In a mixed model it should be avoided to change neighbouring stages at the same time. In case the model also utilises the mixed behaviour at runtime, more complex synchronisation mechanisms have to be chosen to avoid race conditions.

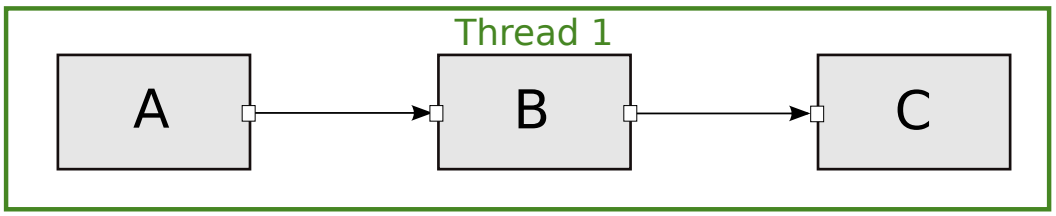
Additionally replacing input and output pipes altogether may be necessary and more restrictions may have to be applied to the configuration.

4.4 Stage Multiplication

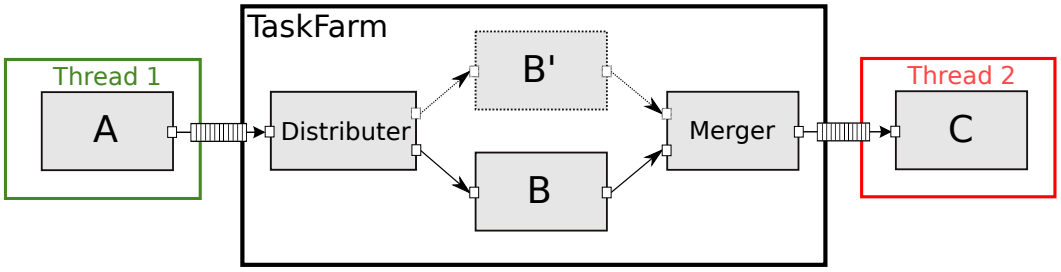
In this thesis we developed an approach for stage multiplication. As future work we recommend to implement and test it. Since our stages are not necessarily stateless, it is necessary to detect state variables. An automatic approach would consist of identifying all given class variables among other things. In general this is possible even in Java, but if we consider all variables as states unnecessary overhead may be added. Without internal knowledge of the stage implementations we can't group the variables. We can't discover which class attribute is a real state or if its values are only used as a temporary variable for some computations. To improve this knowledge the configuration may be simulated beforehand and information about the variable behaviour can be received. However, this method creates even more overhead and even with this information it is not certain if the classification is right or not. More simulation runs can increase the probability of the identification but create even more overhead and it is still not guaranteed to have an accurate solution.

In our approach we propose the use of annotations by the user. We think of all given stage attributes as states except these that are annotated to be no states. In this way we can support legacy *pipe-and-filter* architecture implementations for *TeeTime*.

With these assumptions we can avoid race conditions and reduce the overhead of our approach. The duplication of one stage needs additional stages to distribute the input elements and merge the results. Fortunately, a similar procedure is already implemented by [Wiechmann 2015]. The task farm approach takes a prepared stage, adds an active merger and an active distributer and enables the duplication of the given stage. Figure 4.14 displays an example of how this approach may be used at runtime. In 4.14a a simple configuration is given. Assume we want to duplicate **Stage B**. Now all information needed is gathered. With these the task farm can be adapted to our needs. Thereby, a wrapper class may be useful. It may even regulate the starting and the termination of the farm at runtime. The mentioned distributer and merger are created automatically. Since both are active, two additional threads are needed in this environment. The result is shown in Figure 4.14b. Here **Stage B** is duplicated once. How much duplications are created is regulated by the task farm itself. The maximum number of worker threads can be specified through the task farm configuration. The number of threads used in a certain moment of the execution can not be predetermined. **Stage B** is replaced. The task farm is seen as a new composite stage and further changes are only applied to it. Internal stages of the task farm are solely controlled by this composite stage. Through this view it would be possible to keep **Stage C** as passive, but since the work has to be done by the merger stage it can distort the performance of the farm. Capsulation of the internal behaviour of the task farm and the external configuration would create a similar view like the *DMonA Architecture*



(a) Simple configuration, stage B should be duplicated



(b) Duplication result with the task farm

Figure 4.14. Envisioned approach of stage duplication at runtime

[Michiels et al. 2002]. It divides the system into different part and manages only these big parts. The local strategy of these little systems is again managed by themselves.

Additionally to this multiplication approach at runtime and the needed adaptation of the task farm, a system for type safety is needed. Also an extension to multiply stages with more than one input or output pipe is necessary. Finally there may be some special cases where the duplication may not be possible or create new problems during the activation or deactivation of stages.

Chapter 5

Self-Adaptive Resource Distribution

In Chapter 2 we described the autonomic computing system approach and a general proposed architecture for this purpose. Later on, in Chapter 3 we discussed how we want to translate this general approach to the *pipe-and-filter* architecture. Thereby four different components were displayed, which have to be implemented: The *effectors*, the *sensors*, the *control loop* with the four different phases, and the policies. In Chapter 4 a detailed discussion about the implementation of the effectors is given. The structure of the remaining components is described in this chapter. Additionally we give an explanation to the envisioned behaviour of the adaptive extension. The predefined algorithms and metrics are also displayed later on.

In our approach we encapsulated the four phases of the control loop into two service classes. The monitoring and the analysis is done by the *AnalysisService*. The planning and execution part is controlled in the *ThreadAssignmentService*, which again uses the analysis. We want to give the user a way to predefine the rules for the behaviour of the adaptive system. A set of these rules is described as policy in the autonomic computing. We already discussed why it can be useful to employ different policies in different use cases. Since *TeeTime* is a general purpose framework, which allows the user to design its own custom *pipe-and-filter* architecture, the used stages can be taken from a pool of given stages or can be implemented by the user. Thereby, varying situations can arise, which make it difficult for a single policy to deliver optimal results. Continuing in this style we focus especially to enable our system for an easy replacement and implementation of these rule sets. We express them through the assignment algorithms, which choose in every loop iteration which stages should be active or passive.

Additionally different metrics may be relevant and the results may vary depending on the measured properties. At the same time the assignment algorithm should be independent from the chosen metric. Both parts should be able to be combined without restrictions. We also implement some algorithms and metrics as examples to show the feasibility of our approach.

5.1 Structure of the Self-Adapting Assignment Extension

Figure 5.1 presents a simplified overview of how *TeeTime* was extended. The changes from Chapter 4 are not displayed here. A new service class is introduced to the framework. The

ThreadAssignmentService is added to the *ConfigurationContext* in the style of the existing *ThreadService* class. The latter is used to initialise the configurations and its threads. There exist some minor changes in the initialisation of the execution of the configuration, its start, and its termination. The added service coordinates the implemented functions of the chosen *ThreadAssignment* and the *AnalysisService* with its corresponding *metric*. The *ThreadAssignment* is represented by the abstract super class for all future implementations, the *AbstractThreadAssignment*. The specific classes provide algorithms to initialise the thread to stage assignment before the executions starts. This part is sufficient to implement static assignments.

Furthermore and most important to our goal of a self-adapting thread assignment, in a second and dynamic part of these assignments algorithms are provided that have to decide if the system should be changed or not. The purpose of the assignments is to provide the planning and executions parts of the *MAPE-K* approach, described in Chapter 2. The monitoring and analysing parts are contributed by the *AnalysisService* and the used *metric*. In Figure 5.1 the *metrics* are again represented by an abstract class, the *AbstractMetric*. The *metrics* apply the monitoring of specified properties. Since there exist multiple properties in a *pipe-and-filter* architecture that may be used to measure the systems state, the *metric* should be replaceable. Examples for measurable properties are the throughput of a stage, the execution time per element or the remaining elements in a pipe at a certain moment. There exist many more attributes that can be measured and hence the user can even implement her own *metric*. The service class uses this data to analyse the system state, which can be used by an assignment algorithm to plan the further course of action.

The extension is initialised with a default assignment and a metric. Currently the implemented default assignment is static and represents the behaviour of *TeeTime* without the extension. As the default metric the *PullThroughputMetric* is used. The *Configuration* is complemented by the two methods *setThreadAssignment()* and *setMetric()*. These allow the user, who builds the configuration, to choose another algorithm and metric. The assignment and the metrics are designed to be independent from each other. Therefore the user isn't bound to use the "right" combination of both parts. Instead he can combine every available implementations. Please note that the setter methods are not designed for usage during the execution. This would cause race conditions and may also eliminate the comparability of the metric. In the default setting the added computation is limited to the additional initialisation of the *ThreadAssignmentService* and monitoring operations. To enable system monitoring we had to adjust the pipes to measure properties like throughput during the execution. This adds some computations for every passing element. There are no further big changes to the original code of *TeeTime*.

5.2 The Design of the Thread Assignment

Figure 5.2 displays the structure of our assignment part. The thread assignment is designed for easy implementation and replaceability. The user should be able to use almost all

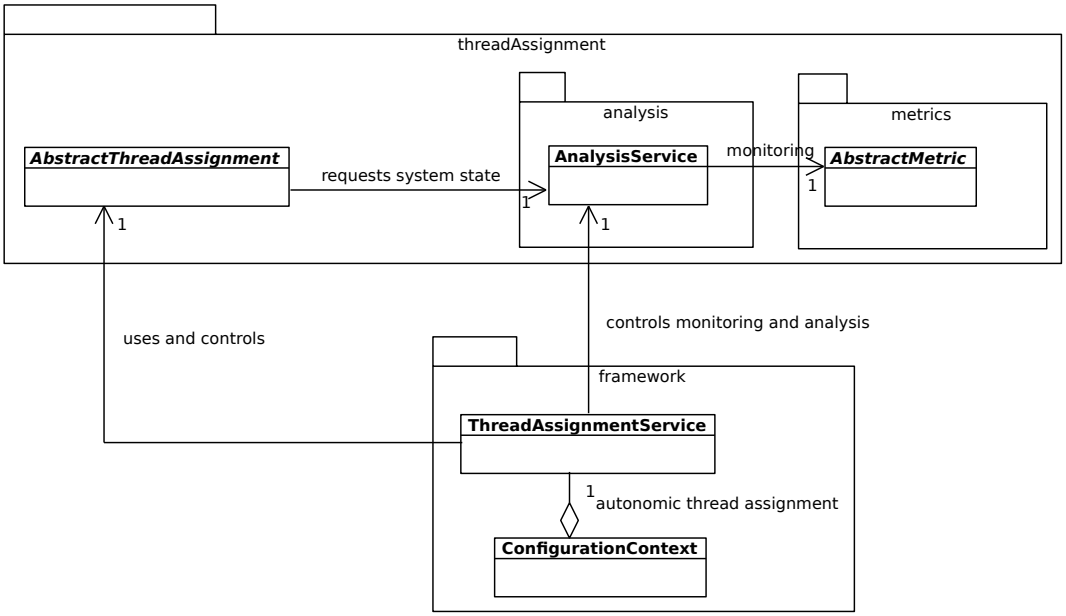


Figure 5.1. Simplified overview of the extension to *TeeTime*

techniques that could be used to decide on the assignment that should be employed next. To do so the dynamic thread assignment part is built around the abstract class *AbstractThreadAssignment*. Implementing a specific assignment requires the implementation of two abstract methods:

setFirstAssignment() The initial thread to stage assignment is done in this method. If the given *pipe-and-filter* architecture should not be changed, this function can remain empty. Changes can be done directly by the stages, for example through *declareActive()*. *TeeTime* resumes its own initialisation afterwards and adds necessary threads.

changeAssignmentAtRuntime() A dynamic thread assignment needs to implement the *changeAssignmentAtRuntime()* method. As the result a map with the stages and an associated integer value is expected. The integer implies if a stage should be passive (0), active (1) or even should be duplicated (> 1). If a stage is not contained by the map, the algorithm tries to set is as passive.

A user does not need to implement more functionality to get her own usable assignment. The abstract class has different mechanisms and implemented methods to support this changeability. Additionally, commonly used objects like the *AnalysisService* are given to the user to simplify the implementation.

dynamic This variable implies if the assignment is set as *static* or *dynamic*. The default case is a *dynamic* assignment. It can only be changed through the constructor.

stages This set includes all stages that are available in the execution. The set is automatically computed. Stages that aren't reached through the execution graph are not listed.

currentThreadedStages Similar to the last variable, this set contains all currently active stages. The set is updated automatically.

analysis In most cases an assignment is decided through monitored and analysed data. These data are provided by the *AnalysisService*, which is accessible via this variable.

allowedActivationDepth As described in Chapter 4 the activation of a stage may require to activate one or more of its following stages. Since this can cascade in big chunks of the configuration to be activated, we chose to implement a limit to this behaviour. If an activation would require more new threads than this variable allows, the activation is declined. A value of 0 would forbid the activation of other stages. A high value, like the largest integer value, will allow an arbitrary number of additional stages to be activated.

changeAssignment() This method receives the output of the user-defined function *changeAssignmentAtRuntime()*. Changes are only applied if they are possible. Hindrances could be ongoing changes or stages impossible to be set as passive. It updates the *currentThreadedStages* set accordingly to the fulfilled changes. It is used by the control loop.

onInitialize() This method is used immediately before the initialisation of the *ThreadService*. Here the changes from the *setFirstAssignment()* are put in action. It is also automatically used by the extension.

startAssignment() After the execution of the configuration is started this function is called to start dynamic assignments and therefore the *AssignmentAdaptationThread*.

requiredThreadsToActivate() This method is another convenience function of the abstract assignment. As input it requires a single stage. The result is the count of threads that are needed to activate the given stage. A value of one implies that only the stage itself has to be set active. The result for an already activated stage is zero, since no additional thread is needed.

finish() At the end of an execution the adaptation thread is stopped and the currently active stages are returned to allow their termination.

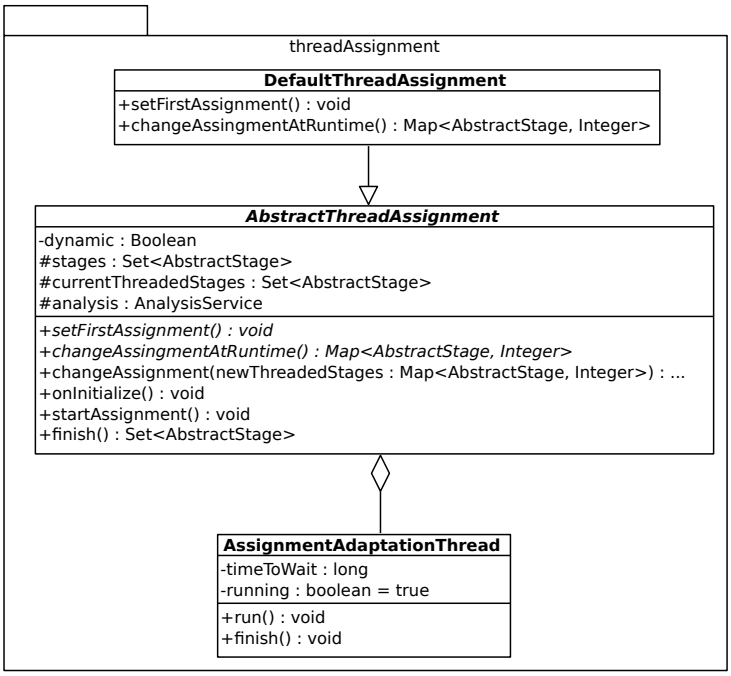


Figure 5.2. Simplified overview of the assignment part

All function calls and changes at runtime are done by the *AssignmentAdaptationThread*. This thread is only created and started if the assignment is a *dynamic* one. In the *static* case the missing thread will not call the methods of the assignment class and therefore will not use resources or create overhead. The only exceptions are the added measurements for the metric, for example the throughput in the pipes. The methods and attributes are kept simple.

timeToWait The *timeToWait* attribute is most important to the *AssignmentAdaptationThread*. This variable determines how long the thread waits between each iteration in milliseconds. If this value is low, the thread is invoked more often and completes more iterations of its main loop and so increases the frequency how often we attempt to change the assignment. This implies more resource usage of the system. A low value like zero may burden the system more than a high value. Especially if the resources are limited, this has to be considered. Of course the complexity of the given assignment algorithm and the metric can influence this further. The monitored data is only collected by the *AssignmentAdaptationThread*. The value of the *timeToWait* attribute plus the time needed for executing the main loop, give an implicit interval for the collected data. In some assignment algorithms it may even be

better to have bigger intervals and hence probably more stable data about the behaviour. In other implementations one may want to react fast to adjust the system immediately. One example would be the simulation of trigger events.

running This boolean is used to start and stop the main loop of the thread.

run() The implemented *run()* method of the threads consists mainly of the *while* loop. Here the monitoring, the analysis, the assignments and the applying of the changes are brought together to create the self-adapting system. The thread initially waits for the time given by *timeToWait*. It then invokes the updates of the measured properties. At first this enables the analysis to process the first data and later on serves to let the analysis work with the most current data. The thread next calls the implemented *changeAssignmentAtRuntime()* of the specific assignment. With the result the *changeAssignment()* method is invoked. After all changes are invoked the loop starts again with the thread sleeping the given time.

finish() At the end of the execution the thread is terminated. This happens to avoid unnecessary computation and to hinder stages to be set as active whilst the configuration is terminated.

5.3 The Design of the Analysis

Figure 5.3 provides an overview of the part of the extension used to monitor and analyse the system. The structure is similar to the one provided for the task farm approach [Wiechmann 2015]. The task farm is also implemented in *TeeTime*, but unfortunately the used elements like the *History* class or the algorithms used for analysing need to be rewritten for the requirements of this approach. Since the author builds a similar self-adaptive system in the *pipe-and-filter* context, we can use techniques implemented in her work and adjust them to our needs. At the same time this means that there exist classes from the task farm and from our approach with code that may be nearly alike. The unification of these classes could be done in the future to meet good software engineering standards.

All access to the monitoring is done through the *AnalysisService*. The service connects the *History* class, the chosen *metric* and the used *AnalysisAlgorithm*. It provides some attributes necessary to fulfil its duty:

stages Similar to the *AbstractThreadAssignment* the service needs to know which stages are there to be analysed. They are also set during the initialising of the services.

getNormalizedSystemState() This method uses the data measured by the *metric* that was saved in the *History* class. The data is saved for every stage. Additionally there may be more data than the last measured ones. All values are processed to represent the current stage

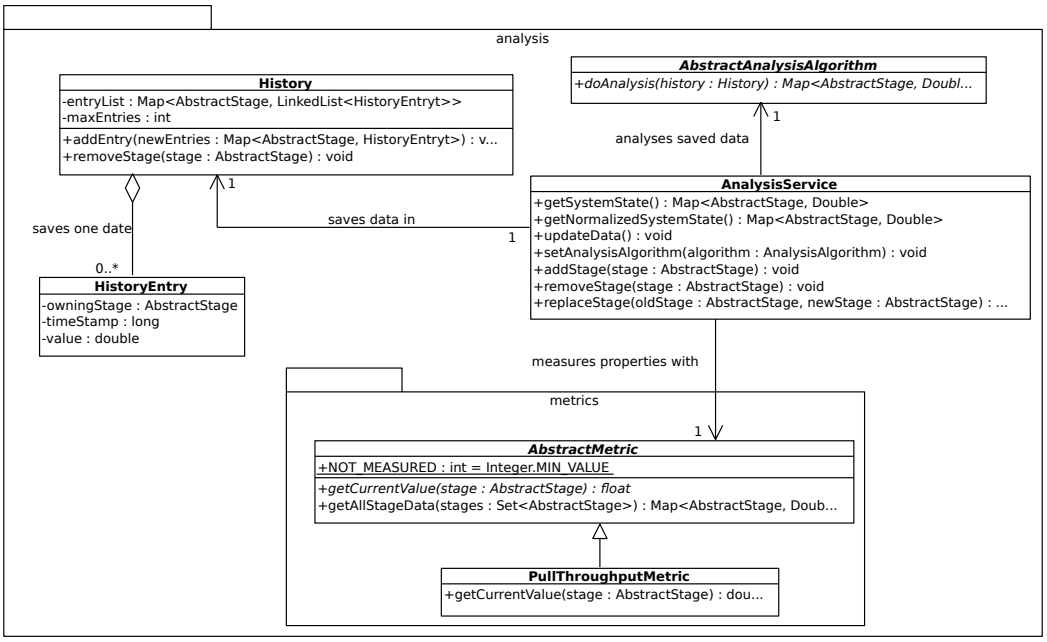


Figure 5.3. Simplified overview of the analysis part

state. For this an *AnalysisAlgorithm* is used. An assumption we make is that every measured property is zero or greater. Now we can search the maximum value and normalise all values by it. This results in all stage states being represented as a value between zero and one, which are returned. This way the normalised value represents a comparison with all other measurements.

getSystemState() Similar to *getNormalizedSystemState()* this method analyses all available data through an *AnalysisAlgorithm*. This time the values are given without normalisation.

updateData() When this method is called the *metric* is used to gather the current data for all *stages*. The measured values are enriched by the time stamp. The resulting *HistoryEntries* are saved by the *History*.

setAnalysisAlgorithm The default analysis algorithm can be replaced with this method. Currently all given algorithms are registered in an *enum* type. The implemented algorithms are the same as given by [Wiechmann 2015].

addStage(), removeStage(), replaceStage() Usually the *stages* should be fixed after the initialisation. Later on the assignment may be able to duplicate or replace stages. In cases like this the monitored stages may change. For example if we decide to replace a stage with the task farm [Wiechmann 2015], the stage is replaced by a composite stage with distributer, a merger and duplications of the old stage. All internal configurations are handled by itself and only the state of the composite stage in total matters. This can be put in comparison to the other stages. These 3 functions are implemented to cover these cases.

The *metric* is the second part of the extension that is designed to be replaceable and chooseable by the user. It is responsible to measure a property of every stage and to transform it into a comparable number. The *AbstractMetric* consists of two methods of which one has to be implemented in the chosen subclass.

NOT_MEASURED In some cases the metric may decide that a stage can not be measured. For example if there is not enough measured data or the stage is currently changing. Since we assume that a measured value is unlikely to be negative we chose the lowest value of the *Long* range as the value of *NOT_MEASURED*. In fact the only metric with negative values we could think of was the pull-push-difference, which can be transformed to the push-pull-difference. This value is a static class attribute so it may be used in other places to react to these invalid measures.

getCurrentValue() This method is used to measure the state of a single stage. Every metric has to implement this function in order to measure its related properties. The stage to measure is given as the only parameter. Through this stage we have access to several attributes and methods of the *AbstractStage*. This allows, for example, to receive all incoming and outgoing pipes and measure their characteristics. The currently implemented metrics require the pipes to implement the *IMonitorablePipe* interface. This restriction is not a fixed one. A user can decide to measure properties of pipes through other meanings or even get data on stages. In the future this system can be extended to use a general monitoring interface, for example using higher order functions.

getAllStageData() Gathering all stage data is done through *getAllStageData()*. The method is pre implemented but can also be overwritten to meet special needs of a metric. In the given version all stages are measured through *getCurrentValue()*. If a stage returns the *NOT_MEASURED* value, this tuple is discarded and not saved. This doesn't exclude future measurements. Since it may be that a pipe doesn't implement the monitoring interface, *ClassCastExceptions* are also caught and handled the same way as invalided values. The result is a map with all stages that got valid values. It is important to notice that the stage with the largest value represents the one with the least potential for optimisation or the "fastest" one. This is necessary for the *AnalysisService* to work the intended way. A reimplementation has to consider this. It may even used to provide this order if the

values would deliver another one, for example if the execution time was measured. Another important note is that all values are taken one after another and a potential time stamp will be different from the time stamps of other values in the same iteration.

All stages that were ever measured in this configuration are saved in the *History* with their data. The data is only saved a limited number of iterations. Every date is saved in a new *HistoryEntry*. The *History* provides the following attributes and methods:

entryList The saved values are represented as a map with the stage as the key and a list as the value. The list contains all saved measurements until a certain size is reached.

maxEntries This attribute determines the amount of the saved data per stage. For example if *maxEntries* is initialised with 5 there exist only 5 or less entries per stage at the same time. This value determines how old a measurement should be to have influence on the analysis.

addEntry() Adding a new iteration of measured values is done through *addEntry()*. The input of this method is a map, with the measured stages on the key side and the current value, encapsulated in a *HistoryEntry*, on the other side. Every list corresponding to a given stage is updated. If the size of the list reaches the value of *maxEntries* the first and hence the oldest value is deleted.

removeStage() In the case that we replace a stage in one way or another, we don't want to have remaining data about non-existent stages. These data could lead to a not intended behaviour of the assignment algorithm and is hence removed. The new stages are added as soon as they were measured and don't need to be added manually.

The *HistoryEntry* saves different information about the measured values. Currently an entry saves the value itself and enriches the measurement with the associated stage and a time stamp. While most analysis algorithms don't need the time stamp, some, like the *RegressionAlgorithm* need this information. This procedure allows for a modular way of saving information and extending the algorithms later on.

