Efficient Engineering and Execution of Pipe-and-Filter Architectures

Dissertation

M.Sc. Christian Wulf

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2019

1. Gutachter:	Prof. Dr. Wilhelm Hasselbring Christian-Albrechts-Universität zu Kiel
2. Gutachter:	Prof. Dr. Steffen Becker Universität Stuttgart

Datum der mündlichen Prüfung: 19. Juli 2019

Zusammenfassung

Pipe-and-Filter (P&F) ist ein wohlbekannter und häufig verwendeter Architekturstil. Allerdings gibt es unseres Wissens nach kein P&F-Framework, das beliebige P&F-Architekturen sowohl modellieren als auch ausführen kann. Beispielsweise unterstützen die Frameworks FastFlow, StreamIT and Spark nicht mehrere Input- und Output-Ströme pro Filter, sodass sie keine Verzweigungen modellieren können. Andere Frameworks beschränken sich auf sehr spezielle Anwendungsfälle oder lassen die Typsicherheit zwischen zwei miteinander verbundenen Filtern außer Acht. Außerdem ist eine effiziente parallele Ausführung von P&F-Architekturen weiterhin eine Herausforderung. Obwohl einige vorhandene Frameworks Filter parallel ausführen können, gibt es noch viel Optimierungspotential. Leider besitzen die meisten Frameworks kaum Möglichkeiten, die einzig vorhandene Ausführungsstrategie ohne großen Aufwand anzupassen.

In dieser Arbeit präsentieren wir unser generisches und paralleles P&F-Framework TeeTime. Es kann beliebige P&F-Architekturen sowohl modellieren als auch ausführen. Gleichzeitig ist es offen für Modifikationen, um mit dem P&F-Stil zu experimentieren. Zudem erlaubt es, Filter effizient und parallel auf heutigen Multi-core-Systemen auszuführen.

Umfangreiche Laborexperimente zeigen, dass TeeTime nur einen sehr geringen und in gewissen Fällen gar keinen zusätzlichen Laufzeit-Overhead im Vergleich zu Implementierungen ohne P&F-Abstraktionen erfordert. Außerdem weisen wir TeeTimes breite Anwendbarkeit und Erweiterbarkeit mit Hilfe von zahlreichen Fallstudien aus der Forschung und der Industrie nach. Wir bieten hierzu ein Replication-Package an, das die Daten und den Quellcode all unserer Experimente beinhaltet. So ermöglichen wir die Verifizierbarkeit sowie die Wiederholbarkeit unserer Experimente und erlauben das Nachvollziehen der Ergebnisse. Zudem bieten wir Referenzimplementierungen für TeeTime in Java sowie in C++ als Open-Source-Software unter `https://teetime-framework.github.io` an.

Abstract

Pipe-and-Filter (P&F) is a well-known and often used architectural style. However, to the best of our knowledge, there is no P&F framework which can model and execute generic P&F architectures. For example, the frameworks Fastflow, StreamIt, and Spark do not support multiple input and output streams per filter and thus cannot model branches. Other frameworks focus on very specific use cases and neglect type-safety when interconnecting filters. Furthermore, an efficient parallel execution of P&F architectures is still an open challenge. Although some available frameworks can execute filters in parallel, there is much potential for optimization. Unfortunately, most frameworks have a fixed execution strategy which cannot be altered without major changes.

In this thesis, we present our generic and parallel P&F framework TeeTime. It is able to model and to execute arbitrary P&F architectures. Simultaneously, it is open for modifications in order to experiment with the P&F style. Moreover, it allows to execute filters in parallel by utilizing the capabilities of contemporary multi-core processor systems.

Extensive lab experiments show that TeeTime imposes a very low and in certain cases even *no* runtime overhead compared to implementations without framework abstractions. Moreover, several case studies from research and industry show TeeTime's broad applicability and extensibility. We provide a replication package containing all our experimental data and code to facilitate the verifiability, repeatability, and further extensibility of our results. Furthermore, we provide reference implementations for TeeTime in Java and in C++ as open source software on `https://teetime-framework.github.io`.

Preface
by Prof. Dr. Wilhelm Hasselbring

Software Engineering of parallel and distributed software systems poses specific challenges for software engineers. The engineered parallel and distributed software systems have to be correct as well as efficient and scalable.