The last part of the analysis are the *analysis algorithms*. While many approaches like the *FDP* approach [Suleman et al. 2010] or *Flextream* [Hormati et al. 2009] don't describe in detail how the monitoring is evaluated, other approaches like the task farm [Wiechmann 2015] use advanced algorithms that consider older measurements to create a stable estimation of the system state. They may even try to forecast the state of the stages. Since the task farm is improved by the use of these algorithms, we adapt them and also use them in our analysis. More analysis algorithms may be developed in the future by extending the *AbstractAnalysisAlgorithm*. The currently available analysis algorithms are registered in an

enum in the *AnalysisService* and can be chosen through the service. There exists only one method that has to be implemented by a specific algorithm. The *RegressionAlgorithm* is the default algorithm, since it provides the best results in the study of [Wiechmann 2015].

doAnalysis() Given the *History* this method analyses the available data per stage. The given stage data is reduced to a single value. This can be used to create some foresight of the expected performance or to eliminate peaks in the measurement. It results in a map, where the stages are returned with their corresponding analysed values.

5.4 The Behaviour of the Thread Assignment

The self-adapting approach consists of different phases. One of them is the decision, what should be changed, based on the system state. Another part is to apply the changes to the system. These two phases are included in this section. While the user can implement her decision algorithm as the thread assignment, the changes computed by it will be automatically put into action by the presented extension to *TeeTime*

The thread assignment distinguishes two cases: It can be *static* or *dynamic*. In the static case, we only want to adjust the initial thread to stage assignment. After the execution of the system has started there is no further need to monitor properties or to try to change the active stages. This may be advantageous in configurations where the computational effort doesn't change and it can be determined beforehand. If the user wants to implement a static assignment, he can enforce this behaviour in the constructor of the *AbstractThreadAssignment*. In a static algorithm only the *setFirstAssignment()* method will be used to configure the thread to stage assignment before the system is started. If the user has decided to set stages as active while building her desired configuration, these settings are not discarded by the implementation. An algorithm can decide to ignore these user wishes, but it has to set every activated stage as passive again by itself. Since the user who builds the configuration may know the best where active stages may be useful, we decided on this implementation.

At this point, where the configuration isn't running, all changes to the stages are done through the *declareActive()* and *declarePassive()* methods. Some approaches to initialise the assignment may not be supported with this implementation. For example [Chandrasekaran et al. 2003] simulate the execution beforehand to evaluate the performance of the stages. This is currently not possible in our *TeeTime* extension.

After the initial assignment is set, *TeeTime* activates all stages which have two different predecessor threads and are passive. Then the configuration is validated and started. A static algorithm ends here and the system is executed uninfluenced. A dynamic one may also use the *setFirstAssignment()* method and behaves the same way until the configuration is started.

Changing the assignment at runtime can be done through different techniques. One procedure would be to halt the execution and make the necessary changes. After all changes are finished, the system is restarted. This approach would require further intrusion

to the original code. Furthermore it could cause performance issues or may need more synchronisation mechanisms, for example, if we have a system with multiple active stages and we want to set more than one as passive. Now the one thread that intends to change the assignment has to process the remaining items in the buffered and synchronised pipes. This has to be done for every stage we want to set passive and only afterwards the execution can be resumed. A variant of this procedure would be to employ threads to every stage that will change, let them do the work and synchronise them while carefully keeping a legal state. Another way could employ flags that are set while the system is running. Checking these flags would cause an overhead for every passing element. The synchronisation may also be a difficult task.

Our approach to change stages at runtime, from Chapter 4, supplies us with an intermediate technique. In our approach we only stop the regular execution of the affected stages and replace the pipes with ones that employ temporary synchronisation mechanisms and flags. The remaining configuration is not influenced. After all necessary elements are processed in our example, a regular state with low overhead is restored.

We provide an abstract class for the actual assignment algorithm. It implements some mechanisms to unify the process of changing the assignment. The user only needs to decide if the assignment should be dynamic or not and implement the *setFirstAssignment()* and *changeAssignmentAtRuntime()* functions. As mentioned before *setFirstAssignment()* is used to provide an initial assignment through setting stages as active and passive. For a dynamic behaviour the *changeAssignmentAtRuntime()* should be implemented. Through the described behaviour of this method it is only necessary to consider stages that should be active in the future. The *AbstractThreadAssignment* provides some attributes for commonly needed objects. First of all the *stages* attribute contains all connected stages of the configuration and can be used to iterate through all available stages. They are collected and given to the algorithm as soon as the *Execution* environment is created. The *currentThreadedStages* attribute, which keeps track of the active stages in the system, is similar. It is changes with the results of the *changeAssignmentAtRuntime()* if necessary. Since *TeeTime* adds threads after *setFirstAssignment()*, the collected active stages may not be the same as the ones used during the first dynamic assignment of the configuration. On the other hand they are a subset of all active stages that are created by the framework and the user, who built the configuration. The stages added by the framework can't be passive anyway. In most of the cases we also want runtime information about our system to decide which assignment should be chosen. To gather this information the *AnalysisService* can be used, which coordinates all monitoring related tasks and is described in depth in the next section.

The iteration of the adaptation circle has to be executed in some manner. One variant would be to let the configuration or one of the active stages decide if the next iteration should be started. This may be useful if the system should be examined every time a specific progress is made or a certain event is triggered [Welsh et al. 2001]. The advantage would be to be able to react directly to events in the system. More augmentations of the original code and possible need of more synchronisation would be the disadvantages.

In our approach we choose to start a new thread solely responsible for monitoring the stages, computing new assignments and changing the system. The thread checks the system every predefined time and decides on the course of action. In this approach the control flow is kept inside the extension. The adaptation can be seen as influencing the configuration from "above", since we see the system as a whole and check it from time to time. The disadvantage is that we can't react directly to events and we need to start an additional thread that consumes resources. Especially if we want to establish a limit of the maximal used threads, this has to be manually considered by the assignment algorithms. This may reduce the option of the potential algorithms. This solution represents the *MAPE-K control loop* [Kephart et al. 2003]. A possible approach to simulate the event triggered behaviour may be to add special stages to the configuration that register these events. The assignment algorithm can now especially check these added stages in a high frequency. We will not look further into the event based adaptation.

The *AssignmentAdaptationThread* is only started if the assignment is set as dynamic, which is the default setting. After the framework has validated the configuration and started it, the adaptive assignment is activated through starting the adaptation thread. This thread runs until the configuration is terminated. Its behaviour is rather simple and acts as the control loop mentioned before. First it waits as long as the time was set at the initialisation. This allows the system to initially process some elements and gather data on the stages. Afterwards the thread updates the measured data through the *AnalysisService*. For the likely case that the assignment algorithm uses these monitoring data the first iteration will already have some data available. In the third and next step we can now use this algorithm with the latest information available. Until now all computation was done in parallel and with no interruption to the stages. With the new assignment, given as the map of stages and integers, the thread influences the set-up directly with the methods described in Chapter 4. Now the loop begins anew with the thread waiting. If the (de-)activation of a stage is not finished until we want to change it again, the new modification will not be done. At the end of the computation the thread is signalled by the assignment to stop. Additionally the assignment returns which stages are currently active. This allows the framework to stop all necessary stages.

Figure 5.4 shows the behaviour of the *AdaptationThread*. This includes most of the assignment adaptation and the complete dynamic part. The missing parts are especially the initialisation and the termination of the thread. In general we can say that the thread is started alongside the execution of the *pipe-and-filter* architecture and is also terminated at the end of it. The initialisation is done during the creation of the execution environment.

After the creation of the *AdaptationThread* the system awaits the start of the execution before the start signal is sent. This guarantees that all initialisation is done undisrupted and finished before anything other is done. Directly after the thread is started we send it to sleep. The time the thread is going to sleep in every iteration is set in the *timeToWait* attribute mentioned before and can be chosen during the implementation of the assignment algorithm. Now the system has executed some time and we can measure the properties

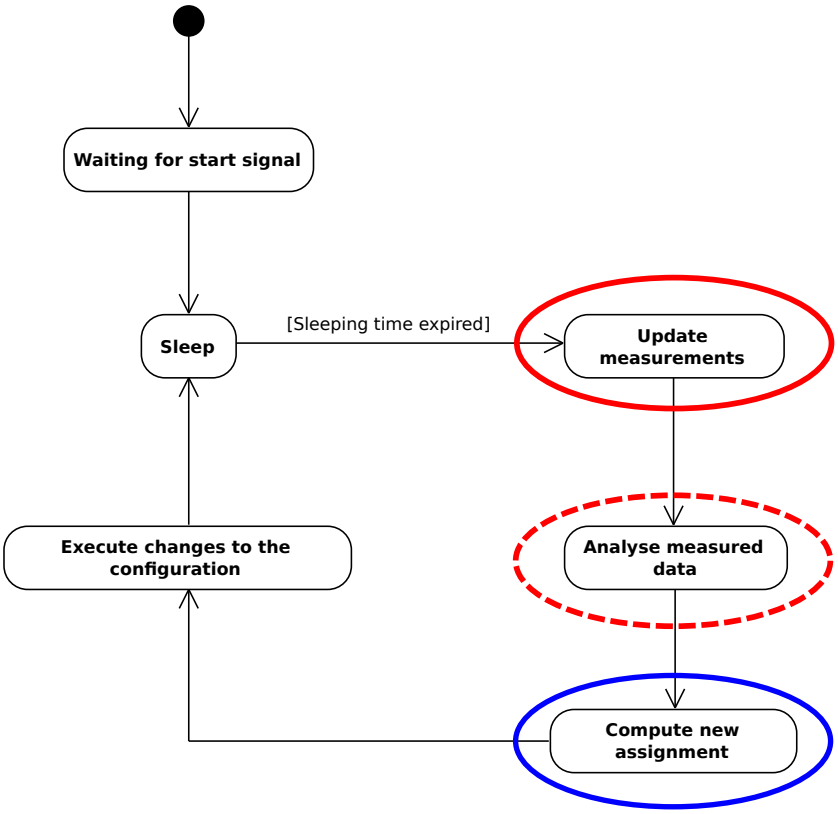


Figure 5.4. The workflow of the AssignmentAdaptationThread

through the given *metric* and update our data. The new data is saved alongside a certain chunk of older measures in the *History*. Of course in the first iteration no other data is available. The gathered values and potential older ones are now used to analyse the system state and to discover performance trends. Thereby the selected *AnalysisAlgorithm* is used. In the next step the analysed system state is given to the used *ThreadAssignment*. This assignment algorithm uses the implemented *changeAssignmentAtRuntime()* and computes a thread to stage assignment that may improve the performance of the system. The improvement is no given restriction. In fact even assignment algorithms with the goal to get the worst performance or one that toggles the active and passive stages in each iteration can be written. With the intended thread to stage assignment on hand, the *AdaptationThread* enters the last phase. All changes are applied to the system if they are possible. If parts of the new assignment are not possible, these are not executed. Other changes are not influenced. After all changes are initialised through the effectors described in Chapter 4 the loop begins anew with the thread sleeping. The next iteration will have more data available

until the predefined cap is reached.

There are three circles in the workflow diagram. They imply some sort of possible user influence. The red colour stands for the analysis part and the blue represents the assignment part. The *metric* can be influenced in two ways. It can be chosen by the user while constructing the configuration. Additionally it is also possible to implement a new metric. The *AnalysisAlgorithms* can indeed be adjusted the same way through given functions but in contrary new implementations by a user are not intended, but possible. Some settings that can be chosen are the history size or the used *AnalysisAlgorithm*. The *assignment algorithm* behaves again like the metric. A specific one can also be chosen during the construction and a new one can also be implemented. Here the *timeToWait* can be chosen and influences the behaviour.

5.5 The Behaviour of the Analysis

A system that strives to adapt itself to certain situations or events needs to be able to register this appearances. Since we displayed the behaviour of the assignment part in the last section and the structure of the analysis in Section 5.3, we now want to give an overview of the behaviour of the analysis. In a system many different observable properties may exist. To be usable by a wide range of algorithms the interfaces have to offer well defined return values. At the same time many metrics deliver data in different meanings that are not necessarily comparable. Even so we want to represent a valid system state by analysing arbitrary data and this state should be comparable and hence used by any assignment. For example we want to be able to measure the throughput of a stage and recognise which is the fastest. If we would measure the execution time per element and stage in the same configuration we don't necessary need to have the same results, but we want to state if the order from "fastest" to "slowest" stage has changed. To enable such a usage the *AnalysisService* coordinates this subsystem and is used through *updateData()* and *getNormalizedSystemState()*.

As soon as the *updateData()* method is called, the service requests the current measurements of all stages known to it from the chosen metric. Now the *getAllStageData()* method is invoked, which might be overwritten. All stages are visited one after another and their measurements are received through the implemented *getCurrentValue()*. The resulting values are assumed to have an order such that the largest value represents the stage with the best performance. At properties, like the throughput of a stage, this is the native order. If we would measure the needed execution time per element, the results are ordered the other way around. This has to be handled by implementing a specific collection method in *getAllStageData()*. Please notice that the sequence of measurements results in values that are not received at exactly the same time. A time stamp may not be the same between two measured values. Stages with invalid values can not be counted as measured and are excluded by the given algorithm. After all stages are visited, the values are saved by the *History*. This class is initialised with a certain size. Every stage can only have as much data

assigned as the value of this size. This border is only applied if an already saved stage gets a new value. If a stage can not be measured in one iteration, old values remain in the list. The latest system data is saved and our history is up to date.

Figure 5.5 displays the general workflow of the update mechanism. The dashed lined boxes represent loops. The first box iterates through all observed stages and measures their current values. With these data the second box is used. Again the loop iterates through the remaining stages. New Entries are created if needed and the old data is deleted if the set history size is exceeded. So the first box represents the behaviour of the *metric* and the second one of the *history*.

The saved and analysed data is retrieved by the second method, *getSystemState()*. A call to this functions invokes the usage of the selected *AnalysisAlgorithm*. The history is given to this algorithm. For each stage available all measured and saved values are taken and a single value is computed with them. An example is the computation of the mean of these values. At the end of this algorithm every stage has a single corresponding analysed value. These values have a minimum of zero but their maximum is not predefined. To simplify decisions based on the analysed properties and to give a better comparability, we provide another method *getNormalizedSystemState()*. Here we choose to normalise all of these values by their maximum. So we first search for the maximum and then divide all values by it. An exception would be if the maximum is zero. This implies that all stage values are zero and hence all are as good and as bad as the others. Now we have reduced our value range to zero to one. A number closer to 1 implies a better performance and vice versa. The overall result of this function call is a map, with the stages and an analysis how well they are doing, compared to the other measured stages in the system.

Figure 5.6 displays the workflow of the analysis part. Since the normalisation is more complex, the starting point is a call to the *getNormalizedSystemState()* method. Currently this is only done by the *AssignmentAdaptationThread*, but in the future it may also be used by other components. The first step is to retrieve all saved data from the *History*. These data contain all stages that were ever measured and a certain number of values corresponding to them. The chosen *AnalysisAlgorithm* computes the expected performance trend for every given stage. At the end all computed values are normalised by the maximum. We decided to separate the update functionality and the analysis part, to enable intermediate access to the system state without compromising the data.

As mentioned before, our approach doesn't provide support for negative measurements. If a metric is chosen that returns such values, care should be taken that they are transformed into positive numbers. The easiest way would be to provide a suitable implementation of the *getAllStageData()* method. A second possibility would be a corresponding *AnalysisAlgorithm* and an enforcement to use the metric and this algorithm at the same time. But this would reduce the generality of approach of changeable system parts

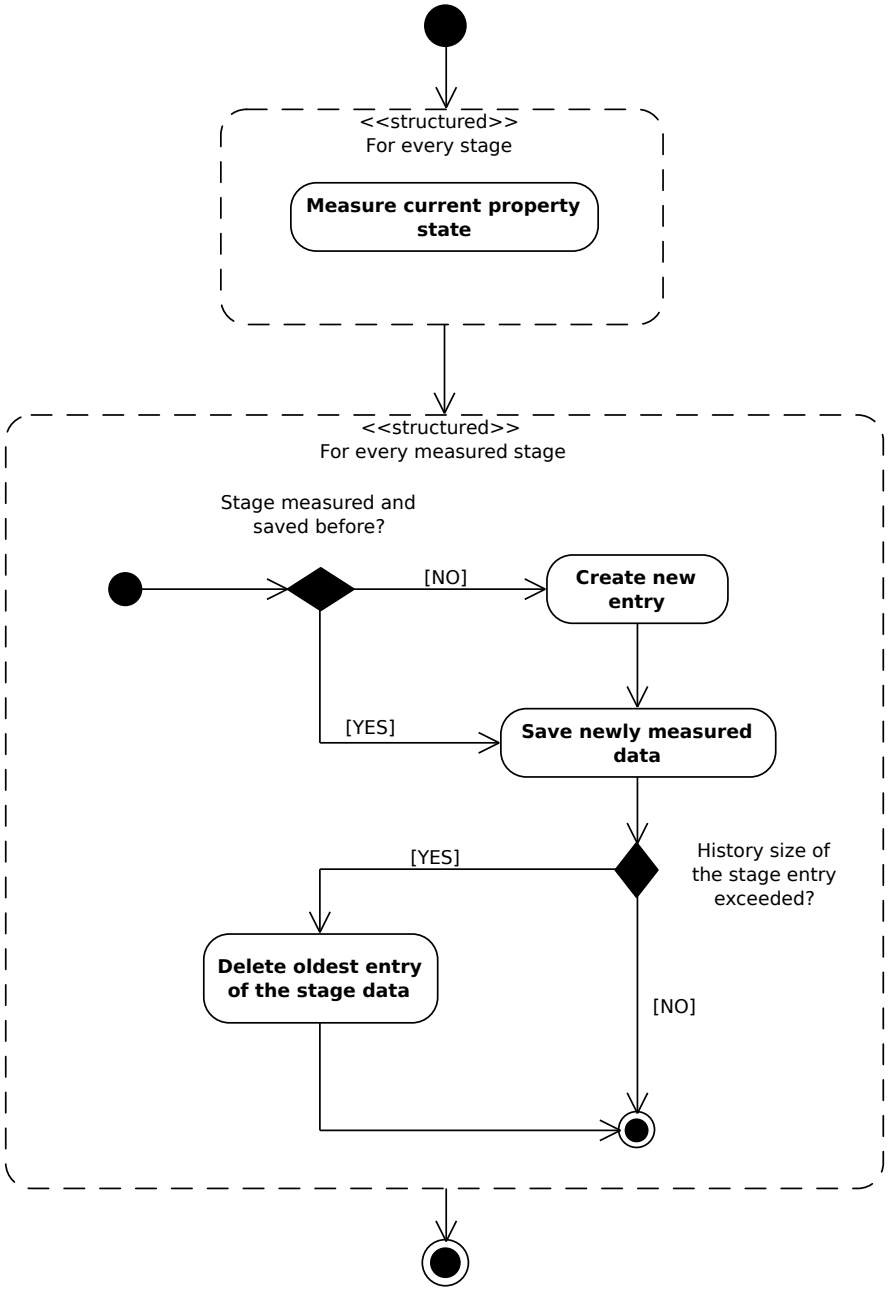


Figure 5.5. The workflow of the analyse part updating the measurements

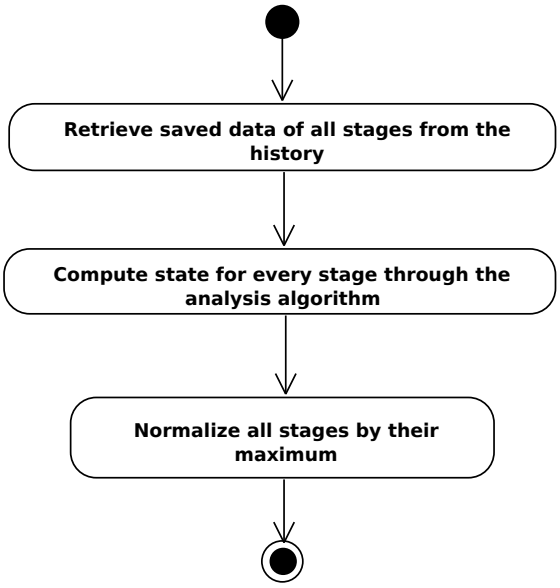


Figure 5.6. The workflow of the analyse part delivering the current normalised system state

5.6 Implemented Metrics

We implemented some metrics as example to prove our concept and display how our extension can be used. Every metric measures a certain property of a single stage, for example the average number of items in the incoming pipes. Already this simple property could be altered by the choice of the maximum or minimum instead of the average. Hence the field of monitoring needs further tuning. As a current issue remains that not all of our metrics are able to measure synchronisation overhead that results from replacing the *UnsynchedPipe* with its bounded or unbounded synchronised version. While the first pipe executes nearly the same code as by a simple function call to pass the elements, the overhead of second class of pipes and especially of the unbounded one is bigger and can delay the execution. In all required pipes we implement the *IMonitorablePipe* interface that is provided by *TeeTime*, if this is necessary.

All implemented metrics use the incoming pipes to measure the properties. With this method producer stages are not measured. Since they are fixed and can not be passive, this behaviour is fine. Other implementations may decide to handle this issue in a different manner. Even in other parts of the configuration metrics can be applied. Generally if a property can be observed, a metric can be implemented to use it.

PullThroughputMetric In this metric the basic idea is that the throughput of the incoming pipes represents how much elements the stage can process and hence how fast it is. We specified this observation and measured only the pull throughput, since this is associated directly to the number of items consumed by the stage. If there is more than one pipe, the mean of all values is computed. Pipes that weren't used and hence have a value of zero, are not measured. All results are already in the assumed ascending order. Additionally a throughput less than zero is not possible.

PushPullDifferenceMetric The next approach origins from the idea of measuring the development of the execution by the unprocessed elements that are waiting in the pipes. Therefore we want to apply a metric that can measure this property. The actual value is easily approximated by computing the difference of the push throughput of a pipe, which represents all added elements and the pull throughput. Again the mean of all used input pipes is calculated for the overall stage value. In this metric a high number implies that the stage where it occurs may be one of the bottlenecks in the system. Since a large number is a bad value and zero is the optimal state, the expected order is not met. Furthermore beside zero all other positive values may be reached, making it difficult to map the results immediately in a right order. To eliminate this problem we overwrite the *getAllStageData()* method, which collects all data. After all stages are measured, we swap the largest value with the smallest, the second largest with the second smallest and so on. Thereby the mapping is unified in a way that fast stages with the same value are mapped to the same slow value. This combines some values but avoids viewing fast stages as slower than they would be.

5.7 Behaviour of the Implemented Thread Assignments

As well as with the metrics we also implemented some assignment algorithms to provide example implementations and prove the validity of our extension. The assignment algorithms offer more freedom to the implementation, since they are not dependent on measurements of properties that may first hove to be implemented in the framework itself. All metrics should be usable by all assignments.

DefaultThreadAssignment The *DefaultThreadAssignment* is used, like the name indicates, as the default assignment of the extension. Since it is *static*, the generated overhead is low and no adaptations are made at runtime. Additionally in the initialisation phase no changes are done to the thread assignment and the behaviour desired by the user is kept. This represents the current behaviour of *TeeTime* without considering the added overhead.

SimpleAssignment In this simple assignment version we place an upper limit for the available threads. This limit is calculated by the number of assigned processors multiplied

by two to optimize hyper-threading and avoid over-utilisation. In some other approaches there are even restrictions that only allow one thread per available core [Min and Eom 2015; Suleman et al. 2010]. In this first approach we don't distinguish between real and virtual cores. The same holds for the other assignment algorithms that initialise the configuration in this way. This can be improved in the future.

In the initialisation phase all producer stages are gathered. Then the remaining number of threads is computed. If some are still available, we assign them to the stages in descending order by the number of in- and output ports. Other initial measurements are also possible but we want to keep it simple. If two stages have the same quantity, the selection is arbitrary. The adaptation at runtime is quite simple. The slowest stage is activated and if we need a free thread, the fastest one is set as passive. All stages are iterated through until we find a suitable one. Only if we can keep our limitation, a stage is activated. It may happen that the number of available threads is lower than the stages that have to be active due to the framework. In this situation the metric and the assignment are called but the latter is not able to change the configuration.

FDPStageAssignment The second and more complex dynamic assignment combines the approaches of *FDP* [Suleman et al. 2010] and *Flextream* [Hormati et al. 2009]. Similar to the *SimpleAssignment* the assignment desires an optimal usage of the assigned processors. The number of threads in the system is limited by the processor count, which is multiplied by two. Again this is a soft limit and may be exceeded if more threads are absolutely needed. During the initialisation the configuration graph is traversed and the required producer threads are calculated. Additionally every stage and its corresponding number of input and output pipes is counted. The remaining threads are distributed amongst the stages with the highest pipe count, since they may be the most important ones for the execution. This initialisation is inspired by the *Flextream* approach. The idea for the runtime adaptation algorithm is taken from the *FDP* approach. A central element we added is the *index* of the configuration and its subsets. The index computes the sum of all stage states, whose values can range from 0 to 1. This sum is divided by number of used stages, to get the average index of the system. A system state near 1 implies that most stages are near the optimum. Additionally we want to gather data on every thread in the system.

We start the dynamic adaptation by gathering all used threads in the configuration. These threads form partitions of the execution graph, which is saved along with some attributes of the partition. Some of the saved properties are the start stage, stages neighbouring other partitions, the slowest stage and the partition index. These data are used in two different behaviour patterns.

The first is the *optimisation mode*. It tries to optimise the executed configuration by reassigning the threads. The *FDP* approach only tries to activate the bottleneck of the system if there is a free thread. We implemented two variants of this assignment. In the first, we try to deactivate a stage if we need one. This still happens in the *optimisation mode*.

The activation of the stages, while there are no remaining available threads, is the most

complex procedure in this assignment algorithm. In first case the slowest stage is already active and has no successors. If we don't improve the bottleneck, no further improvements can be done in this system. The only course of action we can take is to signal that the stage should be duplicated, if possible. In case we need more threads we have to free one. Therefore two partitions have to be fused.

In the first case the bottleneck has more succeeding stages in the same partition, one of them is activated to reduce the load of the thread. This way the computational effort can be reduced one by one. Threads are again freed by fusing partitions. In case the slowest stage is not the starting stage of its partition and no thread is free, we try to fuse the last stage of the partition with the successor partition. If one thread is available, we can just activate the slow stage. Our second version is closer to the original idea. It isolates the optimisation from the second mode and only activates a stage if a thread is available. It will not try to free one here.

The second mode is the *power saving mode*. It will try to free threads and therefore to reduce the needed resources of the system. Since our used framework specifies some restrictions, we can't simply set the fastest stage as passive. The saving mode searches for the two neighbouring partitions that have the best combined index. In other words the two fastest neighbouring partitions are searched and fused. Thereby exactly one thread is freed. Naturally only stages that can be passive are deactivated and their partition is fused with the only predecessor.

The workflow of both versions is the same. The algorithm starts with the *optimisation mode*. If the average index is close to 1, we switch to the *power saving mode*. Else we try to adapt the slowest stage as described. In case the optimisation can't improve the execution, the mode is also switched. Now the saving mode tries to fuse the fastest neighbouring partitions to free threads until this is impossible or the performance is worsened. Meeting this conditions causes the algorithm to switch modes again.

The behaviour should resemble the one shown in Figure 3.1. In contrary to the original *FDP* approach we can't deactivate or duplicate arbitrary stages in our system and have to work with the partitions to free threads. This leads to a divergent behaviour of the algorithms. An idea resulting from this circumstances is to base the assignment solely on partitions and to balance them. To distinguish from this potential new approach the assignment is named *FDPStageAssignment*. The *Flextream* approach [Hormati et al. 2009] also works with partitions and tries to refine it and may inspire more algorithms.

StableTopDownAssignment Since our current implementation is not able to duplicate a stage, the idea of this assignment is adapted from [Guggi and Rinner 2013]. We start with activating all stages. Therefore every stage itself should provide its optimal performance. The bottleneck is still the limiting factor in our system. In every iteration the fastest stage is set as passive. At the moment where this procedure slows down the bottleneck, we revert our last assignment. An important assumption in this case is that the computational effort of the stages does not change. This assignment can be used in cases where the

computational effort is stable and hence the optimal solution too. Furthermore the best assignment could not be computed beforehand.

StableBottomUpAssignment Following the approach that in many use cases computational efforts don't change very much during the execution, we build a second assignment algorithm that tries to find the best thread to stage assignment during the current execution. In the bottom-up variant we start with only the necessary active stages. During the execution the algorithm searches the slowest stage and tries to activate it. If it is already active, all successors are activated one by an other. If the performance drops in this process, the changes are reverted to the last assignment. All tried assignments are saved and the system only changes if the constellation wasn't already tried. Hence, at some point we don't have further assignments that can be tried. This algorithm also uses the assumption that improving non-bottleneck stages will not improve the whole system.

Chapter 6

Evaluation of the Feasibility and the Performance

We had to modify part of the code from our presented approach and our example implementation in *TeeTime*. This enables the framework to change the active and passive state of stages at runtime. First we want to show that our approach is feasible. Thereby, this execution state and hence the amount of used threads can be altered. During this process we also show that this property has a big influence on the throughput of the system. To enable a meaningful usage of these change mechanisms we need information on the system state. Therefore we implemented sensor elements to the existing code. These sensors are located in the pipes and create some overhead every time an element passes through them. In this chapter we also want to evaluate this created overhead and compare our extended pipe implementation to the original one. Putting these parts together, we implemented some dynamic assignment algorithms as proof of concept and as a first exploration of feasible policies. Thereby, an interesting point is, whether we can improve the performance of *TeeTime* with these basic algorithms

According to this the following chapter is divided into six parts. At first we explain the used evaluation methodology and the systems where these tests are executed. Then we present the chosen test scenarios and how they will behave. In the third section we start with the actual evaluation and show the feasibility tests. Additionally, the influence of used threads and the throughput of the system is shown. After this the added pipe sensors are tested. Then the last evaluation is done, where the single assignment algorithms, presented in Chapter 5, are compared to the original behaviour of *TeeTime*. At last we display some threats to validity and describe which points may be improved in future performance tests.

6.1 Evaluation Methodology

During a performance test of Java applications various factors have to be taken into account to gain reliable results. Some of them are described by [Blackburn et al. 2006], who develop the Java Benchmark Suite *DaCapo* and by [Georges et al. 2007].

Amongst other [Blackburn et al. 2006] provide two key points for the chosen test scenarios. The first is to use “diverse real applications”. A Java application should be tested in all situations it will be used in. This guarantees for the test to cover a big part of the

space of use cases and give an overview of the expected behaviour in the real world. Using only good or bad use cases should be avoided. The second point is the "ease of use". Testing an application should be easy. A need for finding the right settings or a difficult configuration before the test can be started should be avoided.

Additionally, they show that different executions of the same program can result in divergent measurements. Especially the continuous successively executed test runs deliver varying values. To avoid these obstacles and receive stable results, we apply a sufficient number of warm-up runs before we start our real runs, which are used for the statistics. They conclude that they "can draw dramatically divergent conclusions by simply selecting a particular iteration, virtual machine, heap size, architecture, or benchmark." Therefore, we also apply multiple executions of the same performance benchmarks to gather enough data for a proper statistic evaluation.

We use three different systems to run our performance tests on. They all differ in the built in processor techniques and architectures. Table 6.1 shows an overview of the used systems. In the future we will refer to them as they are labelled in the top of the table.

	INTEL	AMD	SUN
CPU	Intel Xeon E5-2609	AMD Opteron 2384	UltrasparcT2+
Clock Frequency	2.53 GHz	2.7 GHz	1.4 GHz
Number of Cores	4	4	8
RAM	24 GB	16 GB	64 GB
OS	Debian	Debian	Solaris 10
Kernel Version	$3.16.0-4$	$3.16.0-4$	Version 5.11
JVM Version	$7u91-2.6.3-1\ deb8u1$	$7u91-2.6.3-1\ deb8u1$	$1.8.0\_60-b27$

6.2 Variable Scenarios Used in the Evaluation

Since *TeeTime* does not yet come with build in performance benchmarks, we created a variable computational effort configuration for this purpose. Thereby, we refer to the computational effort each element creates if it is processed in a single stage and the total amount of these elements. The structure of the configurations is pretty straight forward. We always start with one single producer stage. It has a single output port and the number of produced elements can be chosen. The output is a random number. The second part of our benchmark configuration are the consumer stages. Each consumer consists of one input and one output port. The computational effort can be chosen at initialisation. The stage iterates through a loop as often as the variable defining the computational effort intends it to do. Again in every iteration a random number is created and it is multiplied with the value defining the computational effort to avoid possible compiler optimisations which would eliminate this loop. After the loop has finished the latest result is sent to the next pipe.

The configuration itself uses these two stages to create an architecture with a custom number of consumer stages. Also a quota that describes how often a new stage will be

set as a high computational effort stage, is expected. For example a value of 3 will cause every third consumer stage to have a high computational effort. In general if we set this number to n every nth stage is a high computational effort stage and we could divide our configuration with size x in x/n parts with a length of n. Naturally if the total size of the configuration is not a multiple of n, the last part may be shorter than n. We refrain form using random settings in the variable computational effort, to sustain comparability of the results. A more diverse distribution may be possible with more granularity of the settings through parameters, but will result in a more complex interface. Through our created variable benchmark configuration we can achieve the mentioned "ease of use". Unfortunately the created *pipe-and-filter* architectures don't represent all possible "real applications". Instead we created three different scenarios that may display classes where an autonomic adaptation at runtime may yield different results. This may depend on the behaviour of the chosen assignment algorithm.

In our first scenario we consider a system in which every consumer stage has a high computational effort. In theory, giving each stage it's own thread, we should be able to improve the performance, especially since the stages are designed to have the same work to do. No stage is set as active by the configuration, so every algorithm can conduct its own strategy uninfluenced. In the default setting all the stages execute the loop 200 times per element.

In the second test configuration exist only consumer stages that have a low computational effort. The loop, described before, is only executed 10 times. This way the stages have some work, but not too much. Adding more threads to activate stages may result in no improvements. The performance may even get worse than before, for example due to synchronisation overhead.

The third and last test case is a mixed one. It has stages with low computational effort and some with high computational effort. In our default configuration we choose to let every third stage be a high computational effort consumer. Our smallest test starts with one producer and five consumer stages, but the structure is repeated in the style of the shown configuration. Figure 6.1 shows a mixed configuration with only four stages. Hence, the number of created consumer stages was set to 3. The first stage is the producer stage and its thread is therefore executing all following stages by default. The last pointed pipe hints the potential extension with more consumer stages. Since we set the high computational effort ratio to 2, stage **HWC1**, framed in blue, is the only consumer stage in the system with a high computational effort. Thereby, **HWC** and **LWC** denominate the consumer stages with high respectively low computational effort. In this situation an algorithm may be able to improve the performance of the configuration, if it activates the right stages. The shown consumer part is repeated as often as it fits in the chosen total size. The last part may be smaller than two stages. Thereby, stages at the end will be left out. Here, **HWC2** does not exist in the configuration with size 3. If we increase the size to 4 this stage is created at the beginning and appended to **LWC2**. All stages are executed by **Thread1** which belongs to the producer stage. In this example if we would set the size to 6 instead of 4, the part in

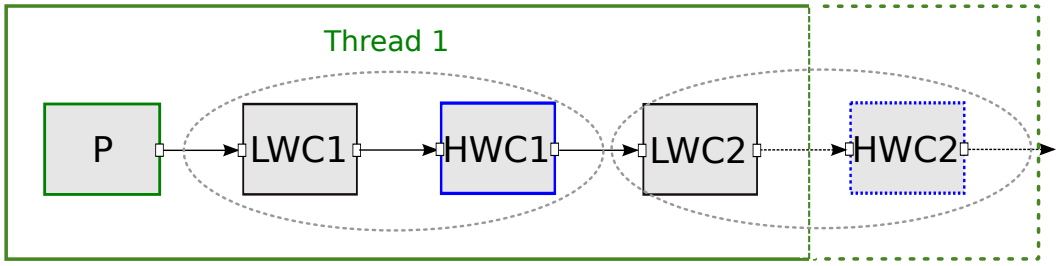


Figure 6.1. A variable *Pipe-and-Filter* architecture drawn as a graph

the grey oval is simply created again and appended to **HWC2**.

We decide not to include I/O-operations in our scenarios or even to build benchmarks around them. Contrary to other approaches, we didn't implement the stage duplication. Therefore the I/O -operations, which may reduce potential parallelism, does not influence our current extension.

The most important variable in an adaptive system is the number of stages. For some assignment algorithms the available CPU resources are also important for the resulting execution. We want to give a first comparison of our implemented algorithms with the performance of the original behaviour of *TeeTime* and amongst each other in different situations. To create these situations we vary the number of stages in each scenario from five to thirty. The amount is increased by five for every performance test. Not every algorithm is dependent on the number of available cores and we want to give a first simple evaluation of our extension. Hence we will keep the amount of available cores the same as given by each system and will not alter any other properties. Since we don't want to stress the system during the changes, we choose not to change the *timeToWait*. It remains at 200 milliseconds.

6.3 Feasibility of the Extension

In our first evaluation we want to show the feasibility of our approach. For this we first show that our approach enables the dynamic changes to the system that are needed for the self-adaptive *pipe-and-filter* architecture. Additionally we show that the number of active stages in a configuration has influence to the throughput of the system.

We implemented a new assignment algorithm to create a high number of changes in each iteration. The intention is to improve the visibility of the changes and their influence made by our extension. The assignment is named *AlternatingAssignment*. As the name suggest the algorithm tries to toggle every stage in each iteration, if possible. So stages, except the producer that were active before the current iteration of the algorithm are set as passive and all passive stages are handled respectively. For this feasibility evaluation we added a special monitoring system. Every time the *AssignmentAdaptationThread* computes

the new assignment and before the changes are applied, it triggers the feasibility monitoring. The current system time, in nanoseconds, as well as the number of the currently active stages are saved. The throughput of the system, measured by the *PullThroughputMetric* is summed up and also saved along with the previous data. Hence, we gather the number of active stages and their corresponding throughput. We do not observe which stage may be active.

For the feasibility evaluation we choose a mixed performance configuration. Our configuration is build with 10 consumer stages. Every second stage is a high computational effort stage with 200 loop iterations and 10,000 elements pass through every stage. We employ 10 warm-up runs and 5 runs afterwards that are used to gather the data. Since the *AlternatingAssignment* behaves non-deterministic and we want to discuss the data of certain timestamps of the measurement, we don't apply statistical operations here.

The *AlternatingAssignment* works as follows. In the beginning before the execution starts we set every second stage as active. Since we save all stages in a *HashSet* the order of its elements can differ between different executions. During the changing phases the algorithm does not use the gathered monitoring data. Instead it takes every stage and sets it as active, if it has been passive. If it has been active the stage is set as passive. Naturally the underlying mechanisms of the effectors don't allow changes in certain situations, for example if a stage is still changing from the last command. Nevertheless we can say that the passive or active stage is toggled in every iteration of the *AssignmentAdaptationThread*. The interval of the measured data is implicitly set through the execution time of the control loop.

Since the behaviour of the *AlternatingAssignment* is non-deterministic and we want to discuss the data of certain timestamps of the measurement, we don't apply statistical operations here. Instead we pick one of each runs per system. The main characteristics of each run are the same and give insights to the behaviour of the *pipe-and-filter* architecture on different systems and how they react to the frequently change of the active and passive stages. We want to compare the behaviour on the different hardware settings. All of them have different specifications and don't operate with the same performance. Hence the *AssignmentAdaptationThread* takes a different amount of iterations throughout the total duration of the benchmark program. So for our diagrams we display the minimum number of iterations from all three chosen measurements. These original diagrams are placed in the appendix due to their size. Here we will show only excerpts of them.

In our test benchmark the used configuration includes five stages with a higher computational effort. Every such stage has to execute 20 times more operations per element than the low computational effort variant. Since these stages have a need for more computation than the others, they are the bottlenecks in our architecture. Like in the other approaches giving them more resources should result in a better work balance for the threads and generate a higher throughput of the whole system. Hence including the producer stage we could assume that the best performance can be reached with 6 active stages, five of them distributed over the high computational effort stages. Please note that the throughput is

very different in every test case. Hence the scale of the y-axis, which shows the throughput, vary in every diagram. The data of the chosen runs can also be found in the appendix.

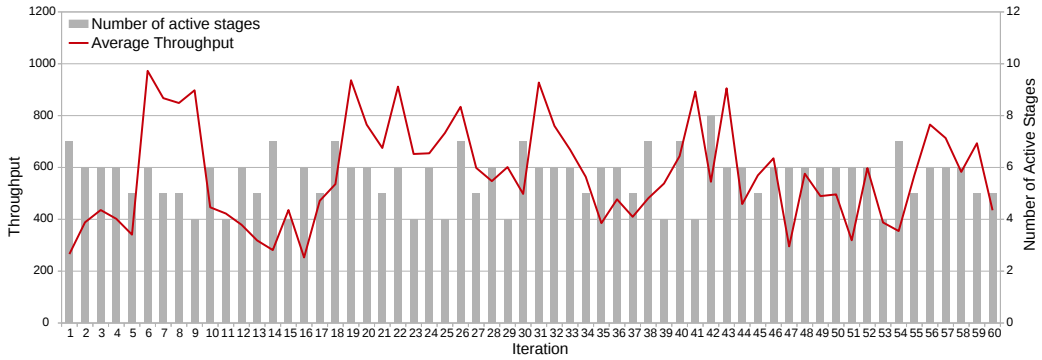


Figure 6.2. Feasibility test on the INTEL with 10 consumer stages, every second stage has high computational effort and the AlternatingAssignment was used

In Figure 6.2 a part of the feasibility test on the *INTEL* system is shown. The complete diagram is found in Figure A.1. We choose 60 continuous values of our bigger diagram. These interval are taken from the 54*th* to the 113*th* iteration. We only display the total number of active stages and don't extract the particular stages. At first glance on the algorithm we could suspect that in every iteration we likely would have a fixed number of 6 running threads in each iteration. This could be the case since we start with 5 active of 10 consumer stages and toggle their state in every iteration. In our first observation we ascertain that this assumption is not true. Due to delays during the pipe changing and combined with the arbitrary thread schedule, stages may be still changing in the following iteration. Since changes still in progress block further (de-)activation, the number of active stages can vary. In the displayed part it ranges from 4 to 8 threads. The total run uses every number of threads between 2 and 10. In general the behaviour of this certain selected run on the *INTEL* is seemingly stable, even in consideration of the complete run. The only displayed spike is in Figure A.1 in the Appendix around iteration 30. This spike settles down quickly.

Now we consider the correlation between the number of active stages and the total throughput of the system. Noticeably, we often get a worse performance if we just put more threads in the program. For example in iteration 42 we employ 8 active stages and only receive a throughput of roughly 600. This is nearly the same amount as in iteration 58 where we have only 6 threads. At the same time in iteration 9 the algorithm activated 3 consumer stages and the throughput peeks with a value of 900. The highest performance in this diagram is 4.75 times faster than the slowest iteration. If we don't limit our values to the same range as the other example, this factor scales up to 17.5.

Even if we employ the same amount of threads, it is not guaranteed to receive a similar

speed-up. This leads to two conclusions. The first is an obvious one. The throughput of the stages and therefore the system is dependent on which stages are activated. A system with partitions that have different total computational efforts will perform worse than with equally distributed threads. On the other hand, if we employ more threads than necessary the performance can drop heavily. Our chosen scenarios seem to be very sensible in this point.

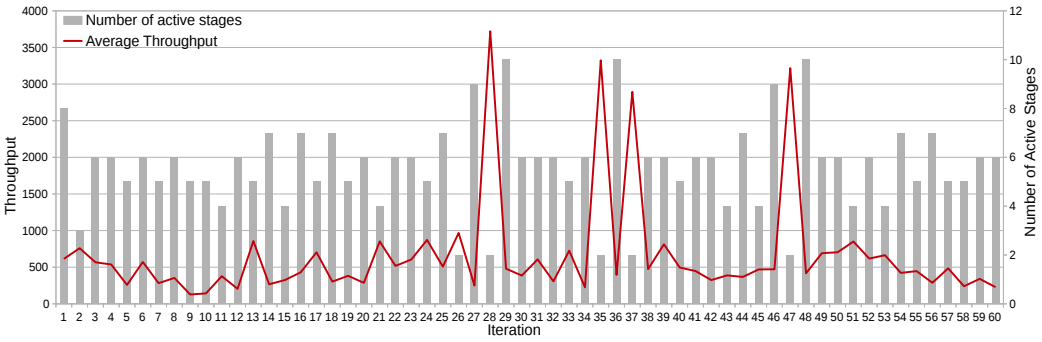


Figure 6.3. Feasibility test on the AMD with 10 consumer stages, every second stage has high computational effort and the AlternatingAssignment was used

In Figure A.2 the results for the *AMD* are found. Again we discuss a part of it in Figure 6.3. Here we cut out iteration 41 to 100. The first observation is that this benchmark varies in the number of active stages more than the first example. This can be caused by the fact that the *AMD* may be a little faster and runaways through delays may not be balanced as quickly through other delays or the scheduler employs an other strategy. On the other hand it may just be a arbitrary property of our example. The number of simultaneously active stages ranges from 2 to 10, exploring most of the possible values. The second and most important issue are the bigger performance spikes. The highest throughput is reached ín iteration 28 of Figure 6.3. The minimum is reached earlier and the throughput is only 127. In the whole run the iteration with the most throughput is 29 times faster than the slowest one. Thereby, nearly all iterations with the highest throughput use only two active stages. Though not every time only two threads are used the performance is good. An example for this is iteration 26. Overall the throughput is the best in the *AMD* system and has a huge gap to the *Intel*

Again we can conclude that the choice of the active stages is important. Other than the first example the speed-up gained by employing more threads is very low. While in the *INTEL* we get the best results with four active stages, in the *AMD* the results indeed get better. However, they are by far not as good as the iterations scenario with solely two threads. This behaviour can be caused by the better performance of the used system. Here the computational effort may be processed fast enough that the synchronisation mechanisms are clearly more expensive than the gain of parallel execution of different

stages. Also the big maximum difference between the throughput spikes and the average performance contributes to this assumption.

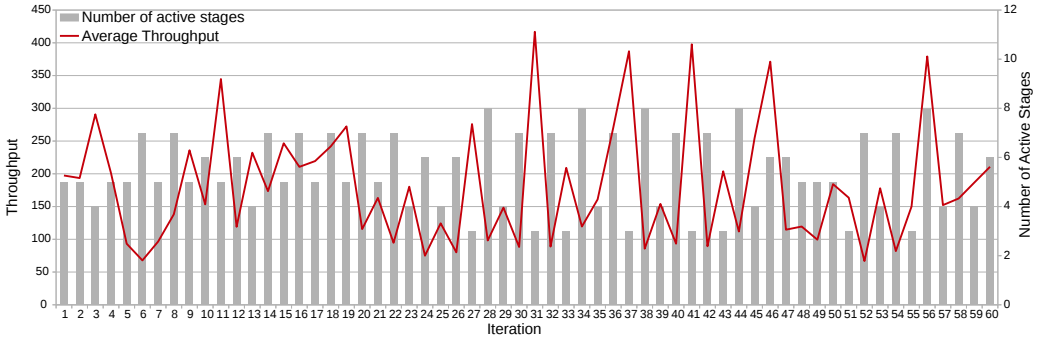


Figure 6.4. Feasibility test on the SUN with 10 consumer stages, every second stage has high computational effort and the AlternatingAssignment was used

In Figure A.3 the measurements on the *SUN* are shown. Again we choose an interval of sixty values to improve the representation. This smaller part is displayed in Figure 6.4. This time we focus on the iterations from 90 to 149. On this system we see parts that have a balanced and stable amount of active stages. There are also parts where the number of used threads changes more keenly. Thereby, the performance gain is not as clearly correlated to the used resources as in the two experiments before. The bigger diagram displays a range of active stages from 3 to 8. In the whole run all possibilities from 1 to 10 are used. In Figure 6.4 the local maximum of the throughputs is reached in iteration 31 with a value of 417. The global maximum is not displayed and lies at 902. With a global minimum of 37 the maximum is nearly 16 times faster than this extremum. In general the *SUN* displays the worst performance of our benchmark environments.

An interesting point in the displayed diagrams is that we do not have such a clear distinction between numerous active stages with bad performance and few active ones with high throughput. Although we can still observe this behaviour, the high number of iterations with comparatively many threads that provide better performance, is noticeable. For example in Figure 6.4 in iteration 56 the throughput nearly peaks up to the local maximum of our diagram. Here 8 threads are employed. Similar behaviour can be observed in and around iteration 10. At the same time there are again many instances where a high number of active stages results in a low performance.

This special behaviour can be caused by the specific architecture of the *Ultrasparc*. The system employs up to 128 real hardware threads and is hence specialized in parallel computing. Therefore additional operations, for example to guarantee the lock-free property, may not be as costly as on the other two systems. The use of multiple threads may not stress the environment as much as the *INTEL* or *AMD*. The remaining issue may be the real delay caused by missing elements, which happens if the pipe is be empty. On the other hand the

SUN only offers 1.4GHz and its behaviour resembles the *INTEL* system, which reached the best performance with four threads. Here a higher number of threads may be employed before the gain is eliminated by the synchronisation. If the partitions of the threads are well balanced, the gain is more likely to compensate for the extra synchronisation effort.

In this section we could show the feasibility of our approach. All stages can be set as active during the execution of the system. This state can then again be toggled to a passive one and vice versa. The number of used threads has great influence on the performance of the system. Using only few active stages can result in a throughput that can still be optimised. On the other hand if we use too many of them the synchronisation costs might surmount the gain of the extra resources. Additionally it can be very important how the partitions, created by the active stages, are composed and if they are balanced. We could observe that every system behaves differently even if we employ the same benchmark configuration. Therefore we can conclude that there can't exist the "perfect" *static* assignment. A main point in the resource distribution seems to be the performance of the system itself. If we use a fast set-up fewer threads may be sufficient depending on the computational effort of the single stages.