Architectural styles and design patterns provide proven solutions to recurring tasks in software development, including the development of such parallel and distributed software systems. Since three decades, this design knowledge is systematically documented in appropriate catalogs for reuse.

The so-called "pipe-and-filter" style is one of the first architectural styles, documented early in the nineties. Even if this style can be considered a "classic" style, it remains a great challenge to implement pipe-and-filter systems efficient and scalable on parallel and distributed hardware platforms.

In this thesis, Christian Wulf designs, implements and evaluates the new, innovative TeeTime approach to realize efficient pipe-and-filter systems in an efficient way. Besides the conceptual work, this work contains a significant experimental part with an implementation and a multifaceted evaluation. This engineering dissertation has been extensively evaluated with various experiments, including data from an industrial system.

Notably, this thesis already now has significant impact. The new Kieker monitoring analysis component has been migrated to TeeTime, and TeeTime is used in various follow-up projects.

If you are interested in designing and implementing pipe-and-filter software systems, this is a recommended reading for you.

Wilhelm Hasselbring
Kiel, August 2019

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Pipe-and-Filter (P&F) architectural style divides a complex task into several successive subtasks such that each of them is implemented by a separate, independent so-called *filter*. Filters communicate with each other by transferring data via *pipes*. Since its first descriptions, it has evolved through various enhancements. Nowadays all standard textbooks on software architecture [TMD09] and software architecture patterns [BMR+96; MSM04] cover this style.

Due to its potential for a high modularity, high throughput, and small memory footprint, the P&F style can be applied in various domains of application. One of the most famous domains is probably the consecutive execution of processes via Unix's pipes. In this case, the P&F style is applied at the operating system level. However, there are also many P&F scenarios which take place at the application level. Two present examples from industry are dispatchers in web frameworks and compilers for domain-specific languages. We use the P&F style for dynamic analysis with Kieker [HWH12], for executing performance tests with RadarGun [HWH17; Bar18], for analyzing runtime user behavior with iObserve [HHJ+13], and for live processing of high-volume monitored traces with ExplorViz [FWB+13; FRH15; FKH17].

1.1 Motivation and Problem Statement

Due to its maturity and modularity, one might expect that the P&F style would have long been provided as a P&F framework and be used in several areas of applications by many different users. So far, however, it is common practice that most developers and researchers write their own P&F implementations from scratch tailored to their specific use cases and require-

ments. This prevents others from effectively reusing available P&F implementations. While there are some frameworks, such as Fastflow [ADK+14], StreamIt [GTA06], Spark [ZXW+16; Foub], Storm [Fouc], Akka [Lig], and GRAMSP [SFB+09], which cover a broader class of P&F architectures, they still focus on specific use cases. Fastflow, StreamIt, and Spark, e.g., are designed to model and execute streaming applications only. These frameworks do not support more than one input and output stream and thus cannot model branches. For the same reason, they often do not support feedback loops. Although some frameworks provide support for loops, they only do so for particular filters. In contrast, an actor in actor-based frameworks, such as Akka, can output to multiple receivers. However, it still has only one input port—often called mailbox. In addition, this mailbox is also untyped such that each incoming message needs to be checked and casted according to its type. Thus, a faulty connection between actors can only be detected at run time. On the other hand, GRAMPS is tailored to graphics pipelines.

Furthermore, such custom P&F implementations often handle concurrency, if at all, only at a coarse-grained level which neglects parallelization potential of contemporary multi-core processor systems. For example, consider a quad-core system on which we run a P&F architecture using a naïve scheduling executing each filter in a dedicated thread. This simple parallelization approach is not very effective when (1) the workload is unevenly distributed over all filters and when (2) the number of available processing units exceeds the number of filters. In the first case, parallelizing all filters can lead to a waste of resources since only the slowest filter is responsible for the overall throughput. In the second case, some processing units remain unused. Hence, finding an efficient, balanced configuration of filters is not a trivial task. Recent research [ADK+14; GTA06; SFB+09; SQK+10] shows that there is still a high potential for optimization, especially in the context of parallelization and the execution on heterogeneous platforms. Thus, an efficient parallel execution of P&F architectures is still work-in-progress and requires further research.