6.3.1 Threats to Validity of the Feasibility Evaluation

In this first evaluation of the extension to *TeeTime* we want to show that our approach is feasible and we can influence the execution. In most points we act in accordance to studies about Java performance tests [Blackburn et al. 2006; Georges et al. 2007]. For this we use automatically generated *pipe-and-filter* architectures. These test scenarios give a first overview of our system, but also have some flaws. They include only one producer thread and are one-dimensional without branches.

In this feasibility evaluation we only measure the global state of the executing system. The total number of active stages and the total throughput are regarded. We don't observe which stages are active and how the throughput of the single stages or of the partitions of the threads changes. In addition, the influence of the *timeToWait* parameter and other options still remains to be studied. Since we only use the *AltertnatingAssignment* in this evaluation, the behaviour and impact of strategies implemented for real use cases are not considered.

Another issue may be the used underlying methods that may not be suitable to be used in parallel. A candidate for this may be the used method to create a random number in every iteration. Other methods to create computational effort could be considered in the future. Furthermore, real use cases and scenarios of *TeeTime* may be useful to evaluate the real diversity of its applications.

There exist some external threats to the validity. We evaluate the performance of our extension on three different test systems with different hardware and architecture. The broadness of the underlying hardware could be increased in future tests. It also has to be considered that we use different JVM versions on the *SUN* and the other two systems. Also

different operating systems are used. In the future, our approach can be evaluated with other JVM versions and operating systems.

6.4 Overhead of the Monitored Unsynchronised Pipe

In Chapter 5 we described, how we modified the framework to enable the dynamic adaptation as described by the *MAPE-K* approach [Kephart et al. 2003]. One key point was to add sensors to the *TeeTime* framework. The currently implemented sensors apply metrics to the pipes. To be more precise we additionally measure the throughput of the pipe with extra operations in the *add()* and *removeLast()* methods. These sensor operations were added in the *UnsynchedPipe* and the *UnboundedSynchedPipe*. Since every passing element will trigger these added code parts, we want to know how much they influence the performance of the pipes and how big the created overhead is.

In our evaluation we simulate the original *UnsynchedPipe* and the modified version with our added code. Similar to [Wulf et al. 2016] we use *JMH* [*Java Microbenchmark Harness*] to build these performance tests. We exclude the *UnboundedSynchedPipe* from our tests since, the produced overhead will be roughly the same and in the future the underlying queue can be changed such that it provides measurement similar to the *BoundedSynchedPipe*.

In this benchmark we measure the throughput of the simulated pipe. The method we use as a benchmark for the pipe implementations is quite simple and the same for both types. We initiate the benchmark with one of these types and use this object through the entire run. During the test an element is put into the pipe and immediately after retrieved. This is repeated until the default time limit of *JMH* is reached. This limit is set to one second by default and not changed here. For the benchmark of each pipe we employ five runs as warmup and ten real runs for the measurements. All tests are repeated in three forks, giving a base of thirty measurements for the result.

In Figure 6.5 we show how both tested pipes behave in the benchmark. Again we employ the three different architectures as the test systems. Therefore we compare the monitored pipes with the corresponding result of the original pipe. The latter is set as reference and we give the percentage of the throughput reached by each pipe. Hence the *UnsynchedPipe* is always 100%. Please note that our y-axis begins at 70% and the gap between both values may appear bigger than they are in reality. All used data can be found in Tabular A.4 in the Appendix.

The benchmark on the *INTEL* provides similar results for both pipes. The modified one is even slightly better than the old one. If we consider the measurement error of the real data, we can reason that both implementations behave mostly equal. If the values are considered with their error range, of both pipes overlap each other.
In the *AMD* system we observe the biggest performance drop due to the modified pipe. The new variant with the implemented sensor functions is about 19% slower than the original pipe. This can further add to the reasons of the overhead with multiple active stages, like seen in the previous section.

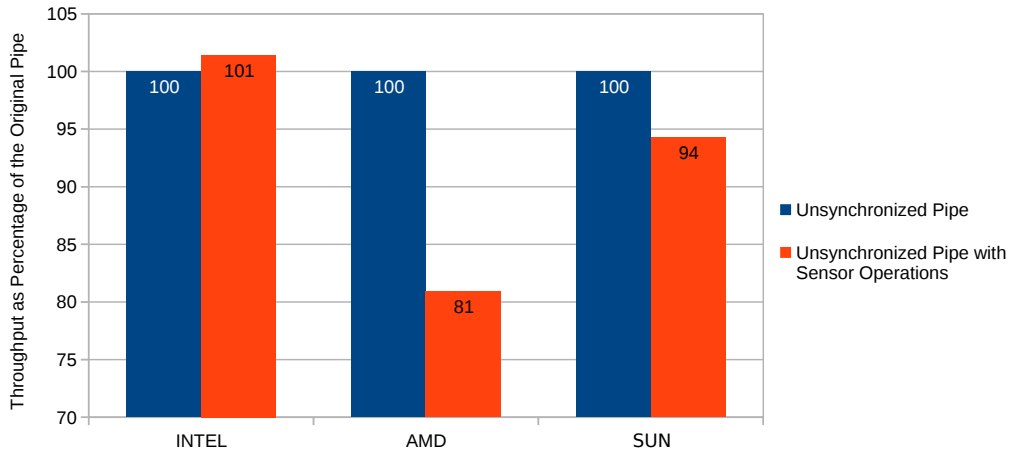


Figure 6.5. Comparison between the UnsynchedPipe without and with sensors

Also the benchmark for the *SUN* shows similar results to the *INTEL*. The pipe with the sensor operations is approximately 6% slower than the one without these operations. The added time to the executions is clearly visible, but it is not big enough to be essential for the total performance.
We can conclude that our sensors naturally add to the overhead of the pipes during each transfer of one element. In two of our three test systems this overhead is low. However in the *AMD* the added burden needs to be considered and further insights on this issue are needed. This may be improved in the future, if other sensor methods are chosen.

6.4.1 Threats to Validity of the Overhead Evaluation

In this evaluation we study the overhead of our changes in the *UnsynchedPipe*. We employ the same procedure as in the benchmarks provided by *TeeTime* [Wulf et al. 2014]. Hence, our results provide a comparable quality and we can argue on the same level. As mentioned before, the modified *UnboundedSynchedPipe* is not evaluated directly. In the future the overhead of this pipe should also be measured. Especially, a comparison between underlying queue implementations that already provide monitoring data and our added sensor operations could improve the system in the future. Even though we simulated the behaviour with *JMH*, the implementation of the pipes in *TeeTime* and their behaviour in real use cases was not evaluated. In this overhead evaluation the same external threats as in the feasibility measurements exist.

6.5 First Performance Evaluation of the Adaptive Assignment Algorithms

In the last part of our evaluation we want to compare the performance of our implemented thread to stage assignments with the uninfluenced behaviour of *TeeTime*, which will online activate producer stages and stages with two or more different preceding threads. We use the three already described test benchmarks for low, mixed and high computational efforts. The number of consumer stages are varied in steps of 5. At first the configuration consists of the producer stage and five consumer stages. The last benchmark uses 30 consumer stages in each computational effort scenario. In each execution run the producer provides $20,000$ elements. The variables for the computational effort are set as described, 200 for high computational effort stages and 10 for the ones with low computational effort. Again we execute every performance test on each of our three example systems. Thereby, every time 5 warm-up runs and 10 real runs were executed to gather stable results. The values further used are the averages of every such benchmark. All measures taken can be found in the Appendix.

As representation of the standard behaviour of *TeeTime* the *DefaultAssignment* is given. It just implements the standard behaviour of the framework as a static assignment. Since our benchmark configurations contain only one producer stage and no branches, the *DefaultAssignment* utilises only 1 thread in every scenario. Furthermore we test all assignment algorithms given in Chapter 5, including both variants of the *FDPStageAssignment*. We want to compare all algorithms with the default values. Hence, we choose not to discuss the measured values directly. Instead we want to consider the ratio of each algorithms compared to the results of the *DefaultAssignment*. For this all data is normalised by the results of this algorithm and we will reason about how much the different approaches deviate from it. Please note, since the results and therefore the ratio vary greatly in each benchmark, the scale of the y-axis it not the same in every image.

6.5.1 Low Computational Effort Performance Tests

We start our discussion with the low computational effort benchmarks. In Figure 6.6 our results on the *INTEL* are shown for this test. At the y-axis the relative execution time is given, in comparison to the *DefaultAssignment*. On the x-axis the different scenarios are listed. Since they mostly vary on the amount of consumer stages, this number is used as the representation. Each algorithm has a single graph with a unique symbol and colour. Here the *DefaultAssignment* is shortly called *Default* and is represented by blue and the triangles pointing to the left. Naturally its values are always 1. The red line with the diamonds represents the relative values of the *SimpleAssignment*, or short *Simple*. The *StableTopDownAssignment*, or *StableTD* and *StableBottomUpAssignment*, or *StableBU*, are represented by the ruby coloured and the turquoise lines, which symbols are the right arrow and the left arrow respectively. The first implementation of the *FDP* approach is

called *FDP1* It is yellow with the arrow down. The second variant, which divides more clearly between power saving and optimisation phase, is called *FDP2*. It is represented by the green line with the arrow up. All symbols stay the same for all following images.

In this low computational effort benchmark on the *INTEL* system no algorithm is able to beat the *Default* assignment. The execution time of the *StableTD* is the worst and up to ten times worse than with the original assignment. *FDP2* is the next in this ranking with a maximum performance loss of factor 6. It is closely followed by the *FDP1*. From 5 to 15 stages *Simple* is also close to the two *FPD's*, but later on it improves and even reaches a factor of roughly 2.2. The best dynamic algorithm is the *StableBU*. With just 5 stages it is as good as the other algorithms, except *StabelTD*. In the 20 stage case *Simple* temporary overtakes it. Otherwise, it is the best tested algorithm on this system. During the 25 stage scenario it even nearly reaches the performance of the default algorithm. Except the *StabelTD* all algorithms have a tendency to draw closer to the *Default* as the number of consumer stages increases.

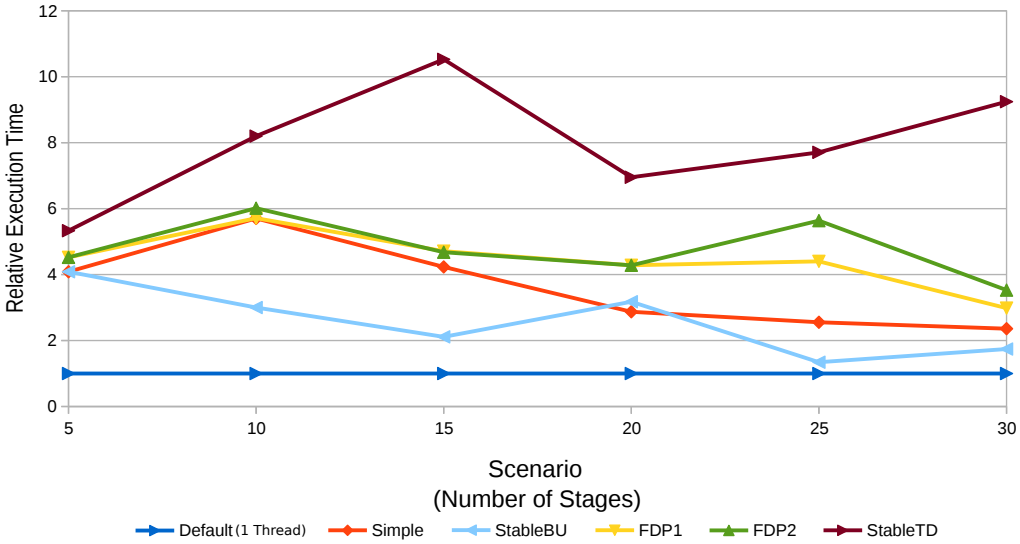


Figure 6.6. Relative execution time in the low computational effort scenarios on the INTEL

In Figure 6.7 the same low computational effort benchmark is executed on the *AMD* system. Again no algorithm is able to improve the performance of the system. The first impression in this diagram is the big deterioration in the usage of the *StableTD*. In general the execution time of this algorithm is around 16 times worse than the default time. Thus it is the worst dynamic algorithm in this figure. All other algorithms improve their performance with increasing consumer stage count. Thereby *FDP1*, *FDP2* and *Simple* behave

very similar. They start with a factor of 6, like the *StableBU*. The factor in other scenarios stays around 4, but *Simple* again decreases to roughly 2.2. *StableBU* is once again the best assignment algorithm. It improves its behaviour constantly and nearly reaches the *Default* in the last scenario with 30 consumers.

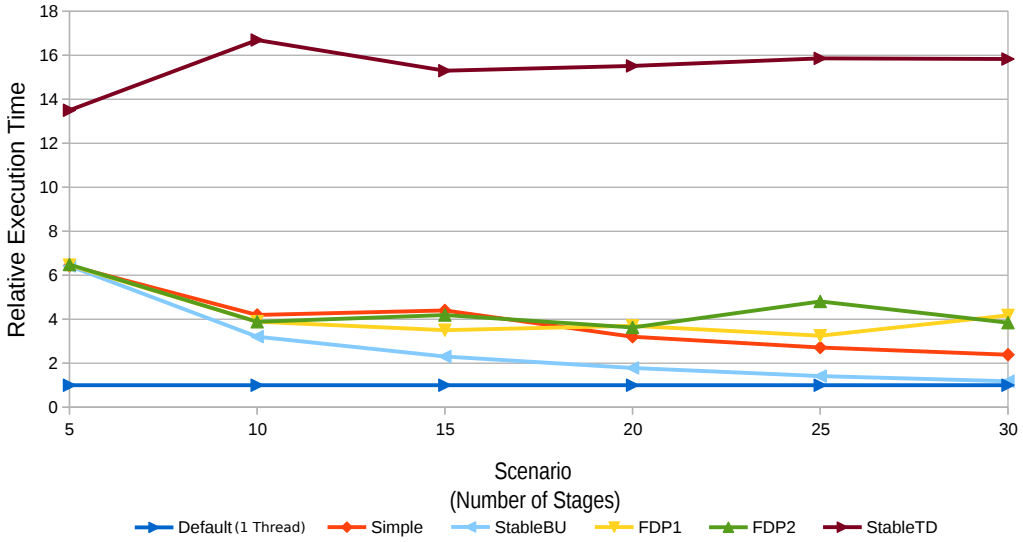


Figure 6.7. Relative execution time in the low computational effort scenarios on the AMD

The last evaluation with this low computational effort benchmarks is done on the *SUN*. Figure 6.8 contains the result of this system. Here we are able to improve the performance compared to the original execution in two scenarios. Again the *StableTD* algorithm is the worst. Its factor goes up to 5. Other than on the previous systems it improves later on and closes the gap to the other results by a big chunk to 2.5. Again the *FDP* assignments and *Simple* behave very similar. The *FDPs* deliver an execution time around twice as high as the *Default* in all scenarios. *Simple* improves this to roughly factor 1.5 with 15 and 20 stages and then goes up again. Here again *StableBU* provides the best results. At 10 and 25 it is even better than the *Default* execution. Here a factor of 0.92 and 0.84 is reached. The error range of the *Default* and *StableBU* overlap each other and this result may also represent a similar behaviour. At the last scenario the behaviour worsens. Thereby it reaches factor 2 again, which is bigger than the result of *FDP* and *Simple*. In general on the *SUN* all dynamic assignments are closer to the *Default* behaviour.

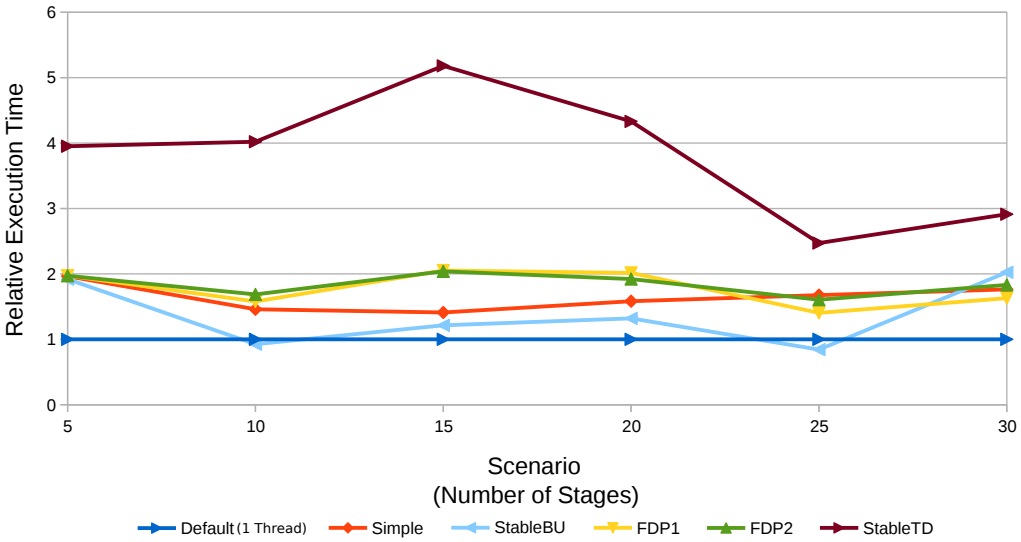


Figure 6.8. Relative execution time in the low computational effort scenarios on the SUN

In the low computational effort benchmarks we could observe different points. In general the assignment algorithms tend to improve if the resources have to be distributed amongst more stages. Even if every added consumer has only a low computational effort, the more stages have to be executed, the more likely it is that a new thread could improve the performance. It is noticeable that the gap between the dynamic algorithms and *Default* widens as the system provides a higher clock frequency. This results in the first real performance improvements on the *SUN*. As seen in the feasibility evaluation the *AMD* is the most sensitive system, if more than two threads are used. The gain from more active stages is here most likely eliminated by the synchronisation overhead. Additionally most algorithms try to distribute resources in a optimal way, and are unlikely to withdraw them completely. This can explain the particular big gap in this results. Especially *StableTD* starts with all stages, set as active, which may cause the bad results. On the other hand *StableBU* only activates a stage permanently if the local throughput is increased. This reduces the employed resources and thereby the added overhead. The other three assignment algorithms mostly provide the same results, with some deviations. Hence, we can conclude that in the low computational effort scenarios, the more complex algorithms fail to achieve their goal compared with the simple *Simple*.

6.5.2 Mixed Computational Effort Performance Tests

In the next part of this section we discuss the results of the mixed computational effort scenarios. Now every third consumer stage is declared as a high computational effort stage.

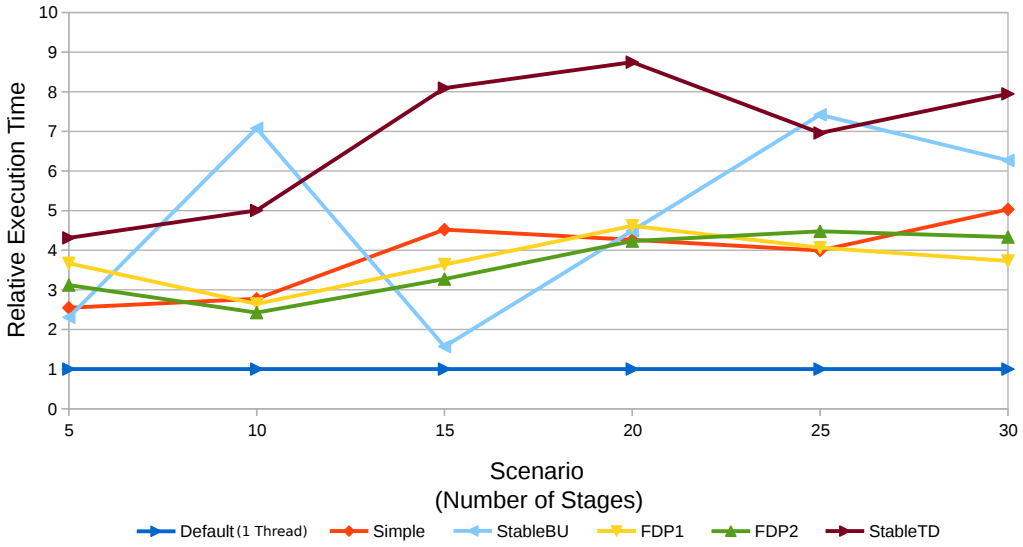


Figure 6.9. Relative execution time in the mixed computational effort scenarios on the INTEL

Again the benchmarks were executed on all three test systems. We start with the *INTEL* in Figure 6.9. Other than the last benchmark on this system the worst factor is 8.75 reached by *StableTD*. Again this algorithm is the worst in all scenarios but two. *FDP1,2* and *Simple* range from 5 to 2.2 and could also reduce the distance to *Default* in general. Their graphs follow similar tendencies and none of them is clearly better. Thus all of the four discussed assignments are closer to *Default* than in the low computational effort scenarios. In this benchmark the *StableBU* algorithm behaves inconsistent. In some scenarios it provides the best results and in others the worst. Due to this strong alternation this strategy does not deliver reliable results in this example.

The results of the mixed benchmarks on the *AMD* are shown in Figure 6.10. Here the results vary strongly from the ones discussed until now. Again the results of the two *FDP* assignments and the *StableTD* are closer to 1. The first two variants perform better with the increasing stage count and become by far the best and reach roughly 1.3. *StableTD* is still the worst in most cases, but also gets better as the configuration grows bigger. These three algorithms reduce the range of the normalised factor, compared to the low computational effort case. Thereby, the maximum factor is also reduced to 13. Both *FPDs* behave very similar and nearly reach the *Default* algorithm, but are not able to get better results. On the other hand *Simple* and *StableBU* deteriorate as more consumer stages are used. Thereby, *StableBU* is a little erratic and becomes the worst at 30.

The last results of the mixed computational effort benchmarks were made on the *SUN*. Figure 6.11 displays the results of the tests. Again we were not able to break through

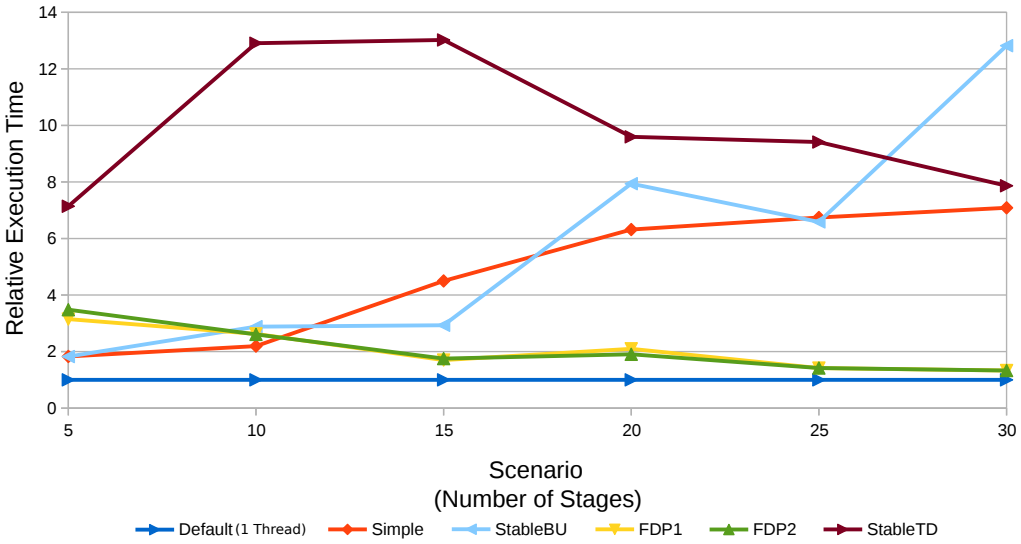


Figure 6.10. Relative execution time in the mixed computational effort scenarios on the AMD

the wall given by *Default*. In the scenario with 20 consumer stages the real data reveals a great performance drop in the *Default* algorithm. Since the other algorithms don't have this anomaly, the comparing graphs show a drop for most of the dynamic ones. The ranking is again more similar to the same benchmark on *INTEL* than to the *AMD*. *StableTD* is still the slowest algorithm. *FDP1*, *FDP2* and *Simple* behave similar again. At the last scenario the *FDP* variants show a tendency toward better results. This trend may correlate to the *FDP* results on the *AMD*. This also contrasts the results of the other strategies, which tend to worsen if more stages are used. *StableBU* creates the best results for most amounts of stages. Even so in the anomaly at 20 stages this algorithm is the worst, which may indicate a spike, like seen at the *AMD*. At the last scenario the algorithms reach a factor of around 3. Therefore, overall the gap increases on the *SUN* compared to the former low computational effort benchmark.