For this reason, a P&F framework must also be extensible and open for modifications. Reusable and composable filters would additionally increase the modularization. Filter scheduling should not be fixed, but exchangeable. Pipe implementations should not be hard-coded, but individually adjustable.

However, most custom P&F implementations are not that flexible.

Finally, the unavailability of a generic and parallel P&F framework also results from obsolete and incomplete descriptions of the P&F style itself. As a consequence, the P&F style is easily misunderstood and implemented in a way that leads to avoidable disadvantages in modularity, reusability, maintainability, and performance. For example, Avgeriou et al. [AZ05] argue that pipes introduce a significant, non-negligible runtime overhead. We show in this thesis that this statement is no longer valid on shared-memory systems. Furthermore, other researchers [BMR+96; SG96; TMD09] describe the P&F style in detail, but neglect to explain how to efficiently execute it on multi-core systems. In this thesis, we present two scheduling approaches and demonstrate their efficiency based on several performance experiments.

1.2 Approach Overview and Contributions

As mentioned in Section 1.1, most developers and researchers use their own custom P&F implementations which are either domain-specific or handle concurrency poorly. Therefore, we conclude that there is a lack of an easy-to-use, broadly applicable, parallel P&F framework. Such a framework should be able to cover all kinds of P&F architectures and simultaneously provide high performance with a low overhead caused by the framework implementation. Hence, we propose and evaluate our generic and parallel P&F framework *TeeTime* which fulfills these criteria. The individual scientific contributions (SC1-SC4) are described below:

SC1: The Generic and Parallel P&F Framework *TeeTime*

In this thesis, we review the P&F architectural style by surveying its evolution from its introduction as a software architectural style in the eighties until today. In particular, we identify obsolete and wrong assessments concerning the relation between the components and the performance of the P&F style. Based on this assessment, we derive requirements on a generic, extensible, and parallel P&F framework, and provide TeeTime as

such a framework for reuse by others. To the best of our knowledge, it is the first framework which is able to model and to execute arbitrary P&F architectures while simultaneously offering a high performance. Besides implementing the well-known P&F concepts, TeeTime provides many novel extensions to the P&F style. Most of them focus on a high throughput by means of parallelization on cache-coherent multi-core systems.

For example, with **SC2**, we propose a new filter scheduling approach which employs a mix of synchronized and unsynchronized pipes to execute filters in parallel. Moreover, with **SC3**, we present a scalable and incremental approach to parallelize almost arbitrary filters in a P&F architecture. Additionally, with **SC4**, we provide two new live visualizations for P&F architectures to optimize and to debug their execution behavior. Beyond that, we present an additional filter scheduling approach for comparison, multiple complementary parallelization approaches, support for a distributed P&F execution, and a domain-specific language to define P&F configurations.

TeeTime targets researchers, software architects, and software developers who intend to experiment with the P&F architectural style or just to use it within their applications. We provide a Java-based and a C++-based reference implementation of our framework as open-source software [Wul15; WHO17]. With these implementations, we also show that the general TeeTime framework architecture is language-independent and can be implemented with many more object-oriented programming languages.

Furthermore, we conducted an extensive performance evaluation which shows that the TeeTime reference implementations impose only a small runtime overhead. Regarding the Java implementation in particular, we show that (1) TeeTime's P&F abstractions introduce either a negligible overhead or none at all, and that (2) the pipes, although synchronized, can still achieve a throughput of many billions elements per second.

SC2: A New Filter Scheduling Approach

We present a new filter scheduling approach for non-distributed P&F architectures on multi-core systems. It focuses on a minimal communication between threads to increase cache locality and to speed up the overall

execution. At the outset of the execution, our scheduling approach assigns a distinct subset of all filters to each of the used threads. Thus, each thread exclusively owns its filters and executes them according to a hybrid push/pull model [BMR+96]. This way of scheduling allows for connecting these intra-thread filters with unsynchronized pipes and inter-thread filters with synchronized pipes. Moreover, it avoids a global coordination by letting each thread schedule its filters by itself. In Section 9.1 and Section 13.2, we show its feasibility and its performance (1) by providing a proof-of-concept implementation integrated in TeeTime and (2) by comparing it with another commonly used globally coordinated scheduling approach [Orad; SFB+09; SLY+11].

SC3: A New P&F Parallelization Approach

We present a new parallelization approach for P&F architectures. We introduce a new composite filter which wraps an existing filter to duplicate it either at compile-time or at run-time. In this way, we can, for example, create more instances of the slowest filter and thus execute them in parallel to speed up the overall execution. In an extensive evaluation, we show that this approach scales well with the number of processor cores. In particular, it allows us to utilize idle resources when having fewer filters than cores. Moreover, by applying a self-adaptive manager on top, we can adapt the number of duplicates according to the current workload at run-time. In this way, we can react to workload variations in a modularized way. Since our composite filter can wrap almost all kinds of filters, we can successively apply it on all filters in a P&F architecture. Hence, our approach represents a scalable, balancing, and incremental way to introduce parallelism into P&F architectures. In Section 9.2 and Section 13.3, we present our approach, our proof-of-concept implementation, and our evaluation in more detail.

SC4: Live Visualization of TeeTime-based Applications

We present a debugging and profiling approach which allows us to monitor the execution of an arbitrary TeeTime-based application and to visualize the results live at the runtime of this monitored application. The monitoring

part records and aggregates all filter executions to derive the execution time of each filter within a user-defined time interval. The visualization part displays the whole P&F architecture live at its runtime in two different new diagrams. The *Execution Behavior Diagram* colors each filter based on the filter's aggregated execution time within the currently selected time interval. This diagram makes it possible to identify the most active and the most idle filters during a particular time interval. The *Timing Behavior Diagram* displays all filter instances side by side and shows their executions at each point in time. This diagram allows to identify the duration of each execution and highlights the concurrent execution of different filters. Hence, our approach optimizes the performance of TeeTime-based applications within the development process. It eliminates any delay between the execution and the analysis. In Section 10.3.2, we present our approach in more detail by applying it on an example scenario. In addition to that, we show the corresponding visualizations of our associated proof-of-concept implementation.

1.3 Preliminary Work

In this section, we briefly summarize our 11 related publications and 13 co-supervised student works that have contributed to this thesis in ascending chronological order. For a more detailed description on the term papers as well as on the bachelor's and master's theses, we refer to the corresponding works. Note that, for now, we will use the term *stage* as synonym for *filter*.

1.3.1 Related Publications

▻ [FWW+13] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. "Live trace visualization for comprehending large software landscapes: the explorviz approach." In: *1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. Sept. 2013, pp. 1–4

In this work, we propose a P&F architecture for processing large traces in real-time. It builds the basis for rudimentary design decisions which were taken into account in the development of our P&F framework

TeeTime, in particular in terms of parallelization and high-throughput processing.

▻ [WWF+13] Jan Waller, Christian Wulf, Florian Fittkau, Philipp Döhring, and Wilhelm Hasselbring. "SynchroVis: 3D Visualization of Monitoring Traces in the City Metaphor for Analyzing Concurrency." In: *1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. Sept. 2013, pp. 1–4

In this work, we propose a 3D visualization of monitoring traces for analyzing concurrency issues. It provides an initial graphical insight into processing a large amount of data in parallel. In particular, it helped us with identifying bottlenecks in high-throughput scenarios and with developing an improved live visualization for TeeTime-based applications.

▻ [WEH14] Christian Wulf, Nils Christian Ehmke, and Wilhelm Hasselbring. "Toward a Generic and Concurrency-Aware Pipes & Filters Framework." In: *Proceedings of the Symposium on Software Performance 2014: Joint Descartes/Kieker/Palladio Days*. Nov. 2014

In this work, we present the first version of our P&F framework TeeTime. We introduce its framework architecture and present our corresponding design decisions. In particular, we describe how TeeTime is able to model arbitrary P&F architectures and how it executes them efficiently on multi-core systems.

▻ [DW16] Gunnar Dittrich and Christian Wulf. "Extraction of operational workflow-based user behavior profiles for software modernization." In: *Proceedings of the Symposium on Software Performance*. Nov. 2016

In this work, we monitor an industrial application and extract its operational user behavior profiles in order to identify components for software modernization. For this purpose, we successfully build and apply two TeeTime-based analyses.