In the mixed computational effort benchmarks we can observe different key points of the behaviour of the different assignment strategies. Additionally, the results vary once again between each system. Overall on the *Intel* and the *AMD* the gap between the assignment algorithms and *Default* closes. Contrary to this the performance of the compared strategies got worse on the *SUN* in general. The ranking of the algorithms is mostly the same on every system. *StableTD* is usually the worst algorithm. *FDP1,2* and *Simple* are always close to each other, except on the *AMD*. On the *AMD* the assignments in the *FDP* style provide the best resource usage and approach the *Default*. On the other hand the simple *Simple* algorithm clearly can't find a good solution. The results of *StabelBU* are inconsistent between the

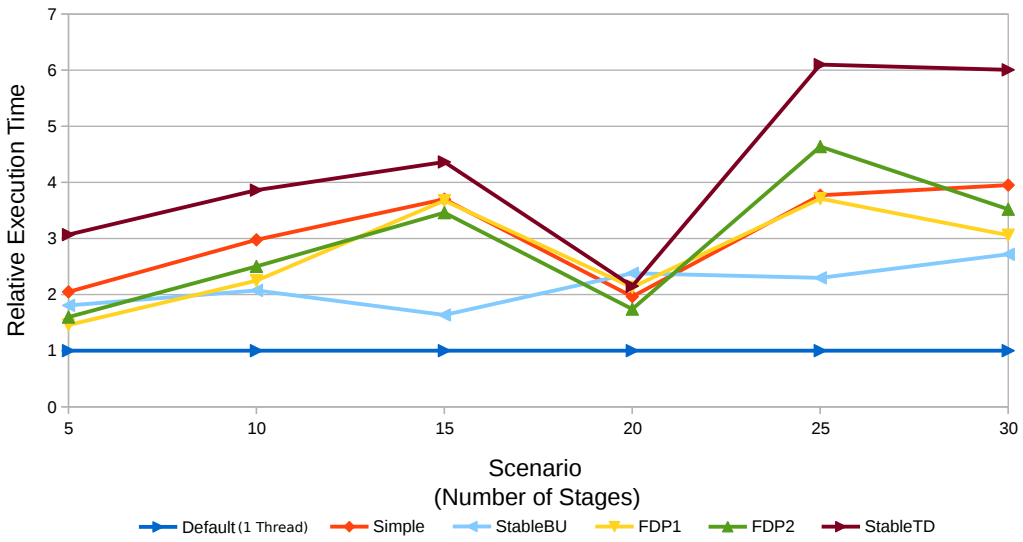


Figure 6.11. Relative execution time in the mixed computational effort scenarios on the SUN

systems. On the *SUN* it is once again the best. On the *AMD* is starts mediocre and worsens as more stages are used. Additionally, it behaves arbitrary on the *INTEL*, where it alternates between good and bad results and ultimately gets worse. Therefore, other than before the more complex algorithms seem to get the better results in the mixed computational effort. Though not all of them can adapt and improve to the scenarios on all systems.

6.5.3 High Computational Effort Performance Tests

The last part of this performance evaluation provides uniform scenarios again. Here we employ high computational effort configurations. These configurations are similar constructed as the ones used in the last two benchmarks. Now every stage is set as a high computational effort stage. Hence, this is similar to the low computational effort benchmark, but the overhead created by active stages may be more likely compensated by the added computational power.

Once again we start with the *INTEL* system. Figure 6.12 shows the relative results of this benchmark. Compared to the mixed computational effort results, overall the ratios are higher again. No strategy falls below a ratio of 3. Hence, the gained speed-up of the item processing can't compensate the overhead here. Most of the time *StableTD* is once again the slowest algorithm. Though it closes the gap to the other strategies and even reaches them, for example at 20 stages. *Simple* often follows as the next best algorithm. In these runs *FDP1* shows some differences in the behaviour compared with its sibling. Both alternate

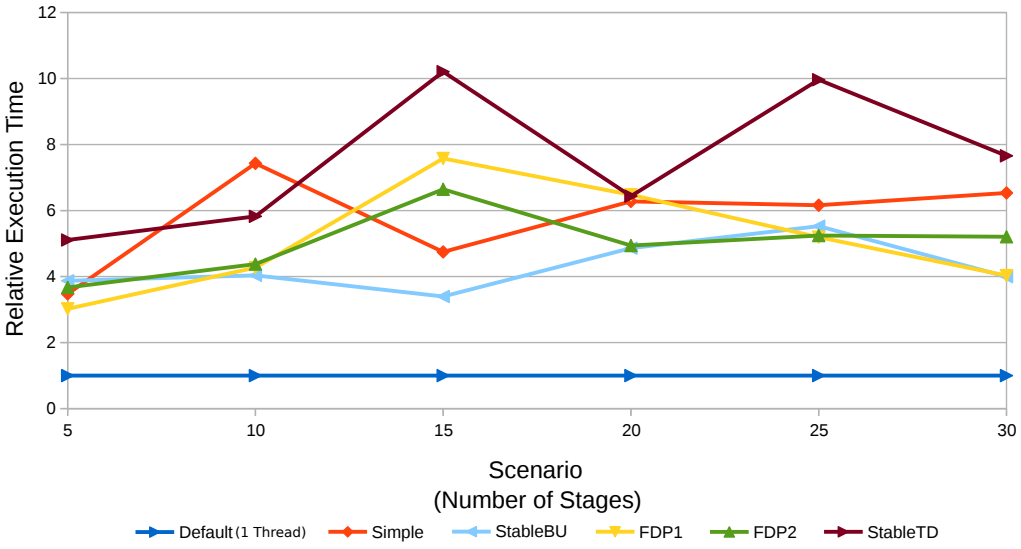


Figure 6.12. Relative execution time in the high computational effort scenarios on the INTEL

in the performance ranking and in the last scenario *FDP1* is slightly better. The results of *StableBU* are always one of the best, but they don't clearly overtake the other ones. In general all strategies only get slightly better or worse as the configuration size increases. This is contrary to the other benchmarks on the *INTEL*.

Figure 6.13 presents the results for the same benchmarks on the *AMD*. Compared to the low and mixed computational effort scenarios the algorithms behave differently. Again we can't beat the *Default* algorithm. In fact the results have properties of both previous runs on this system. First of all *StableTD* still provides the worst performance and uses up to 22.7 times the execution time of the *Default* algorithm. As in the mixed case *Simple* worsens as the configuration size increases and ranks as the second worst assignment algorithm of this test. Unlike in the low computational effort benchmark *StableBU* can't improve the performance and gets even worse. It has not the same erratic behaviour as in the mixed case and follows *Simple* as the third worst algorithm. The best results are achieved by the *FDP* assignments. In this diagram *FDP1* can provide execution times close to the values of *Default*, but it doesn't reach them. Especially the scenarios with 15, 20 and 25 stages result in the values 1.32, 1.26 and 1.20 respectively. *FDP2* can't reach the same performance as its sibling. Only at 15 stages both algorithms are close to each other. While both *FDPs* get closer to the *Default* and roughly stay there as the number of stages increases, *StableTD* and *Simple* get worse at the same time. *StableBU* worsens until 20 stages. Afterwards it shows a tendency to improve the performance again.

Figure 6.14 displays the last benchmark on the *SUN*. Here all results are far away from

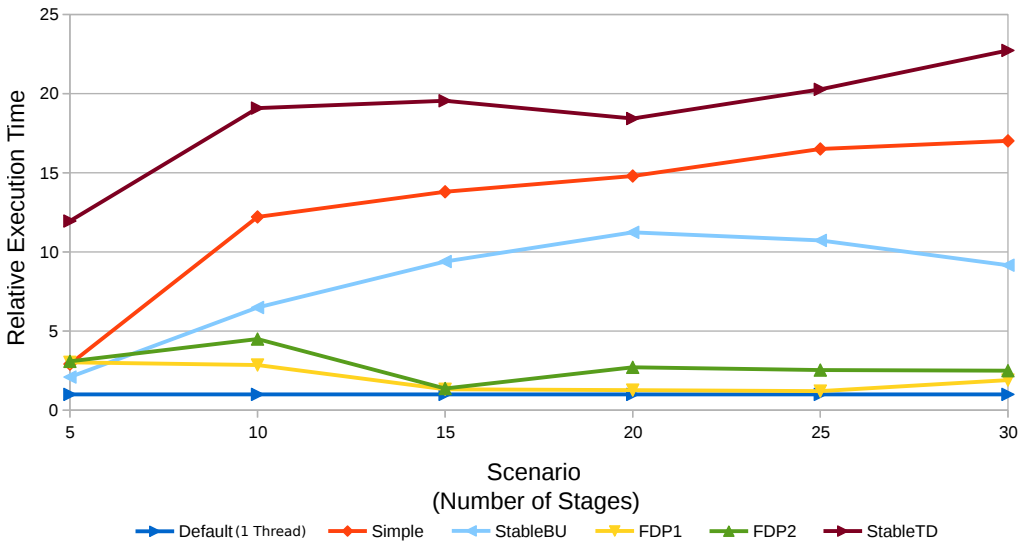


Figure 6.13. Relative execution time in the high computational effort scenarios on the AMD

reaching the *Default*. Like before we can observe properties of the both earlier benchmarks. All strategies are close to each other and can't be ranked as clearly as before. This is like in the low overhead benchmark. As the number of stages increases the factor of each algorithm increases until 15, respectively 20 stages are reached. At 25 and 30 stages all values range between 4 and 5. This tendency is also observed in the mixed computational effort scenarios. Even the *StableBU*, which has the best results in both earlier benchmarks only provides a mixed performance, compared to the other algorithms. This also applies to *StableTD*, which has the worst results in every other diagram.

In all high computational effort benchmarks we could observe a behaviour with portions from the ones provides by the low and mixed computational effort scenarios. In general the gap of the respectively unsuited strategies to *Default* increased. In fact on the *SUN* and the *INTEL* the performance dropped further. Although the stages utilize more computations per element the synchronisation overhead could not be compensated. On the *AMD* the good tendency of the *FDP* assignments is kept. They are not influenced by the added computational efforts like the other strategies. Even though *StableBU* is worse than in the other benchmarks, it stopped its erratic behaviour from the mixed one.

6.5.4 Conclusion of the First Performance Evaluation

In all presented benchmarks we are not able to reach a clear improvement of the execution time compared to the default behaviour of *TeeTime*. The sole point where we can provide

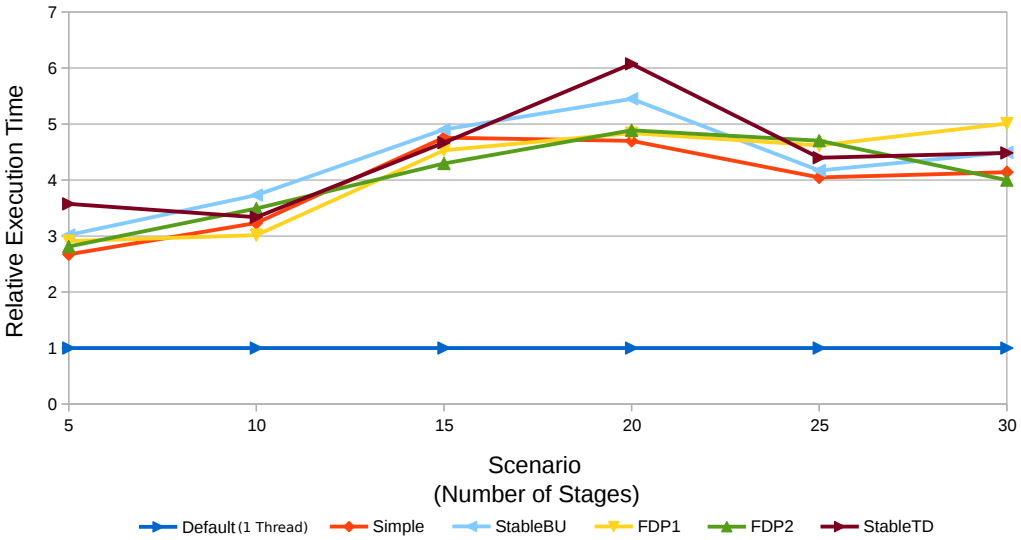


Figure 6.14. Relative execution time in the high computational effort scenarios on the SUN

a real increase of the performance is in the low computational effort benchmark on the *SUN*. Especially in these scenarios we can observe algorithms coming close to the value of *Default*. However, in all cases the values and their error range have to be considered, especially at the two points that breach the default border. Although the measurements show the increasing execution time for all algorithms, the set computational effort of the high computational effort stages may be not sufficient to make an activation worthwhile. An other point can be underlying the used methods of our evaluation scenarios that maybe can't be computed in parallel.

Generally, *StableBU* provides the best performance of all dynamic algorithms in the low computational effort benchmark. This changes in the mixed one. Here this algorithm provides erratic results. In the high computational effort scenario this stops, but the results are not as good as in the first. This algorithm needs further exploration and improvements. On the other hand *StableTD* is always the worst assignment algorithm. Since it starts with all stages set as active and stops completely if it finds that the total performance has dropped, it tends to use more active stages. Hence, the synchronisation overhead can turn out really big and may be an explanation for these results. In general a threshold value that catches insignificant variations in the performance monitoring, can improve the behaviour of both *Stable* algorithms. Thus, more parameter studies are needed.

The *FDP* assignments often provide the second or third best performance throughout the benchmarks. Especially on the *AMD* they are the best at the mixed and high computational efforts. In general they use all possible resources, but not more, and try to distribute them

in an optimal way. Therefore, most of the time they will never deactivate stages unless for redistribution of the threads. Since the *AMD* only provides 4 hardware threads and the algorithms limits the usage of threads accordingly, these strategies automatically limit the resource consumption and may even do less changes due to this fact. On the *INTEL* and *SUN* more threads are provided and thus more resources will be used. In turn this increases the synchronisation overhead. Even though both algorithms are close to each other, *FDP1* often provides slightly better results.

Simple provides its best results in the low overhead scenarios. The performance is mediocre in the low computational effort benchmarks and gets worse in the mixed and high ones. Since this algorithm just activates the slowest and deactivates the fastest stage, if needed, it does not balance the computational effort on the stages in any way.

In general we could observe that the behaviour of the single strategies strongly depends on the executed scenario and the used machine. Thereby, the number of threads and the provided performance seem to be important. Hence, future assignment algorithms have to consider the provided resources and adjust themselves accordingly. All of our implemented policies seem to lack the ability to improve or even get the same results as the *Default*. Especially, they can't compensate the synchronisation overhead. In the future algorithms should be considerate that *TeeTime* seems to be very sensible in this point of our benchmark scenarios. They should only use the minimum of the provided resources that improves the execution. Based on this evaluation *StableBU* and one of the *FDP* assignments may be the best candidates for further improvements.

6.5.5 Threats to Validity of the First Performance Evaluation

In this performance evaluation we use again the variable computational effort configurations. Our test scenarios that are built with these configurations are scalable in multiple ways and create comparability between the single tests, their size and their computational effort. Thereby, many options can be regulated according to the planned scenario. Again all configurations in all scenarios are one-dimensional systems. We have no branches or feedback loops. In the whole system exists only one producer stage, whose thread also drives the execution in the *Default* strategy. This can be extended to create more complex configurations. Also, we classify stages either as high- or low-computational effort stage. The behaviour and their effort is the same in the same computational effort class. The high computational effort may be too small to create enough computational effort to compensate the costs of the synchronisation mechanisms. Another issue may be again that underlying methods used in the variable configurations may not be able to be used in parallel.

In general many settings and options are not touched in this evaluation. Like before, the *timeToWait* of the *AssignmentAdaptationThread* are not changed. The only used *AnalysisAlgorithm* is the regression algorithm, which provided the best results in the task farm [Wiechmann 2015]. Other already implemented or new algorithms can be used in future evaluations. Future benchmarks may also consider changing more aspects of the dynamic adaptation, like the used *metrics*, the sizes of the pipes, the size of the *History* and many

more. Again, the external threads to validity from the previous evaluations are still valid here.

6.6 A Second Performance Evaluation on the INTEL

In our previous evaluations we experienced some unexpected behaviour of all dynamic algorithms and in the feasibility tests. Due to this we investigated our benchmark scenarios further and changed the behaviour of the stages that provide a variable computational effort. In our first version we utilised a loop to generate a certain load for the stage. In this loop a random number generator is used. Since this generator often is a singleton and not optimised for parallel usage, we replace the whole loop and use a method of *JMH* [*Java Microbenchmark Harness*]. The computational effort variable is now given to the *consume()* method of the *Blackhole* provided by *JMH*. It creates computational effort on the CPU dependent on the given number. The computational effort rises "almost linear" [*Java Microbenchmark Harness*]. We increase the computational effort of the high computational effort stages to 2000. The remaining settings are kept as in the other performance evaluation. Also the algorithms and the number of investigated consumer stages stay the same as before. In this thesis we only evaluate this adapted benchmark on the *INTEL*. In the future other systems have to be evaluated as well. Here we consider the mixed and high computational effort scenarios. In the case where every stage has only a low computational effort per element, we still can't observe that our dynamic algorithms create a better execution time. The overhead for the synchronisation is still to high too be compensated through the gain of extra threads. The real data and the low effort scenario are given in the appendix.

We begin our discussion with Figure 6.15. Here the results of the mixed computational effort scenarios ar displayed. The mapping between colours and symbols and the dynamic assignment algorithms stays the same as before. Again the execution times are normalised by the result of the *Default* assignment. Here we can achieve real improvements of the performance. All algorithms tend to get better as the amount of consumer stages increases, like in the other tests. Unlike in all previous results *StableTD* gives the best results. At the end the algorithm performs more than 5 times as fast as the *Default* execution. Hereby, an interesting observation is that in every run the execution times stay nearly the same. The other algorithms start with an execution time greater than 1 and begin to improve from 15, respectively 20 stages. The *FDPs* are the second best and reach up to 0.4 of the original execution time. The differences between the two siblings melt as the number of stages increases. *Simple* and *StableBU* perform the worst of all dynamic algorithms and reach their best performance with a factor of around 0.75 at 30 stages. Again *StableBU* shows indications of an erratic behaviour.

In Figure 6.16 the benchmark with the high computational effort scenarios is shown. Again, all consumer stages in this scenarios have a high computational effort. Like before, most algorithms perform better as the number of stages increases. *StableTD* provides the best results, like in the mixed scenarios. Here it even reaches an execution time which is

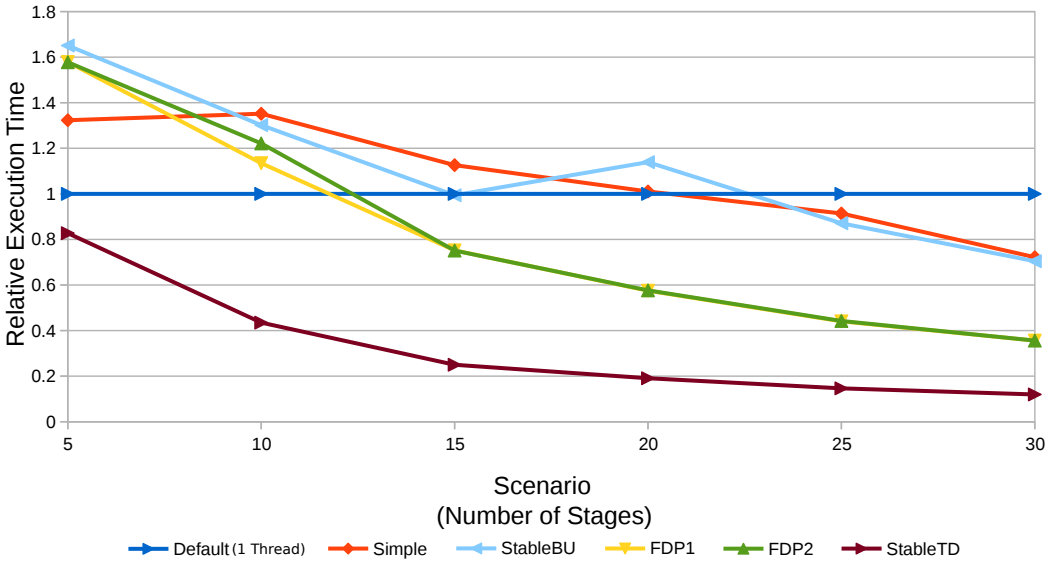


Figure 6.15. Relative execution time in the mixed computational effort scenarios on the INTEL

ten times faster than the one of the *Default*. In this benchmark the execution times are also very similar in every run. With 5 stages most algorithms don't find a better assignment than *Default*. The results of the *FDPs* differ slightly. They perform up to 5 times better than *Default*. *Simple* is the second last in this ranking. *StableBU* performs the worst and the execution time goes up again in the 30 stage scenario. However from 15 stages onwards all algorithms achieve a better performance than the *Default* behaviour.

In the evaluation with the second version of our variable configuration we can greatly improve the execution of our chosen benchmark scenarios. Other than in the previous evaluation all algorithms are able to improve the performance of *TeeTime*. Thereby, *StableTD* provides the best results. This is contrary to the first results, where the algorithm was always the worst. Especially the fact that every execution needs a similar time to finish has to be studied further. The performance gain of *Simple* and *StableBU* is the lowest. Especially the latter indicates a slightly erratic behaviour again. The *FDPs* improve the performance as the second best algorithms. In consideration of both evaluations the *FDP* algorithms provide the most consistent results. In general, if the load of some stages is high enough for a thread to compensate the synchronisation overhead and there are no hindrances for a parallel execution, our dynamic adaptation algorithms can improve the performance of the system. However, such described obstructions are not diagnosed accordingly by all algorithms and this needs further improvements. If this could be solved *StableTD*, can potentially be one of the best algorithms.

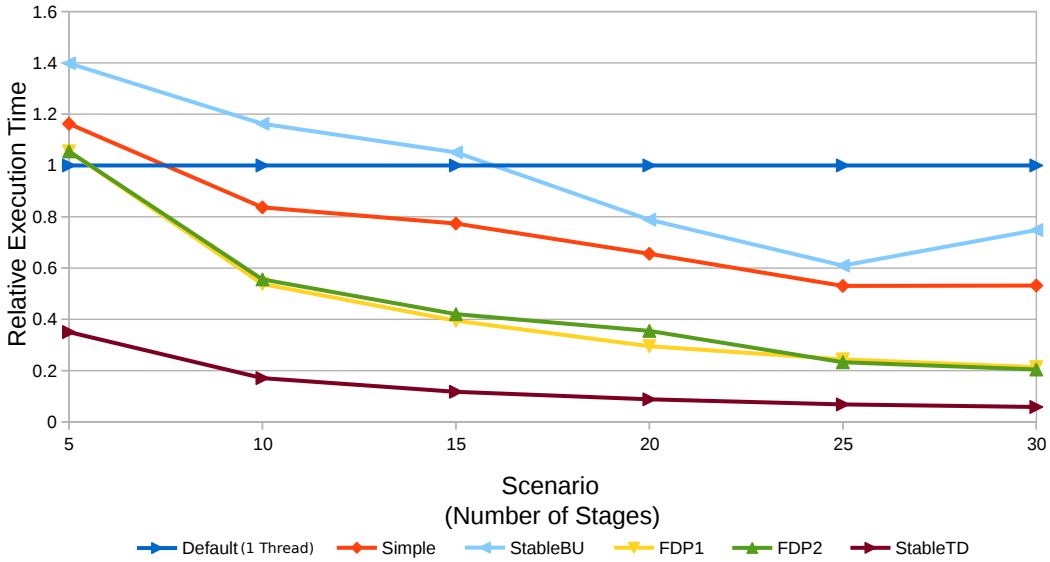


Figure 6.16. Relative execution time in the high computational effort scenarios on the INTEL

6.6.1 Threats to Validity of the Second Performance Evaluation

In the second performance evaluation, we execute our benchmark scenarios only on the *INTEL* with the same JVM as before. Through the modifications in our variable computational effort configurations we increase the potential for parallel execution in the scenarios. Since we use other methods to produce the computational effort, the parameters, which define this effort, are not comparable. In this evaluation the resulting execution times of all algorithms differ from the ones measured before. Even though we could improve the performance of the default behaviour of *TeeTime*, we did not compare our dynamic algorithms with good static thread assignments. Also the scenarios remain simple and have no branches or loops. This should be considered in future performance evaluations. The remaining threats to validity essentially stay the same for this benchmark as in the first performance evaluation.

Chapter 7

Related Work

The *pipe-and-filter* architectural style has many applications and similar fields, especially the field of stream processing. It often splits the input stream into multiple elements. Each element out of this stream has to go through,in multiple single steps to complete the whole procession. Additionally, there exist many special use cases for the *pipe-and-filter* architecture. Each of the created application may have different properties, like stateless stage. Also the underlying hardware may vary. Hence, in the literature many approaches to adapt and optimise such programs can be found. Thereby, often stage replication is used to balance slow stages. Here we display some of the approaches and discuss how they generally work and how they differ to our approach .

In the *Feedback-Directed Pipeline Parallelism* (*FDP*) approach [Suleman et al. 2010] the authors want to adapt a given *pipe-and-filter* architectures to reach two goals. First, they want to find an optimal thread to stage assignment for this application. Second, the unneeded utilisation of cores and hence their power consumption should be avoided and minimised. The origin of their optimised application is loop parallelisation through pipeline parallelism. They use an implementation where one core can execute multiple stages and a single stage can be processed by multiple cores, if needed. Especially the way how the stages are assigned to the cores is different from our approach. The procedure of their algorithm is used as the base of two assignment algorithms in our work.

The *DMonA* approach [Michiels et al. 2002] is a component bases architecture, which extends the DiPS architecture with monitoring and adaptation strategy components. It divides the system in balanced parts and every part is managed according to its own local strategy and in harmony with the meta strategy of the whole system. The software is generally seen as a *pipe-and-filter* architecture, but with heterogeneous elements, which may need special care. This resembles the *MAPE-K* approach. The adaptation is done through operations like stage parallelism or increased caching of data. Even though the control design is similar to our approach, with replication and controllable caching, their effectors offer other operations to the managed elements.