▻ [EW16] Florian Echternkamp and Christian Wulf. "Kieker in Eclipse - a plug-in for application performance monitoring and dynamic analysis in Eclipse." In: *Proceedings of the Symposium on Software Performance*. Nov. 2016

In this work, we present our Eclipse plug-in for Kieker which allows monitoring and analyzing Eclipse projects. It provides an integrated solution for application performance monitoring and profiling. All corresponding analyses and visualizations are built with TeeTime.

▻ [JW16] Reiner Jung and Christian Wulf. "Advanced typing for the kieker instrumentation languages." In: *Proceedings of the Symposium on Software Performance*. Nov. 2016

In this work, we propose an improved version of Kieker's instrumentation record language, a domain-specific language for defining monitoring records. We used this language to specify the records necessary for our live visualization of TeeTime-based applications. Moreover, it inspired the development of our own domain-specific language for TeeTime-based configurations.

▻ [SW16] Hannes Strubel and Christian Wulf. "Refactoring Kieker's Monitoring Component to further Reduce the Runtime Overhead." In: *Proceedings of the Symposium on Software Performance*. Nov. 2016

In this work, we describe how we have reduced the runtime overhead of Kieker's monitoring component. For the most part, we could successfully transfer these insights into the optimization of TeeTime's implementations.

▻ [WWH16] Christian Wulf, Christian Claus Wiechmann, and Wilhelm Hasselbring. "Increasing the Throughput of Pipe-and-Filter Architectures by Integrating the Task Farm Parallelization Pattern." In: *Proceedings of the International Symposium on Component Based Software Engineering (CBSE)*. 2016

[WH17] Christian Wulf and Wilhelm Hasselbring. "Increasing the throughput of pipe-and-filter architectures by integrating the task farm parallelization pattern." In: *Proceedings of the Software Engineering (SE'17)*. Vol. P-267. Lecture Notes in Informatics (LNI). Feb. 2017, pp. 83–84

In these works, we present our task farm stage which integrates the task farm parallelization pattern into P&F architectures in order to increase the throughput on multi-core systems. Moreover, we present a

self-adaptive manager associated with the task farm stage which automatically adapts the grade of parallelism at runtime based on the current workload. Both for the implementation and for the evaluation, we use TeeTime.

▻ [WHO17] Christian Wulf, Wilhelm Hasselbring, and Johannes Ohlemacher. "Parallel and generic pipe-and-filter architectures with TeeTime." In: *Proceedings of the International Conference on Software Architecture (ICSA)*. Apr. 2017

In this work, we present an updated, enhanced version of TeeTime's framework architecture and features. Moreover, we highlight TeeTime's extensibility by a small, real-world application example.

▻ [HWH17] Sören Henning, Christian Wulf, and Wilhelm Hasselbring. "RadarGun: Toward a Performance Testing Framework." In: *Symposium on Software Performance*. Nov. 2017

In this work, we present requirements on a performance testing framework to distinguish it from a functional testing framework and a benchmarking framework. Based on these requirements, we propose such a performance testing framework for Java, called RadarGun. We use this framework to automatically measure the performance of TeeTime in several different P&F scenarios. Moreover, we use it for a systematic way of ensuring a high throughput for several years of development.

1.3.2 Co-Supervised Student Works

Below, we list and briefly describe in chronological order works related to TeeTime.

▻ [Wie15] Christian Claus Wiechmann. "On improving the performance of Pipe-and-Filter architectures by adding support for self-adaptive task farms." Master's thesis. Department of Computer Science, Oct. 2015

Wiechmann presents the first prototype implementation of TeeTime's task farm stage and its associated self-adaptive manager. Moreover, he performs an in-depth evaluation of the feasibility, the performance, and the scalability. This work builds the base for [WWH16].

▻ [Zlo15] Martin Zloch. “Development of a domain-specific language for Pipe-and-Filter stage developers.” Studienarbeit. Department of Computer Science, 2015

Zloch presents the first prototype for a domain-specific language to define TeeTime stages. It abstracts from a particular programming language and provides short notations to declare ports and to define the execution logic.

▻ [Neu16] Mathis Neumann. “Extension of a domain-specific language for Pipe-and-Filter stage developers in teetime.” Studienarbeit. Department of Computer Science, 2016

Neumann presents an enhanced version of the domain-specific language by Zloch [Zlo15]. It allows for declaring multiple input ports per stage and reacting to the start event and the termination event. Furthermore, it informs the user within an integrated development environment about syntax errors and language constraints.