In the approach of [Guggi and Rinner 2013] it is assumed that all stages run with their maximal throughput. Increasing the performance of a single stage is not possible. To reduce the overhead and the unnecessary used resources the whole system is slowed down to match the speed of the bottleneck. This is ultimately done through a reduction of the speed in which new elements are produced. This differs from our approach, since we don't touch producer stages directly and try to regulate the throughput through the amount of used

threads and their location.
Weir [Burtsev et al. 2014] "is an imperative, streaming programming language". Its main purpose is to be used for scripting, including at the command-line level. Thereby, the main goal is to build analysis algorithms as a *pipe-and-filter* architecture. This framework provides events as control mechanism similar to the signals in *TeeTime*. The execution of these events can be controlled through a scheduler. These last two points are the main difference to our approach. Additionally we don't apply restrictions for the build use cases.
In *Flextream pipe-and-filter* architectures on multicore systems are considered. The goal of the framework is to adapt the application at runtime to changes in the pool of available resources. For example if a second program is running in the same system, it needs resources for itself, like CPU cores. If this program terminates, the allocated resources are freed and the *Flextream* can utilise them. Therefore, this adaptation is event based. During the adaptation the stages and the used cores are balanced. Thereby, the system, represented as a graph, is split into work partitions. This approach differs to our work, as it provides a direct stage to core assignment, reacts to system events and workloads are not considered directly.
The goals of the *DANBI* programming model [Min and Eom 2015] are similar to our work. They want to provide irregular stream applications that are able to handle a varying workload at runtime. The thread to stage assignments, here called schedules, are either event based or rely on probabilities. They are able to use a given amount of CPU cores as well as GPU's. The optimisations they apply to the system are low level. This contrasts our approach, which mostly works with the high level abstractions of the *pipe-and-filter* architectural style, like pipes, ports and stages. They don't follow the *MAPE-K* approach like we do. Additionally they utilize a scheduler for each thread.
In [Chandrasekaran et al. 2003] web services and their workflows are perceived as *pipe-and-filter* architectures. Even though their work is only indirectly related to our approach, they provide the idea to simulate such systems beforehand to generate execution data.
SEDA [Welsh et al. 2001] also models web services as *pipe-and-filter* architectures. Resources are automatically assigned, through the single stages of the service, as the number of requests and hence their workloads increase. The stages in question are replicated to distribute the load. Additionally, several options like thread pool size and the event scheduling may be chosen. The goal in this approach is also the balancing of dynamic workloads at runtime. Through the limitation to stateless web services the replication of stages used for this is kept simple. Since we want to create this balancing for a general purpose framework, our approach does not use this replication as an instrument for load balancing.
[Hirzel et al. 2014] provide several optimisations that can be applied to a stream processing application. They consider these systems to consist of single stages like a *pipe-and-filter* architecture. They present how these applications can be optimised through different operations on the single stages. For example the *fusion* and *fission* of stages is described.

Chapter 8

Conclusions

In the beginning of this thesis we defined three main goals for our extensions. Following the *MAPE-K* approach our first objective was to create effectors, which can control the resource distribution in a *pipe-and-filter* architecture. Thereby, we focused on the distribution of threads to single stages. In this approach only these threads are considered as shared resources. They can be assigned to and withdrawn from the stages. Since we considered execution models in which one active stage with its own thread executes neighbouring passive stages, we had to keep some restrictions in mind. For example in the *push-model*, used by *TeeTime*, producers and stages with multiple preceding threads have to be active all the time. Additionally every thread has only one active stage that is executed by it and vice versa. Our approach for implementing the effectors and providing the possibility to (de-)activate a stage at runtime uses the circumstance that the incoming pipes of it have to change anyway. Here we created intermediate pipes, which are used to handle the occurring concurrency issues and ultimately restore a configuration as it would be constructed by *TeeTime* without the extension. This pipe changing approach to alter the executing thread of a stage is our first contribution. It resembles the *fusion* and *fission* used by some other approaches. For the sensor part we employed a simple pipe monitoring, which uses built-in measurements of the throughput and related metrics. In pipe implementations where this functionality was not given by the underlying structure, we implemented them ourselves.

In the second part we extended *TeeTime* with the autonomic adaptation principle. As suggested by the *MAPE-K* approach, we split our computation in different phases. All of these phases are used one after another by the control loop in the *AssignmentAdaptationThread*. It gathers the monitored data through the sensors and then analyses it. With this analysis it plans how the system should be configured in the next iteration and how this goal should be reached. Thereby, *policies*, also called *assignment algorithms*, are used to determine the course of action. At the end the effectors are used to change the configuration at runtime. The *policies* can be chosen and even be implemented by the user. They can represent the different strategies described in other approaches. In our work we implemented some algorithms as proof of concept. Also the metric, which controls the choice of the monitored property, can be chosen as well as self-implemented. As examples we give the choice between the pull-throughput of the pipes associated with the measured stage and their remaining items. However, other metrics, like the execution time per element, will need additional sensor implementations. Furthermore, other options, like the used analysis algorithm and the time between every iteration of the control loop, can be chosen.

In the last part we evaluated the approach with our the example implementation in *TeeTime*. All tests were done on three different systems. Thereby, we used a variable computational effort generator, which creates simple, periodic configurations with different computational efforts. At first we have shown that our approach is feasible. Stages can be changed from active to passive and vice versa at runtime. Thereby, the choice and the amount of active stages has a great influence on the performance of the system. Our tests show that the throughput of our benchmark scenarios is sensible to the choice of the active stages. The wrong stages or too many threads cause a heavy performance drop. Additionally we have evaluated the overhead of our monitoring operations for the intra-thread communication. Here, we can show that the overhead is low but can't be ignored on some systems. In the last part of the evaluation we measured the throughput of the single assignment algorithms provided by us. All were compared to the default behaviour of *TeeTime*. This benchmark was done in three different scenarios. The first scenario had only stages with low computational effort. The second scenario included only high computational effort stages. In the configurations of the last scenario stages from both types were present. In nearly all cases the default behaviour got better results than our adaptive algorithms. However, they often show a tendency to improve the execution as more stages are used, especially if the stage number is bigger than the amount of available resources. In a further investigation, we replaced the random number generator with an other method to consume CPU cycles. We evaluated the performance again on the *INTEL*. This led to real improvements of the performance of the scenarios. Therefore, our algorithms are able to improve the execution of such scenarios at runtime, but they need further improvements to identify situations where changes can only result in a worse performance. We did not test environments where the computational effort changes dynamically at runtime.

We made the first steps to enable a *pipe-and-filter* framework to dynamically adapt itself and possibly react to computational effort changes during runtime. Thereby, the *MAPE-K* approach was used. In the considered execution models the effectors can mostly be implemented by changing the pipes accordingly. The performance of the execution can be influenced by such dynamic assignments. The results of each algorithm also depend on the chosen hardware and the executed scenarios. Therefore, it is not possible to give a perfect *static* assignment for every situation. The implemented policies have to be further refined to create a good performance even if the given configuration can't benefit from multiple used threads. If this hindrances are not present we could improve the execution in our benchmarks. Dynamic computational efforts weren't considered and have to be evaluated in the future. The design of our extension and the given option to implement new assignment algorithms and metrics should allow a fast refinement of the given policies and even to develop new strategies. A more in-depth analysis of the improvement possibilities of *pipe-and-filter* architectures can help in this procedure. For example, determining optimal static assignments for simple configurations can hint how the results of their dynamic counterparts may look like.

Chapter 9

Future Work

In the future the implemented assignment policies should be tuned to achieve improvements in the performance of all possible scenarios of *TeeTime*. Especially the computational effort balance in the single partitions of the threads has to be considered. Adding more tolerance to the monitored data and limiting erratic behaviour can improve existing algorithms. Additionally, the resource usage should be limited according to the provided hardware. Thereby, it can happen that the options of the algorithms become restricted. An alternative approach would be to apply load balancing algorithms like the honeybee algorithm used in cloud computing [Randles et al. 2010]. In general, computing optimal assignments for several scenarios can be useful to gain insight information about where the current assignments are lacking. Thereby, approaches like the simulating runs beforehand or balancing each stage to have the same throughput may be useful. A comparison between good static assignments and the dynamic algorithms should be done to further explore the gain of our approach.

Also other metrics can be tested and may improve the results. For example measuring the execution time of a stage per element may provide another way to recognise their resource demands. This can result in multiple sensor operations running at the same time. However, this can lead to several extra computations that are applied to every element and thus slowing down the whole system. To avoid this a general monitoring interface that only applies the sensor operations if they are needed for the specific metric, can reduce the overhead. Even composite metrics can be implemented in this way, but the *AnalisesService* and the *History* have to be extended for this purpose. If enough resources are available, procedures like described by [Fittkau and Hasselbring 2015] can be applied to increase the performance if huge configurations or many metrics are used.

Future evaluations should consider a broader range of benchmark scenarios. For example configurations with different branches and stages that vary their computational needs according to the content of an input file provide more complex use cases. Especially the effects of dynamic computational efforts at runtime should be analysed. Additionally, the remaining options, like the time between every iteration of the control loop or the analysis algorithms, can be explored. A general benchmark suite could take over this responsibility. Furthermore, this would make it easier to provide different and reproduceable evaluation results. Thereby, an option to import and export *pipe-and-filter* architectures can improve the comparability of such frameworks in the long run.

To improve the execution itself and utilise more resources different steps can be taken.

Stage duplication or even multiplication can be implemented as suggested in Chapter 4. Also a scheduler for the threads may be useful, especially if more threads are used than cores. Corresponding to the scheduler it may be useful to balance the thread to core assignment directly, as done in other approaches. Also GPU's and distributed systems may be utilised in the future. In the second case the pipe changing approach can also be applied to redistribute stages on computation nodes. Since the pipes control the communication and the execution flow, stages that are not changed don't have to be touched directly. The pipes will do the communication and can be responsible for the connection between the single nodes.

Bibliography

[Blackburn et al. 2006] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. Portland, Oregon, USA: ACM, 2006, pages 169–190. (Cited on pages 59 and 67)

[Bruni et al. 2012] R. Bruni, A. Corradini, F. Gadducci, A. Lluch Lafuente, and A. Vandin. Fundamental approaches to software engineering: 15th international conference, fase 2012. In: Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. Chapter A Conceptual Framework for Adaptation, pages 240–254. (Cited on page 10)

[Burtsev et al. 2014] A. Burtsev, N. Mishrikoti, E. Eide, and R. Ricci. Weir: A Streaming Language for Performance Analysis. *SIGOPS Oper. Syst. Rev.* 48.1 (May 2014), pages 65–70. (Cited on pages 1, 14, and 86)

[Buschmann et al. 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. A system of patterns: Pattern-oriented software architecture (1996). (Cited on page 7)

[Chandrasekaran et al. 2003] S. Chandrasekaran, J. A. Miller, G. S. Silver, B. Arpinar, and A. P. Sheth. Performance Analysis and Simulation of Composite Web Services. *Electronic Markets* 13.2 (2003), pages 120–132. (Cited on pages 14, 46, and 86)

[Chen 2016] J. Chen. *Flow in Games*. Visited: March 7, 2016. 2016. URL: http://www.jenovachen.com/flowingames/. (Cited on page 12)

[Dept. 2016] U. B. E. Dept. *Ptolemy Project Website, Ptolemy 2*. Visited: March 7, 2016. 2016. URL: http://ptolemy.eecs.berkeley.edu/ptolemyII/. (Cited on page 6)

[Fittkau and Hasselbring 2015] F. Fittkau and W. Hasselbring. Elastic application-level monitoring for large software landscapes in the cloud. In: *Service Oriented and Cloud Computing*. Volume 9306. Lecture Notes in Computer Science. Springer-Verlag, Sept. 2015, pages 80–94. (Cited on page 89)

[Georges et al. 2007] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. *SIGPLAN Not.* (2007), pages 57–76. (Cited on pages 59 and 67)

[Guggi and Rinner 2013] H. Guggi and B. Rinner. Increasing Efficiency of Data-flow Based Middleware Systems by Adapting Data Generation. In: *Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on*. IEEE. 2013, pages 189–198. (Cited on pages 14–16, 56, and 85)

[Hirzel et al. 2014] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* 46.4 (Mar. 2014), 46:1–46:34. (Cited on pages 8 and 86)

[Hormati et al. 2009] A. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures. In: *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*. 2009, pages 214–223. (Cited on pages 1, 13, 15, 45, 55, 56)

[Horn 2001] P. Horn. *An architectural blueprint for autonomic computing*. Visited: March 7, 2016. 2001. URL: http://people.scs.carleton.ca/~soma/biosec/readings/autonomic_computing.pdf. (Cited on pages 2, 10, 13, 16, 17)

[*Java Microbenchmark Harness*] *Java Microbenchmark Harness*. Visited: March 7, 2016. 2016. URL: http://openjdk.java.net/projects/code-tools/jmh/. (Cited on pages 68 and 81)

[Kephart et al. 2003] J. Kephart, J. Kephart, D. Chess, C. Boutilier, R. Das, J. O. Kephart, and W. E. Walsh. An architectural blueprint for autonomic computing. *IBM White paper* (2003). (Cited on pages 2, 10–13, 16, 48, and 68)

[Michiels et al. 2002] S. Michiels, L. Desmet, N. Janssens, T. Mahieu, and P. V. DistriNet. Self-adapting Concurrency: The DMonA Architecture. In: *Proceedings of the First Workshop on Self-healing Systems*. ACM, 2002, pages 43–48. (Cited on pages 7, 13, 36, and 85)

[Min and Eom 2015] C. Min and Y. I. Eom. Dynamic Scheduling of Irregular Stream Programs toward Many-Core Scalability. *Parallel and Distributed Systems, IEEE Transactions on* 26.6 (June 2015), pages 1594–1607. (Cited on pages 55 and 86)

[Monroe et al. 1996] R. T. Monroe, A. Kompanek, R. Melton, and D. B. Garlan. Architectural styles, design patterns, and objects. *IEEE software* (1996), page 43. (Cited on pages 1 and 5)

[Randles et al. 2010] M. Randles, D. Lamb, and A. Taleb-Bendiab. A comparative study into distributed load balancing algorithms for cloud computing. In: *Advanced Information Networking and Applications Workshops (WAINA), 2010 IEEE 24th International Conference on*. Apr. 2010, pages 551–556. (Cited on page 89)

[Soulé et al. 2013] R. Soulé, M. I. Gordon, S. Amarasinghe, R. Grimm, and M. Hirzel. Dynamic Expressivity with Static Optimization for Streaming Languages. In: *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. DEBS '13. Arlington, Texas, USA: ACM, 2013, pages 159–170. (Cited on pages 7 and 14)

[Suleman et al. 2010] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt. Feedback-directed Pipeline Parallelism. In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. Vienna, Austria: ACM, 2010, pages 147–156. (Cited on pages 7, 8, 14, 15, 45, 55, and 85)

[Taylor et al. 2009] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009. (Cited on pages 1, 5, 6)

[Van Hoorn 2014] A. van Hoorn. Model-Driven Online Capacity Management for Component-Based Software Systems. Doctoral thesis/PhD. Kiel, Germany, Oct. 2014. (Cited on page 10)

[Van Hoorn et al. 2009a] A. van Hoorn, M. Rohr, I. A. Gul, and W. Hasselbring. An Adaptation Framework Enabling Resource-efficient Operation of Software Systems. In: *Proceedings of the 2nd Warm Up Workshop (WUP 2009) for ACM/IEEE ICSE 2010*. ACM, 2009, pages 41–44. (Cited on page 10)

[Van Hoorn et al. 2009b] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. *Continuos Monitoring of Software Services: Design and Application of the Kieker Framework*. Forschungsbericht. Kiel University, Nov. 2009. (Cited on page 1)

[Van Hoorn et al. 2012] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In: *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM, Apr. 2012, pages 247–248. (Cited on page 1)

[Welsh et al. 2001] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-conditioned, Scalable Internet Services. *SIGOPS Oper. Syst. Rev.* 35.5 (2001). (Cited on pages 6, 47, and 86)

[Weyns and Holvoet 2007] D. Weyns and T. Holvoet. An Architectural Strategy for Self-Adapting Systems. In: *Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '07. Washington, DC, USA: IEEE Computer Society, 2007. (Cited on page 10)

[Wiechmann 2015] C. C. Wiechmann. On Improving the Performance of Pipe-and-Filter Architectures by Adding Support for Self-Adaptive Task Farms. Master's thesis. Kiel University, Oct. 2015. (Cited on pages 19, 35, 42–46, and 80)

[Wulf et al. 2014] C. Wulf, N. C. Ehmke, and W. Hasselbring. Toward a Generic and Concurrency-Aware Pipes & Filters Framework. In: *Symposium on Software Performance 2014: Joint Descartes/Kieker/Palladio Days*. Nov. 2014. (Cited on pages 1, 6, 8, 9, and 69)

[Wulf et al. 2016] C. Wulf, N. C. Ehmke, and W. Hasselbring. The Pipe-and-Filter Architectural Style Revisited: From Basic Concepts toward Smart Framework Implementations. In: Submitted for Review, 2016. (Cited on pages 34 and 68)

[Wulf and Hasselbring 2016] C. Wulf and W. Hasselbring. *Java-based reference implementation of the TeeTime framework*. Visited: March 7, 2016. 2016. URL: http://teetime.sourceforge.net/. (Cited on pages 1, 7–9, and 13)

Appendix A

Appendices

A.1 Diagrams of the Feasibility Evaluation

In the feasibility diagrams the used configuration contained ten consumer stages. Every second consumer stage is a high computational effort stage. The *AlternatingAssignment* is used to activate every passive stage and deactivate every active stage in every iteration. All results are collected with the first version of the variable workload stages.

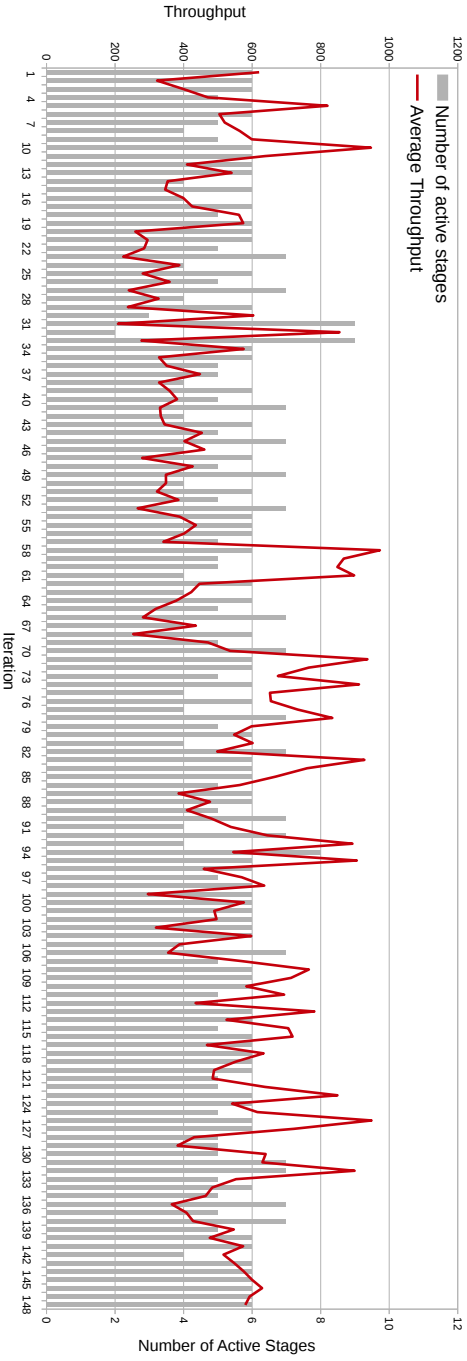


Figure A.1. The Results of the Feasibility Test on the INTEL

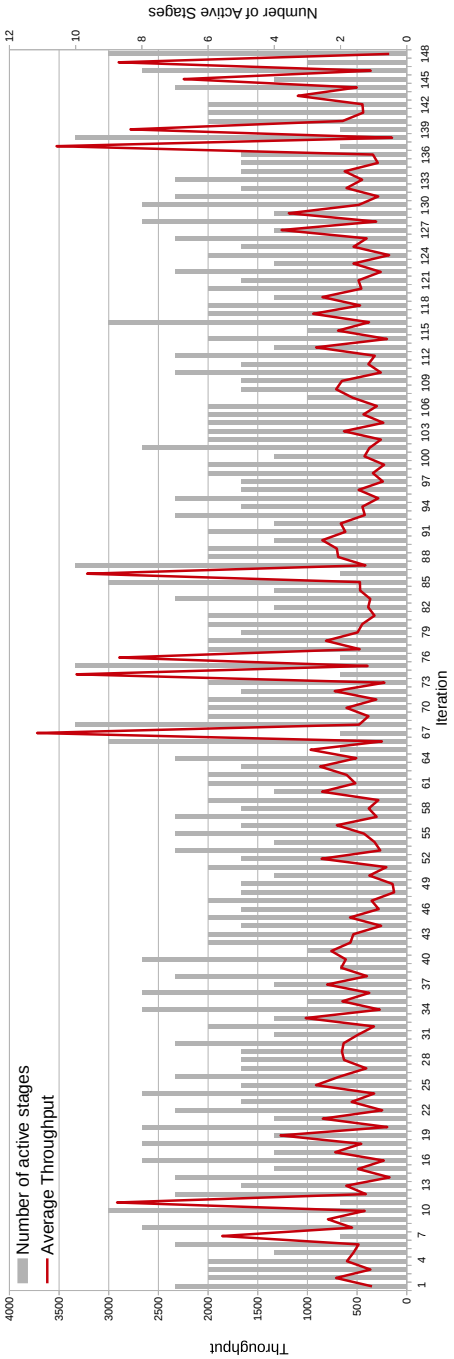


Figure A.2. The Results of the Feasibility Test on the AMD

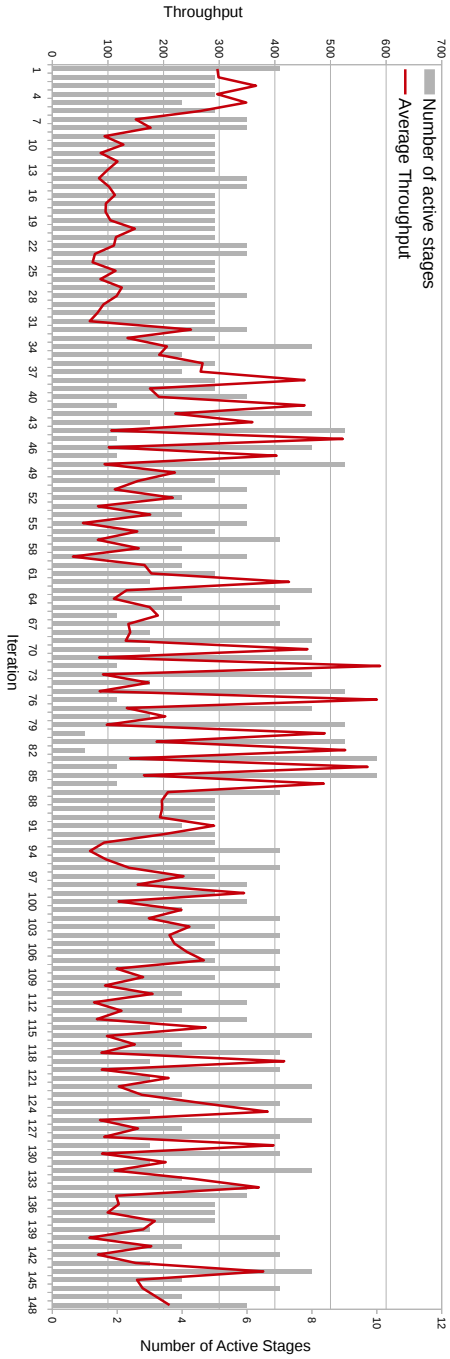


Figure A.3. The Results of the Feasibility Test on the SUN

A.2 Pipe Comparison Data

AMD

	Unsynchronized Pipe	Unsynchronized Pipe with Sensor Operations
Throughput	440.249	356.206
Error	4.504	5.792
in Operations/Microsecond		
Normalized	100	80.9101213177

INTEL

	Unsynchronized Pipe	Unsynchronized Pipe with Sensor Operations
Throughput	380.739	386.176
Error	5.974	1.441
in Operations/Microsecond		
Normalized	100	101.4280123654

SPARC

	Unsynchronized Pipe	Unsynchronized Pipe with Sensor Operations
Throughput	21439	20213
Error	0.001	0.003
in Operations/Microsecond		
Normalized	100	94.2814496945

Figure A.4. Comparison of the pipe implementations on different systems. All data is measured in operations per microsecond.

A.3 Performance Evaluation Data

A.3.1 INTEL Data

In these data all measured values are given in milliseconds. They represent the total execution time of a single run in a given scenario. This scenario is identified through the number of stages at the top of each entry. The error is computed as the standard deviation of the values.

Low Computational Efforts

Stages		5	10	15	20	25	30
Algorithm	Default						
Run1		46	88	96	127	158	287
Run2		45	73	96	128	158	259
Run3		45	64	95	127	158	195
Run4		61	64	95	127	157	190
Run5		63	63	95	127	158	190
Run6		49	63	95	126	158	249
Run7		46	64	95	126	158	287
Run8		46	64	95	126	158	263
Run9		46	63	95	126	158	198
Run10		45	64	95	126	158	190
Avg		49.2	67	95.2	126.6	157.9	230.8
Error		6.86	7.96	0.42	0.70	0.32	41.94
Algorithm	Simple						
Run1		201	402	403	403	404	606
Run2		201	202	403	403	403	404
Run3		201	402	403	403	403	605
Run4		201	402	402	403	403	404
Run5		201	402	403	211	404	605
Run6		201	402	403	201	403	605
Run7		201	402	402	403	403	805
Run8		201	403	403	403	403	403
Run9		201	402	404	402	403	403
Run10		201	402	403	403	403	603
Avg		201	382.1	402.9	363.5	403.2	544.3
Error		0.00	63.28	0.57	83.04	0.42	135.63
Algorithm	FDP1						
Run1		402	403	403	607	866	719
Run2		211	605	606	405	403	404
Run3		202	402	403	405	807	829
Run4		201	403	650	767	807	674
Run5		201	402	404	611	808	607
Run6		201	202	405	606	819	824
Run7		202	402	403	404	813	973
Run8		201	202	404	807	606	608
Run9		202	403	403	407	619	842
Run10		202	402	403	404	403	403
Avg		222.5	382.6	448.4	542.3	695.1	688.3
Error		63.14	114.24	95.23	159.09	176.58	188.36
Algorithm	FDP2						
Run1		201	403	403	404	824	403
Run2		201	201	403	686	818	1132

Run3	403	604	403	703	403	791
Run4	202	402	614	402	1385	1069
Run5	201	607	404	605	1008	607
Run6	201	402	403	402	1058	405
Run7	201	402	405	404	1494	840
Run8	204	201	404	807	404	1009
Run9	209	403	403	605	807	1044
Run10	202	404	612	403	699	845
Avg	222.5	402.9	445.4	542.1	890	814.5
Error	63.47	134.84	88.34	156.85	361.83	266.31

Algorithm	StableTD					
Run1	201	476	956	924	1290	2163
Run2	201	484	1021	856	1237	2167
Run3	206	507	1006	895	1249	2143
Run4	401	603	1019	833	1148	2151
Run5	401	522	1005	844	1346	2087
Run6	401	478	1004	886	1240	2131
Run7	202	603	1008	838	1194	2139
Run8	207	504	1004	873	1132	2162
Run9	201	669	995	1006	1179	2106
Run10	202	647	1003	845	1155	2084
Avg	262.3	549.3	1002.1	880	1217	2133.3
Error	95.73	73.75	17.90	52.91	68.09	30.92

Algorithm	StableBU					
Run1	201	202	202	202	403	604
Run2	201	201	202	402	603	403
Run3	201	201	201	402	203	604
Run4	201	201	202	403	202	403
Run5	201	201	202	402	402	603
Run6	201	201	202	403	603	603
Run7	201	201	202	402	201	603
Run8	202	202	202	202	402	403
Run9	202	201	201	202	603	603
Run10	201	201	201	402	212	402
Avg	201.2	201.2	201.7	342.2	383.4	523.1
Error	0.42	0.42	0.48	96.75	174.43	103.58

Mixed Computational Efforts

Stages		5	10	15	20	25	30
Algorithm	Default						
Run1		260	520	895	1154	1529	1957
Run2		260	685	894	1154	1529	1950
Run3		261	682	894	1153	1529	1791
Run4		261	520	895	1153	1530	1789
Run5		261	521	894	1154	1530	1790
Run6		261	521	894	1153	1530	1791
Run7		261	521	894	1153	1530	1790
Run8		261	520	894	1153	1529	1790
Run9		261	520	894	1154	1529	1790
Run10		261	521	894	1154	1530	1790
Avg		260.8	553.1	894.2	1153.5	1529.5	1822.8
Error		0.42	68.73	0.42	0.53	0.53	68.91
Algorithm	Simple						
Run1		603	1233	3386	5756	5522	9593
Run2		803	1585	4658	5088	5328	9422
Run3		602	1487	5787	5303	6014	8507
Run4		803	1463	2649	4529	6087	9855
Run5		613	1867	2356	5142	6413	8292
Run6		602	1745	5657	4498	6444	8760
Run7		602	1604	4680	3613	6395	8883
Run8		602	1713	3048	5569	6470	9545
Run9		607	1846	4216	5067	5655	10216
Run10		806	803	3993	4615	6755	8627
Avg		664.3	1534.6	4043	4918	6108.3	9170
Error		96.47	320.50	1186.68	623.44	471.90	641.37
Algorithm	FDP1						
Run1		1015	1847	5172	3055	7736	10886
Run2		1004	1207	6606	2896	5478	10482
Run3		1005	1978	2409	4806	8562	7763
Run4		1005	1207	1623	2403	4220	8682
Run5		690	1406	3559	4481	8001	2848
Run6		835	1207	3662	3806	2613	11477
Run7		1004	1694	2755	9617	8289	2675
Run8		804	1469	2010	7761	7399	3044
Run9		1004	1407	2617	4769	7283	6930
Run10		1205	1205	2135	9659	2610	3209
Avg		957.1	1462.7	3254.8	5325.3	6219.1	6799.6
Error		143.64	285.90	1563.57	2714.74	2317.78	3593.97
Algorithm	FDP2						
Run1		1205	1247	2215	5146	2818	2815
Run2		604	1207	2566	4529	7424	3223

Run3	867	1475	2093	2998	6446	9998
Run4	1005	1207	2800	3073	10967	7738
Run5	604	1205	2012	4223	3013	10370
Run6	805	1407	4163	6874	7669	10777
Run7	1204	1407	3754	4851	3138	10756
Run8	638	1241	3942	6056	11659	2811
Run9	602	1607	3279	5296	5966	10486
Run10	603	1408	2426	5770	9400	10010
Avg	813.7	1341.1	2925	4881.6	6850	7898.4
Error	247.88	139.51	802.97	1236.38	3219.42	3523.44

Algorithm	StableTD					
Run1	1002	2622	7142	6786	11774	17607
Run2	907	2316	7309	7459	10142	12660
Run3	1118	2743	7595	8562	10246	12060
Run4	1139	2807	7038	10512	9670	12341
Run5	1100	2855	7043	11211	10391	13015
Run6	1192	3044	7256	11611	10761	13021
Run7	1302	2879	7302	11417	11088	15828
Run8	1120	2814	7183	10347	9773	15835
Run9	1024	2849	7430	11241	11904	16614
Run10	1342	2756	7062	11753	10691	15872
Avg	1124.6	2768.5	7236	10089.9	10644	14485.3
Error	131.93	192.25	181.04	1820.83	764.44	2053.23

Algorithm	StableBU					
Run1	429	649	3254	3815	11914	6390
Run2	404	1863	4023	5212	11606	12130
Run3	1203	1475	2210	5209	9860	11904
Run4	402	1833	2408	2496	10243	10075
Run5	1003	2715	3785	1603	12604	7409
Run6	601	1645	2004	4622	12062	11280
Run7	602	3450	2405	3892	7383	11806
Run8	412	3006	3206	3475	9077	9979
Run9	802	1509	3475	4867	7087	11162
Run10	602	3915	1403	5180	11348	11426
Avg	646	2206	2817.3	4037.1	10318.4	10356.1
Error	276.46	1021.07	851.96	1237.24	1953.44	1970.03

High Computational Efforts

Stages		5	10	15	20	25	30
Algorithm	Default						
Run1		604	1213	1817	2426	3028	3797
Run2		603	1213	1818	2426	3027	3793
Run3		604	1213	1818	2587	3027	3636
Run4		604	1213	1819	2595	3027	3647
Run5		603	1213	1979	2426	3028	3647
Run6		603	1213	1978	2425	3028	3636
Run7		604	1213	1818	2426	3257	3797
Run8		604	1214	1818	2427	3027	3796
Run9		604	1212	1817	2425	3028	3636
Run10		603	1214	1817	2425	3026	3796
Avg		603.6	1213.1	1849.9	2458.8	3050.3	3718.1
Error		0.52	0.57	67.78	69.70	72.63	82.01
Algorithm	Simple						
Run1		1943	8825	7990	23402	17750	24327
Run2		1640	8259	8038	14939	19705	23991
Run3		2527	8438	10023	14357	18128	23734
Run4		2100	9576	7737	13747	19816	24904
Run5		2383	9202	9548	13565	18815	25265
Run6		2268	7052	9066	14990	18599	23842
Run7		2458	9376	8128	15105	18980	26101
Run8		2042	9462	9429	18306	18359	21833
Run9		1459	10516	8359	13312	19546	23981
Run10		2153	9422	9508	12611	18262	24932
Avg		2097.3	9012.8	8782.6	15433.4	18796	24291
Error		345.13	936.44	818.71	3201.72	708.57	1142.73
Algorithm	FDP1						
Run1		2466	5821	10421	13113	13881	29240
Run2		1204	5909	10828	11322	4218	4422
Run3		1406	5875	13542	14004	17509	26365
Run4		3084	5780	15890	13593	14873	4614
Run5		1205	4019	15346	18032	19216	4529
Run6		2230	5131	14175	4737	19614	23798
Run7		1205	5472	13712	21703	17469	5087
Run8		2217	5289	16015	17672	20970	20364
Run9		1616	3759	16138	22930	18654	4857
Run10		1606	4750	14129	22136	12003	26476
Avg		1823.9	5180.5	14019.6	15924.2	15840.7	14975.2
Error		644.23	777.24	2033.66	5695.20	4942.80	11057.24
Algorithm	FDP2						
Run1		1407	9319	8001	10472	19388	23640
Run2		2010	10402	8051	12989	4410	16002

Run3	1652	5320	8070	13048	17044	4721
Run4	1607	3362	8941	3610	19429	22133
Run5	1406	5929	14763	12102	15333	5268
Run6	1606	3889	13602	12934	21911	26305
Run7	3693	4006	14504	13529	17576	23974
Run8	4403	4942	13891	13635	4409	26112
Run9	2934	3384	17223	14622	20343	24714
Run10	1429	2560	15895	14531	20122	20715
Avg	2214.7	5311.3	12294.1	12147.2	15996.5	19358.4
Error	1080.82	2609.93	3621.25	3227.68	6383.37	8136.21

Algorithm	StableTD					
Run1	3284	6771	18395	15480	30080	28352
Run2	3477	7366	18686	17999	30602	26980
Run3	3496	6393	19362	16135	29403	27771
Run4	3250	7401	18337	14631	31330	29658
Run5	2479	7249	18668	15826	30604	26080
Run6	2865	6849	19037	15201	30308	28771
Run7	3342	6811	19387	15189	30073	28493
Run8	3116	7225	19134	15372	30705	29480
Run9	2759	7297	18496	16345	29994	30903
Run10	2768	7293	19303	16114	30827	28238
Avg	3083.6	7065.5	18880.5	15829.2	30392.6	28472.6
Error	346.22	336.31	410.63	926.15	537.36	1371.56

Algorithm	StableBU					
Run1	2613	3833	7383	18806	12432	17146
Run2	2804	3138	7948	16895	15284	13533
Run3	1402	3809	7778	14829	16793	16490
Run4	2003	4468	7798	17120	14084	16273
Run5	1713	4529	6647	18517	8823	13063
Run6	2405	4880	6873	9951	14799	18269
Run7	3405	4305	3934	19372	14169	16250
Run8	4127	5206	6298	15823	13827	18079
Run9	4102	5206	5143	14454	15015	15288
Run10	2337	4896	6280	11952	16859	14862
Avg	2691.1	4427	6608.2	15771.9	14208.5	15925.3
Error	935.72	671.82	1280.33	3052.94	2313.79	1751.80

A.3.2 AMD Data

Low Computational Efforts

Stages		5	10	15	20	25	30
Algorithm	Default						
Run1		33	62	86	113	154	184
Run2		29	63	87	128	144	171
Run3		32	58	89	111	137	168
Run4		29	59	86	111	140	164
Run5		31	57	85	112	139	169
Run6		29	57	99	112	139	164
Run7		32	57	85	113	136	175
Run8		29	67	82	113	139	168
Run9		33	84	87	112	159	169
Run10		36	66	90	113	140	169
Avg		31.3	63	87.6	113.8	142.7	170.1
Error		2.36	8.27	4.58	5.05	7.66	5.82
Algorithm	Simple						
Run1		201	202	411	403	407	408
Run2		202	202	405	405	408	404
Run3		202	202	406	405	404	407
Run4		202	218	406	407	407	405
Run5		202	404	203	406	406	404
Run6		201	404	405	403	415	405
Run7		201	202	406	203	406	408
Run8		201	404	403	203	406	407
Run9		201	202	404	405	407	407
Run10		201	203	404	406	207	404
Avg		201.4	264.3	385.3	364.6	387.3	405.9
Error		0.52	96.53	64.09	85.18	63.42	1.66
Algorithm	FDP1						
Run1		202	203	409	411	201	684
Run2		202	406	409	405	409	660
Run3		202	214	416	404	410	829
Run4		202	202	203	410	642	823
Run5		203	203	208	749	669	841
Run6		202	203	408	409	201	412
Run7		202	202	204	404	641	813
Run8		203	202	405	407	406	814
Run9		202	411	203	408	409	816
Run10		202	204	204	203	651	411
Avg		202.2	245	306.9	421	463.9	710.3
Error		0.42	86.25	108.09	131.94	179.71	169.36
Algorithm	FDP2						
Run1		203	202	409	411	630	682
Run2		210	409	408	410	697	415

Run3	203	203	403	410	1068	412
Run4	203	214	204	409	878	1046
Run5	202	202	413	408	815	882
Run6	202	203	409	409	410	410
Run7	201	203	202	204	661	822
Run8	202	203	408	654	658	637
Run9	202	203	407	404	408	823
Run10	202	404	407	408	635	409
Avg	203	244.6	367	412.7	686	653.8
Error	2.54	85.41	86.47	106.43	200.04	235.41

Algorithm	StableTD					
Run1	402	1088	1646	1741	2235	2873
Run2	401	1132	1292	1750	2250	2932
Run3	401	1057	1254	1812	2210	2788
Run4	403	1157	1454	1851	2526	2877
Run5	403	1019	1396	1716	1917	2508
Run6	407	1017	1269	1681	2132	2618
Run7	402	1017	1237	1791	2354	2621
Run8	602	967	1351	1718	2390	2599
Run9	403	1020	1060	1749	2226	2494
Run10	402	1043	1443	1849	2390	2618
Avg	422.6	1051.7	1340.2	1765.8	2263	2692.8
Error	63.06	58.32	157.85	57.79	167.24	160.38

Algorithm	StableBU					
Run1	202	202	203	217	204	204
Run2	201	202	204	203	203	203
Run3	202	202	203	203	203	203
Run4	202	202	203	203	203	204
Run5	202	202	203	202	203	203
Run6	202	202	203	203	203	203
Run7	201	202	203	203	203	202
Run8	201	202	203	203	202	203
Run9	213	201	202	212	203	203
Run10	201	202	202	203	202	201
Avg	202.7	201.9	202.9	205.2	202.9	202.9
Error	3.65	0.32	0.57	5.05	0.57	0.88

Mixed Computational Efforts

Stages		5	10	15	20	25	30
Algorithm	Default						
Run1		223	448	759	973	1300	1504
Run2		219	448	756	985	1277	1486
Run3		222	449	761	970	1277	1485
Run4		218	448	756	970	1275	1484
Run5		222	442	749	964	1280	1482
Run6		221	445	762	961	1274	1486
Run7		222	447	745	972	1282	1511
Run8		220	443	751	969	1274	1516
Run9		219	464	747	968	1290	1516
Run10		224	440	749	961	1276	1515
Avg		221	447.4	753.5	969.3	1280.5	1498.5
Error		1.94	6.57	6.08	6.96	8.38	15.09
Algorithm	Simple						
Run1		403	1011	3640	6013	10208	11889
Run2		403	837	2197	5400	9382	10261
Run3		403	1009	3280	6145	8786	9199
Run4		417	1096	4667	8343	9385	10156
Run5		403	604	5997	3210	9561	11589
Run6		403	1211	3928	6511	7516	9384
Run7		403	1009	1379	5976	7671	9373
Run8		402	1016	2537	6896	8681	11043
Run9		402	808	3400	6722	7236	13138
Run10		401	1211	2863	6018	7923	10160
Avg		404	981.2	3388.8	6123.4	8634.9	10619.2
Error		4.62	187.23	1302.31	1294.81	1006.80	1279.16
Algorithm	FDP1						
Run1		403	1237	1234	2942	1843	2035
Run2		606	1013	1461	1810	1829	2036
Run3		808	1013	1407	1422	1826	2035
Run4		698	1015	1215	1215	1829	2027
Run5		613	1212	1214	1624	1831	2034
Run6		837	1216	1214	5436	1624	2053
Run7		805	1209	1214	1621	1829	2028
Run8		811	1211	1215	1423	1825	2026
Run9		604	1212	1214	1429	1850	1824
Run10		772	1410	1416	1414	1831	1822
Avg		695.7	1174.8	1280.4	2033.6	1811.7	1992
Error		138.49	126.50	102.94	1289.10	66.42	89.39
Algorithm	FDP2						
Run1		698	1418	1237	1641	1868	2044
Run2		819	903	1432	1445	1834	1833

Run3	1008	1215	1484	1624	1630	2035
Run4	827	1011	1281	4403	1847	1839
Run5	403	1034	1288	1624	1832	2027
Run6	1009	1457	1418	1435	1828	2039
Run7	695	1212	1214	1420	2031	2027
Run8	404	1199	1215	1417	1849	2027
Run9	1010	1410	1214	2019	1822	2028
Run10	820	807	1414	1423	1626	2021
Avg	769.3	1166.6	1319.7	1845.1	1816.7	1992
Error	225.82	224.31	105.84	917.90	116.63	82.51

Algorithm	StableTD					
Run1	1797	4663	8973	8417	14523	12185
Run2	1607	8711	11097	7548	9640	14074
Run3	2764	4411	8491	10143	9529	10510
Run4	1773	6105	6710	12815	11258	10738
Run5	1005	5053	7949	8696	12514	9643
Run6	1470	4991	13110	7428	8994	12782
Run7	1084	7293	8286	9876	10411	10222
Run8	1407	7755	9655	7631	11618	14216
Run9	1466	4164	11558	9898	21053	10410
Run10	1408	4601	12289	10558	10976	13072
Avg	1578.1	5774.7	9811.8	9301	12051.6	11785.2
Error	488.55	1604.17	2097.82	1694.49	3551.86	1686.66

Algorithm	StableBU					
Run1	403	1662	1336	5726	4249	13339
Run2	403	1609	4444	2617	6420	5495
Run3	403	604	9870	8704	16333	17249
Run4	403	805	6325	7260	11123	2742
Run5	402	604	4329	8173	4258	10665
Run6	402	1408	6661	2411	5503	15249
Run7	402	603	5561	8854	4364	6474
Run8	403	603	4931	3411	15325	13946
Run9	402	805	7380	3969	7694	10418
Run10	403	1289	2209	7695	8435	19211
Avg	402.6	999.2	5304.6	5882	8370.4	11478.8
Error	0.52	442.72	2477.94	2576.15	4494.23	5331.43

High Computational Efforts

Stages		5	10	15	20	25	30
Algorithm	Default						
Run1		509	1018	1515	2035	2551	3046
Run2		509	1014	1527	2064	2523	3025
Run3		509	1023	1519	2025	2520	3025
Run4		509	1021	1518	2016	2524	3027
Run5		508	1015	1516	2017	2518	3024
Run6		508	1013	1544	2018	2525	3017
Run7		507	1013	1517	2023	2519	3036
Run8		508	1014	1516	2021	2526	3024
Run9		515	1034	1516	2019	2539	3021
Run10		508	1011	1518	2024	2522	3025
Avg		509	1017.6	1520.6	2026.2	2526.7	3027
Error		2.21	6.90	8.90	14.37	10.37	8.22
Algorithm	Simple						
Run1		1513	12854	24921	31076	36462	52282
Run2		1624	12605	18951	26411	41020	57301
Run3		1242	11953	21829	30254	40767	49099
Run4		1993	10040	19856	34873	43571	50834
Run5		2017	11459	18488	31910	43252	54086
Run6		1510	10733	20175	27685	41918	50572
Run7		1509	12349	17896	30261	43565	53182
Run8		907	14338	23732	29110	42880	49090
Run9		1212	12827	22992	26586	41573	49508
Run10		1319	15121	21016	31627	41984	49187
Avg		1484.6	12427.9	20985.6	29979.3	41699.2	51514.1
Error		342.80	1528.21	2348.15	2624.99	2100.10	2707.47
Algorithm	FDP1						
Run1		1209	1816	2047	2439	3042	3244
Run2		4256	1816	2024	2430	3044	26561
Run3		1010	4377	2021	2646	3233	3615
Run4		1209	9525	1823	2434	3059	3239
Run5		1007	2016	2041	2837	3035	3415
Run6		1211	1815	2021	2993	3037	3616
Run7		1009	2018	2024	2423	3036	3451
Run8		1006	2016	2024	2627	2828	3615
Run9		1206	1816	2019	2424	3028	3414
Run10		2267	1815	2059	2430	3164	3422
Avg		1539	2903	2010.3	2568.3	3050.6	5759.2
Error		1026.11	2456.12	67.17	204.67	103.83	7310.32
Algorithm	FDP2						
Run1		1009	4989	2018	2676	3222	3440
Run2		1010	4114	2022	2487	3016	3641

Run3	1220	1613	2032	2844	3020	44686
Run4	2202	2016	2223	2643	3219	3438
Run5	3332	8828	2236	29962	3017	3417
Run6	2301	3941	2030	2614	3017	3256
Run7	1008	6897	2023	2620	3018	3422
Run8	1005	2016	2023	3818	36520	3626
Run9	1210	9514	2227	2613	3215	3214
Run10	1411	1812	2021	2631	2817	3432
Avg	1570.8	4574	2085.5	5490.8	6408.1	7557.2
Error	787.34	2947.01	98.93	8606.70	10581.00	13046.41

Algorithm	StableTD					
Run1	10171	18834	29753	40577	48467	100310
Run2	6629	19080	30748	38157	53053	95052
Run3	4391	19088	29189	38237	53529	86114
Run4	5570	18670	31671	35300	49935	59458
Run5	9812	20537	28553	36040	51558	57164
Run6	4453	20220	29952	37095	50200	55273
Run7	5452	20232	29057	38332	52776	59121
Run8	5508	19051	29357	39163	49745	60875
Run9	4761	19765	29691	38809	48722	58203
Run10	4083	18752	29308	31678	54154	56461
Avg	6083	19422.9	29727.9	37338.8	51213.9	68803.1
Error	2190.25	697.62	900.82	2503.01	2068.84	17665.12

Algorithm	StableBU					
Run1	1205	10049	15434	24574	25178	26918
Run2	967	7545	12623	27743	31724	18073
Run3	1205	10796	10548	22439	32350	28015
Run4	1115	3877	19214	28033	24290	28448
Run5	987	3411	7461	21182	25381	29503
Run6	970	6756	16657	26175	22823	31678
Run7	803	3410	10095	24410	32677	28083
Run8	1203	7902	15243	23207	17160	27954
Run9	1206	6013	15864	17979	26142	30457
Run10	1004	6248	19720	11991	33329	28170
Avg	1066.5	6600.7	14285.9	22773.3	27105.4	27729.9
Error	140.51	2589.37	4014.45	4852.61	5285.26	3667.41

A.3.3 SUN Data

Low Computational Efforts

Stages		5	10	15	20	25	30
Algorithm	Default						
Run1		209	433	668	915	1435	1437
Run2		209	431	660	921	1426	1450
Run3		207	436	661	907	1422	1423
Run4		214	432	662	915	1431	1428
Run5		209	432	666	912	1442	1428
Run6		210	438	662	910	1432	1447
Run7		209	432	660	916	1435	1434
Run8		215	434	666	910	1434	1431
Run9		209	431	662	912	1431	1430
Run10		209	436	659	912	1429	1421
Avg		210	433.5	662.6	913	1431.7	1432.9
Error		2.49	2.42	3.03	3.92	5.46	9.48
Algorithm	Simple						
Run1		408	611	862	1490	1618	2368
Run2		408	609	1021	1415	2968	1793
Run3		410	813	806	1614	3053	4050
Run4		408	610	807	1284	2396	3313
Run5		408	662	806	1485	2295	1944
Run6		407	605	806	1314	2337	2205
Run7		408	603	1009	1208	2349	1611
Run8		461	606	806	1410	2477	1811
Run9		402	604	1414	1611	2241	3656
Run10		403	603	1007	1612	2252	2512
Avg		412.3	632.6	934.4	1444.3	2398.6	2526.3
Error		17.29	65.81	193.15	144.91	398.78	854.50
Algorithm	FDP1						
Run1		416	617	1218	1644	2033	2034
Run2		409	833	1549	2018	2064	2262
Run3		412	650	1225	1494	2032	2447
Run4		410	665	1213	2187	2231	3140
Run5		410	844	1085	1529	1935	2637
Run6		410	606	1617	1943	1827	1842
Run7		462	606	1445	2119	2442	1835
Run8		405	607	1211	2080	1872	2181
Run9		405	807	1819	1961	1824	2688
Run10		404	605	1210	1420	1842	2278
Avg		414.3	684	1359.2	1839.5	2010.2	2334.4
Error		17.15	101.80	235.70	287.52	200.10	407.20
Algorithm	FDP2						
Run1		415	631	1418	1428	2062	2430
Run2		410	634	1215	1983	2915	2028

Run3	412	1012	1217	1769	2512	3965
Run4	410	808	1276	1678	2250	3233
Run5	410	651	1820	2103	2254	1837
Run6	409	645	1418	1619	2228	3083
Run7	461	605	1515	2053	2031	2620
Run8	405	608	1210	1448	1675	2229
Run9	404	707	1211	2031	2669	2767
Run10	403	1009	1211	1429	2377	2056
Avg	413.9	731	1351.1	1754.1	2297.3	2624.8
Error	16.96	158.80	199.16	272.87	348.75	657.54

Algorithm	StableTD					
Run1	1069	1319	2690	2399	3415	4093
Run2	805	1428	3749	5398	3939	4888
Run3	830	1206	3638	2326	3419	3960
Run4	841	2323	3625	5335	3461	4227
Run5	873	2169	3498	3011	3486	3964
Run6	805	1339	3792	4583	3257	3880
Run7	803	2251	3100	3577	3376	4491
Run8	825	2334	2686	3818	3812	4144
Run9	610	1651	2874	5318	3347	3989
Run10	838	1407	4670	3780	3876	4095
Avg	829.9	1742.7	3432.2	3954.5	3538.8	4173.1
Error	110.33	468.71	612.41	1173.65	242.61	305.02

Algorithm	StableBU					
Run1	409	410	811	2251	1811	1811
Run2	209	609	804	1261	1681	1812
Run3	411	409	818	1411	1609	2616
Run4	209	408	806	1005	2413	1971
Run5	209	409	603	1207	1408	2013
Run6	408	410	804	1408	1615	2221
Run7	458	615	805	1409	1812	2211
Run8	402	604	603	1016	1608	2613
Run9	202	403	804	1207	1609	2610
Run10	403	401	804	1205	1206	2901
Avg	332	467.8	766.2	1338	1677.2	2277.9
Error	108.55	97.75	86.13	352.98	314.46	384.75

Mixed Computational Efforts

Stages		5	10	15	20	25	30
Algorithm	Default						
Run1		1712	3449	5886	12599	9027	11879
Run2		1710	3449	5891	12588	9035	11873
Run3		1710	3454	5879	12600	9021	11854
Run4		1716	3449	5883	12598	9027	11867
Run5		1711	3449	5885	12592	9036	11867
Run6		1711	3455	5881	12584	9023	11867
Run7		1711	3450	5878	12596	9026	11867
Run8		1716	3452	5883	12590	9031	11869
Run9		1711	3449	5878	12596	9024	11870
Run10		1710	3454	5886	12590	9021	11866
Avg		1711.8	3451	5883	12593.3	9027.1	11867.9
Error		2.30	2.49	4.16	5.29	5.36	6.28
Algorithm	Simple						
Run1		2557	14621	22705	26210	33809	38986
Run2		4476	10790	21612	27702	25843	35017
Run3		3950	10459	21776	20495	40317	69902
Run4		3333	10390	15932	30122	36227	39830
Run5		3790	15317	24417	25348	26364	36985
Run6		3566	11225	17948	24150	31497	55749
Run7		1979	6927	25141	18308	31440	63657
Run8		3280	10933	21543	23474	32740	49933
Run9		3540	5958	26847	26767	31659	39954
Run10		4591	6109	19750	24720	50452	38940
Avg		3506.2	10272.9	21767.1	24729.6	34034.8	46895.3
Error		796.80	3217.29	3297.59	3427.33	7151.44	12297.83
Algorithm	FDP1						
Run1		2448	6976	16257	22485	25206	29184
Run2		2120	5155	17118	22763	52552	33338
Run3		2009	8164	23769	31807	30420	30019
Run4		3330	7167	37213	38545	28828	40016
Run5		2508	5649	22427	20667	29738	30966
Run6		2075	10790	20155	25119	31279	32093
Run7		2315	3341	15076	26241	29742	52492
Run8		2461	10128	18278	20480	38972	29570
Run9		3421	11887	23103	25539	41988	30682
Run10		2273	8312	22862	34602	26267	54734
Avg		2496	7756.9	21625.8	26824.8	33499.2	36309.4
Error		493.64	2667.18	6300.98	6156.13	8502.26	9647.50
Algorithm	FDP2						
Run1		2347	12719	19798	21257	60783	27429
Run2		4243	5549	18093	30808	29635	52679

Run3	2287	11846	31407	21343	26821	47949
Run4	3417	7319	27933	15303	42982	33179
Run5	2010	11369	18189	13505	30681	42454
Run6	2181	10690	19401	16359	38956	42304
Run7	2210	4630	16118	21755	43433	42280
Run8	2254	12496	19830	24633	39275	29074
Run9	3479	3409	19516	30127	55795	59868
Run10	2933	6342	13286	24059	50367	40794
Avg	2736.1	8636.9	20357.1	21914.9	41872.8	41801
Error	747.05	3550.57	5369.71	5817.68	11274.71	10150.74

Algorithm	StableTD					
Run1	5874	11835	28299	21087	55130	41892
Run2	6031	12580	15299	48518	27175	52941
Run3	6258	20820	25714	26807	42476	51314
Run4	3886	20935	23976	22189	77711	104485
Run5	5237	6071	26879	27348	46497	91919
Run6	6503	5979	13043	17260	61990	125867
Run7	5914	21006	11618	25345	53366	38181
Run8	4545	16464	20291	24131	72139	33897
Run9	2173	6069	35500	22117	77524	111859
Run10	6121	11525	56143	35950	36566	60439
Avg	5254.2	13328.4	25676.2	27075.2	55057.4	71279.4
Error	1358.53	6219.92	13037.84	9012.46	17357.59	33934.76

Algorithm	StableBU					
Run1	3656	8328	12108	28008	34379	24171
Run2	3606	5611	17994	28324	18530	35901
Run3	3468	14967	18375	32167	31648	34573
Run4	2295	7339	30081	18070	34507	35723
Run5	4133	4211	19805	28402	46096	33630
Run6	1429	13863	17181	20775	52716	36144
Run7	3493	5232	13950	24473	24362	52725
Run8	1404	4571	18637	40357	48763	29063
Run9	3331	11935	15679	25818	36451	40217
Run10	3092	7158	9628	29979	20739	32270
Avg	2990.7	8321.5	17343.8	27637.3	34819.1	35441.7
Error	952.94	3917.22	5504.70	6154.79	11680.21	7484.15

High Computational Efforts

Stages		5	10	15	20	25	30
Algorithm	Default						
Run1		3969	7939	11915	15884	17669	21234
Run2		3969	7936	11913	15917	17663	21207
Run3		3969	7943	11907	15893	17651	21217
Run4		3973	7948	11908	15902	17670	21231
Run5		3968	7939	11916	15894	17672	21208
Run6		3968	7943	11908	15890	17649	21213
Run7		3970	7939	11908	15902	17658	21205
Run8		3973	7940	11912	15892	17658	21222
Run9		3968	7940	11917	15899	17664	21221
Run10		3968	7943	11911	15888	17657	21216
Avg		3969.5	7941	11911.5	15896.1	17661.1	21217.4
Error		1.96	3.33	3.69	9.40	7.87	9.81
Algorithm	Simple						
Run1		13087	25674	55787	82664	82323	90827
Run2		12123	26095	56739	68927	85894	104699
Run3		12016	24502	52076	52123	63051	93654
Run4		14534	21516	53913	127490	63187	78645
Run5		8634	29904	59318	90277	69018	78626
Run6		8129	24069	74535	62249	66605	97183
Run7		10461	24404	55819	75248	62619	80635
Run8		11323	23612	53511	64142	90218	86840
Run9		8230	29383	57317	64837	66832	85647
Run10		7619	27657	47583	59106	64885	81676
Avg		10615.6	25681.6	56659.8	74706.3	71463.2	87843.2
Error		2380.48	2642.85	7066.00	21713.16	10486.54	8688.46
Algorithm	FDP1						
Run1		11422	17817	47643	56380	79598	104298
Run2		11931	26335	54287	84622	81820	96824
Run3		10169	28500	51313	87302	93259	121510
Run4		11358	22862	61620	91645	87960	128411
Run5		8179	20241	72600	81430	98082	99182
Run6		14464	25599	55318	80067	61427	85437
Run7		12038	21790	53016	60224	69147	115173
Run8		13486	25025	53657	63982	98525	87928
Run9		12158	23017	44102	66774	69340	109036
Run10		10313	28388	46208	96624	76481	114817
Avg		11551.8	23957.4	53976.4	76905	81563.9	106261.6
Error		1759.00	3464.92	8254.57	14037.26	12810.92	14127.37
Algorithm	FDP2						
Run1		9837	26699	52708	71198	74629	80073
Run2		11070	30871	44896	66452	62465	85382

Run3	13239	26811	46041	77117	90411	85625
Run4	11845	27179	45384	104134	76006	81889
Run5	11067	32327	45044	90383	82965	80372
Run6	10418	29639	45638	40363	83129	94946
Run7	11776	24648	69435	65659	90659	87688
Run8	10319	26095	53310	99791	96733	87187
Run9	11720	21942	43995	69048	60795	83329
Run10	10406	31112	65214	92909	112690	82074
Avg	11169.7	27732.3	51166.5	77705.4	83048.2	84856.5
Error	1004.27	3231.18	9168.47	19327.49	15696.45	4447.07

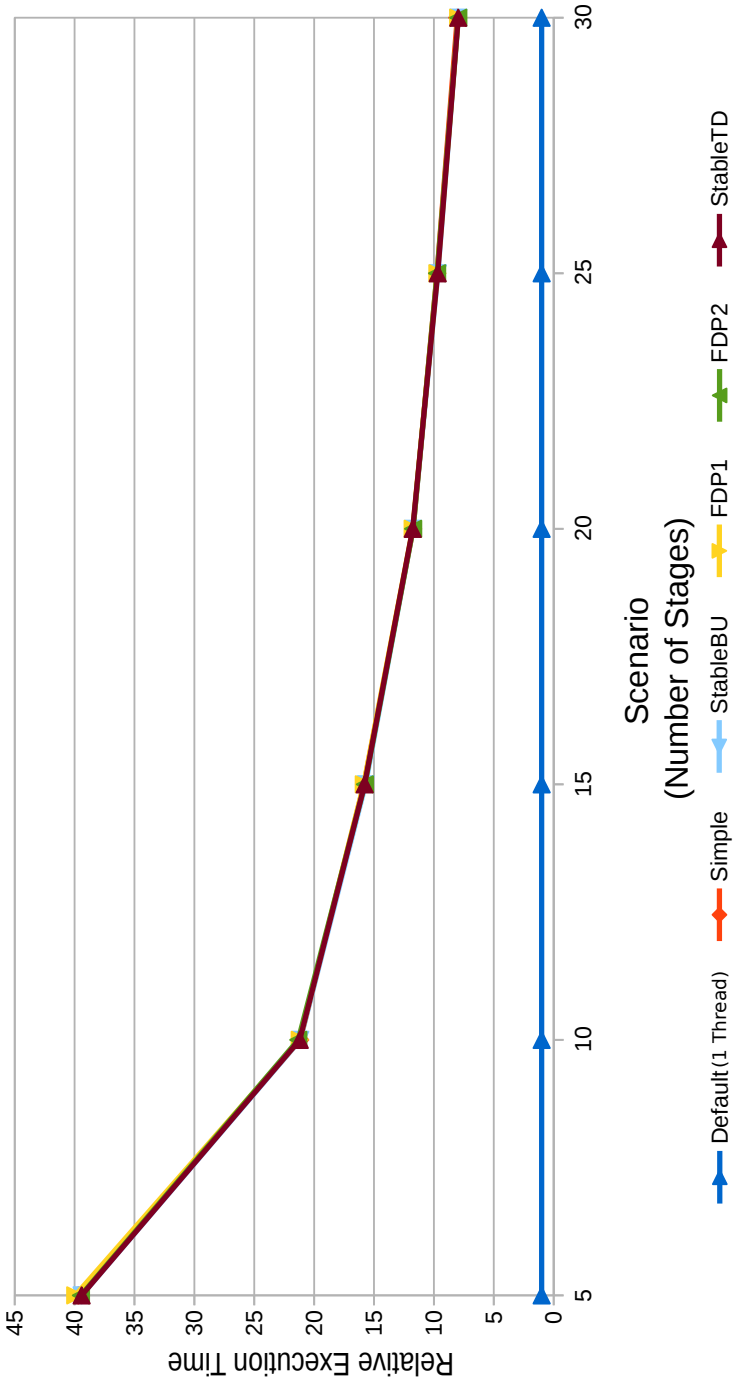
Algorithm	StableTD					
Run1	8515	22381	67575	132612	70563	88959
Run2	20260	39179	46700	63029	64147	79188
Run3	16398	20919	46287	110655	70417	92681
Run4	7514	39479	74703	130402	84282	83339
Run5	8997	24766	32868	84380	70047	97451
Run6	21047	21557	30902	97879	74401	77922
Run7	12635	17945	79711	76756	65347	84169
Run8	9720	22189	86642	111304	87617	81262
Run9	16390	26687	41513	92602	64237	186915
Run10	20436	29859	48977	65965	125775	79566
Avg	14191.2	26496.1	55587.8	96558.4	77683.3	95145.2
Error	5361.71	7511.88	19967.35	24674.37	18689.97	32859.88

Algorithm	StableBU					
Run1	12397	21836	40513	87311	73190	175075
Run2	16381	26367	45327	68062	83065	88718
Run3	12684	24847	59144	79715	58103	93612
Run4	7479	23410	65150	62077	108553	61839
Run5	11521	21247	58461	43429	83244	99835
Run6	10984	24969	57691	94359	64588	117065
Run7	10331	23733	38432	94691	52759	88972
Run8	10086	25292	34615	105902	57538	116115
Run9	13429	29685	70714	66268	76614	124410
Run10	11985	29627	58394	86637	73644	95348
Avg	11727.7	25101.3	52844.1	78845.1	73129.8	106098.9
Error	2339.74	2857.03	12224.17	18758.46	16414.96	30145.36

A. Appendices

A.3.4 Second Evaluation on the INTEL

Diagram of the Low Computational Effort Execution



Low Computational Efforts

Stages		5	10	15	20	25	30
Algorithm	Default						
Run1		5	9	13	17	21	25
Run2		5	9	13	17	20	25
Run3		5	9	12	17	21	26
Run4		5	9	13	17	21	24
Run5		5	9	13	17	21	25
Run6		5	13	13	17	22	26
Run7		6	9	13	18	22	26
Run8		5	9	13	17	21	25
Run9		5	10	12	18	20	27
Run10		5	9	13	17	20	25
Avg		5.1	9.5	12.8	17.2	20.9	25.4
Error		0.32	1.27	0.42	0.42	0.74	0.84
Algorithm	Simple						
Run1		201	201	202	201	202	202
Run2		201	201	202	202	202	202
Run3		211	201	202	202	203	202
Run4		201	201	201	201	214	203
Run5		201	202	201	202	203	202
Run6		201	201	201	202	202	202
Run7		201	201	212	202	203	202
Run8		201	201	202	202	202	202
Run9		205	201	202	202	202	202
Run10		201	201	202	202	202	225
Avg		202.4	201.1	202.7	201.8	203.5	204.4
Error		3.27	0.32	3.30	0.42	3.72	7.24
Algorithm	FDP1						
Run1		201	201	202	203	202	203
Run2		202	201	202	202	202	203
Run3		202	201	201	203	210	203
Run4		222	202	202	203	202	202
Run5		202	201	201	202	202	202
Run6		202	202	202	202	202	202
Run7		201	201	202	204	201	203
Run8		201	202	202	202	202	202
Run9		202	202	202	202	202	202
Run10		202	201	213	201	202	202
Avg		203.7	201.4	202.9	202.4	202.7	202.4
Error		6.45	0.52	3.57	0.84	2.58	0.52
Algorithm	FDP2						
Run1		202	202	202	202	202	202
Run2		201	201	202	202	202	203

Run3	201	201	202	202	202	203
Run4	202	202	202	202	203	203
Run5	202	201	202	202	203	202
Run6	201	213	202	202	202	203
Run7	201	202	201	202	203	202
Run8	201	201	202	201	214	201
Run9	201	201	202	201	202	202
Run10	201	202	201	202	202	202
Avg	201.3	202.6	201.8	201.8	203.5	202.3
Error	0.48	3.69	0.42	0.42	3.72	0.67

Algorithm	StableTD					
Run1	201	201	203	202	203	204
Run2	201	202	202	202	203	202
Run3	201	202	201	207	202	202
Run4	201	201	202	202	203	203
Run5	201	202	201	202	202	202
Run6	201	202	203	202	202	202
Run7	201	201	203	202	202	202
Run8	201	201	202	201	202	202
Run9	201	201	202	202	202	202
Run10	201	201	201	202	202	202
Avg	201	201.4	202	202.4	202.3	202.3
Error	0.00	0.52	0.82	1.65	0.48	0.67

Algorithm	StableBU					
Run1	201	201	201	202	203	203
Run2	201	202	201	202	203	203
Run3	202	202	202	202	203	203
Run4	201	201	202	202	203	202
Run5	201	201	202	202	203	202
Run6	201	201	202	202	203	202
Run7	201	202	202	202	202	202
Run8	202	202	201	202	202	202
Run9	202	202	201	202	202	202
Run10	201	201	201	202	202	202
Avg	201.3	201.5	201.5	202	202.6	202.3
Error	0.48	0.53	0.53	0.00	0.52	0.48

Mixed Computational Efforts

Stages		5	10	15	20	25	30
Algorithm	Default						
Run1		232	463	807	1037	1381	1935
Run2		232	462	807	1037	1382	1773
Run3		232	463	807	1078	1382	1614
Run4		232	462	807	1191	1381	1612
Run5		232	462	807	1037	1381	1613
Run6		232	462	807	1038	1382	1612
Run7		232	463	807	1037	1381	1770
Run8		233	463	807	1039	1380	1772
Run9		232	463	806	1037	1382	1612
Run10		340	462	807	1037	1381	1774
Avg		242.9	462.5	806.9	1056.8	1381.3	1708.7
Error		34.12	0.53	0.32	48.86	0.67	112.33
Algorithm	Simple						
Run1		201	603	834	1144	1206	1410
Run2		401	603	1005	1037	1426	1049
Run3		402	618	1015	1017	1005	1207
Run4		201	603	607	1004	1206	1005
Run5		401	603	1005	1408	1426	1004
Run6		402	602	804	1005	1004	1412
Run7		402	613	1004	1005	1292	1205
Run8		201	602	1005	831	1206	1406
Run9		402	803	1005	1005	1450	1205
Run10		201	602	803	1222	1405	1433
Avg		321.4	625.2	908.7	1067.8	1262.6	1233.6
Error		103.62	62.72	139.93	156.64	167.50	174.43
Algorithm	FDP1						
Run1		403	604	606	606	607	608
Run2		402	607	611	611	614	616
Run3		403	402	609	607	613	613
Run4		402	608	611	610	606	606
Run5		402	604	608	606	613	606
Run6		405	608	605	605	605	607
Run7		402	404	606	605	605	607
Run8		201	403	604	605	607	607
Run9		412	606	605	606	605	607
Run10		402	402	604	615	608	607
Avg		383.4	524.8	606.9	607.6	608.3	608.4
Error		64.16	105.05	2.69	3.34	3.62	3.34
Algorithm	FDP2						
Run1		402	604	606	624	606	608
Run2		402	607	611	612	645	612

Run3	403	402	608	610	611	611
Run4	402	403	607	607	606	607
Run5	402	607	608	606	605	607
Run6	402	606	604	611	608	606
Run7	403	607	604	605	608	607
Run8	412	605	604	605	606	606
Run9	403	604	605	606	607	606
Run10	201	604	605	605	606	614
Avg	383.2	564.9	606.2	609.1	610.8	608.4
Error	64.09	85.60	2.30	5.86	12.14	2.88

Algorithm	StableTD					
Run1	201	201	201	201	203	205
Run2	201	201	202	202	202	215
Run3	201	201	201	202	203	202
Run4	201	201	202	202	202	207
Run5	201	201	202	201	203	203
Run6	201	201	202	202	202	202
Run7	201	201	201	201	202	202
Run8	201	201	201	202	202	203
Run9	201	201	202	201	202	202
Run10	201	202	202	202	203	203
Avg	201	201.1	201.6	201.6	202.4	204.4
Error	0.00	0.32	0.52	0.52	0.52	4.06

Algorithm	StableBU					
Run1	401	602	1003	1004	1606	1005
Run2	402	657	1003	1205	1356	1406
Run3	402	602	804	1204	1204	1605
Run4	402	802	1005	1204	1306	1205
Run5	402	602	1003	1086	1615	1204
Run6	402	602	1003	1215	1275	1767
Run7	402	601	1004	1002	1004	1605
Run8	401	602	802	1204	1450	1519
Run9	401	802	1027	1004	1404	1916
Run10	401	602	802	1204	1203	1204
Avg	401.6	647.4	945.6	1133.2	1342.3	1443.6
Error	0.52	83.28	98.90	97.02	188.07	288.88

High Computational Efforts

Stages		5	10	15	20	25	30
Algorithm	Default						
Run1		574	1145	1717	2289	3024	3757
Run2		573	1146	1718	2290	3185	3596
Run3		573	1145	1718	2289	3023	3432
Run4		574	1308	1717	2288	2861	3433
Run5		574	1307	1718	2289	2863	3433
Run6		574	1145	1718	2289	2861	3433
Run7		573	1145	1718	2289	2861	3433
Run8		573	1145	1717	2290	3024	3434
Run9		574	1145	1717	2289	3023	3435
Run10		574	1146	1717	2289	2862	3433
Avg		573.6	1177.7	1717.5	2289.1	2958.7	3481.9
Error		0.52	68.41	0.53	0.57	113.11	109.36
Algorithm	Simple						
Run1		603	1204	1434	1617	1409	1608
Run2		802	1004	1205	1406	1448	1836
Run3		802	1012	1407	1442	1529	1782
Run4		803	803	1248	1607	1807	2008
Run5		603	1004	1273	1406	1405	1920
Run6		615	1005	1206	1742	1759	2107
Run7		625	1004	1204	1760	1687	1833
Run8		611	1003	1204	1605	1605	1806
Run9		602	810	1522	1204	1404	1805
Run10		602	1003	1581	1217	1638	1805
Avg		666.8	985.2	1328.4	1500.6	1569.1	1851
Error		93.80	112.83	145.03	197.50	152.27	135.63
Algorithm	FDP1						
Run1		603	624	809	644	817	726
Run2		616	809	611	812	818	816
Run3		605	630	608	620	700	809
Run4		605	644	607	778	807	704
Run5		603	607	807	658	728	810
Run6		604	603	604	658	812	638
Run7		604	603	810	666	636	636
Run8		605	604	604	605	629	809
Run9		604	604	604	624	624	676
Run10		604	604	716	689	645	809
Avg		605.3	633.2	678	675.4	721.6	743.3
Error		3.83	63.41	96.34	68.08	85.42	75.76
Algorithm	FDP2						
Run1		603	604	612	816	814	674
Run2		614	660	607	816	691	679

Run3	604	810	610	864	654	657
Run4	604	612	610	1215	634	655
Run5	604	607	752	807	670	767
Run6	603	619	806	806	625	651
Run7	605	804	604	665	641	644
Run8	605	603	805	657	810	882
Run9	605	618	806	673	687	687
Run10	604	606	1008	806	656	817
Avg	605.1	654.3	722	812.5	688.2	711.3
Error	3.21	82.17	136.64	159.93	68.61	82.14

Algorithm	StableTD					
Run1	201	201	202	202	202	202
Run2	201	201	202	202	202	203
Run3	201	201	201	202	203	202
Run4	201	202	201	202	202	203
Run5	201	201	202	201	202	203
Run6	201	201	201	201	202	203
Run7	201	201	201	201	202	202
Run8	201	201	202	202	202	203
Run9	201	201	202	201	203	203
Run10	201	201	202	202	202	203
Avg	201	201.1	201.6	201.6	202.2	202.7
Error	0.00	0.32	0.52	0.52	0.42	0.48

Algorithm	StableBU					
Run1	640	1204	1404	1606	2007	2065
Run2	803	1090	1805	1582	1443	3009
Run3	602	1056	1405	1806	1615	1670
Run4	803	1145	1437	1606	1805	2531
Run5	802	1013	1605	1663	1648	2754
Run6	601	1404	1805	1616	1529	2405
Run7	619	1228	1607	1426	1882	2605
Run8	802	1003	1065	1408	1403	2744
Run9	802	1004	1605	1604	1603	2731
Run10	802	1369	1805	1805	1804	2605
Avg	727.6	1151.6	1554.3	1612.2	1673.9	2511.9
Error	97.06	147.42	234.41	131.13	195.86	386.61