▻ [Peg16] Adrian Pegler. “Evaluating approaches to detect bottlenecks in the Pipe & Filter framework TeeTime.” Bachelor’s thesis. Department of Computer Science, Mar. 2016

Pegler presents a systematic approach to identify and to evaluate bottlenecks in P&F architectures. He evaluates his approach by using TeeTime.

▻ [Ado16] Marc Adolf. “Self-adapting execution of Pipe-and-Filter systems.” Master’s thesis. Department of Computer Science, Mar. 2016

Adolf presents the first concept and prototype implementation for a self-adaptive execution of P&F architectures. He adopts the self-adaptive manager of the task farm stage and applies it to arbitrary stages. In this way, he achieves an automatic workload balancing across all available stages of a given TeeTime-based configuration.

▻ [Dit16] Gunnar Dittrich. “Extraction of user behavior profiles for software modernization.” Master’s thesis. Department of Computer Science, May 2016

Dittrich presents the first concept and prototype implementation for the extraction and the visualization of operational user behavior profiles. He evaluates his approach with respect to an industrial application in order to prioritize particular modernization tasks. This work builds the base for [DW16].

▻ [Hen16] Sören Henning. "Visualization of performance anomalies with Kieker." Bachelor's thesis. Department of Computer Science, Sept. 2016

Henning presents a concept and a prototype implementation for analyzing and visualizing performance anomalies. For this purpose, he builds a P&F-based analysis using TeeTime.

▻ [Tav16] Nelson Tavares de Sousa. "Live execution time visualization of component-based software systems." Master's thesis. Department of Computer Science, Nov. 2016

Tavares de Sousa presents a concept and a prototype implementation for two new live visualizations of component-based software systems. These visualizations represent the execution state, time, and behavior of each single component at the runtime of the system. The author evaluates his visualizations and the underlying live processing architecture by using a P&F example application built with TeeTime.

▻ [Ohl16] Johannes Ohlemacher. "Conception and development of a Pipe & Filter framework for C++." Master's thesis. Department of Computer Science, Nov. 2016

Ohlemacher presents his C++ reference implementation for TeeTime. In his work, he adopts the general TeeTime framework architecture and evaluated it with several different P&F example applications in C++. The results show that the concepts and the high performance of TeeTime are independent of Java and C++.

▻ [Zlo16] Martin Zloch. "Development of a domain-specific language for Pipe-and-Filter configuration builders." Studienarbeit. Department of Computer Science, 2016

Zloch presents the first prototype for a domain-specific language to define TeeTime configurations. It abstracts from a particular programming

language and provides short notations to declare and to connect stages. This prototype did not yet support distributed P&F architectures (see below for an enhanced version by Echternkamp [Ech17]).

▷ [Str16] Hannes Strubel. "Terminierung von Pipe & Filter Architekturen mit Feedback-Schleifen." Studienarbeit. Department of Computer Science, 2016

Strubel presents a concept and an extension to TeeTime to support feedback loops in TeeTime-based P&F architectures.

▷ [Str17] Hannes Strubel. "Monitoring distributed traces with Kieker." Master's thesis. Department of Computer Science, May 2017

Strubel presents several concepts for monitoring distributed traces with Kieker. For their evaluation, he implements an associated trace processing analysis using TeeTime. His work therefore shows another successful application of TeeTime.

▷ [Ech17] Florian Echternkamp. "Distributed Pipe-and-Filter architectures with TeeTime." Master's thesis. Department of Computer Science, May 2017

Echternkamp presents an extension to TeeTime such that it is able to model and to execute distributed P&F architectures. He evaluates 13 Java frameworks which have relations to distributed communication or fault tolerance and integrated the winner into TeeTime. Moreover, he develops an extension to TeeTime's domain-specific language to support the definition and the automatic deployment of distributed P&F architectures.

1.4 Document Structure

This thesis consists of the following four parts:

▷ Part I describes the foundations for this thesis.

- ▷ In Chapter 2, we introduce the P&F architectural style and describe its evolution and its implementations within the last three decades.

This chapter builds the basis for this thesis and introduces common terms as well as concepts of the Pipe-and-Filter domain.

- ▻ In Chapter 3, we give an overview of shared queues since they are crucial for the design and the performance of TeeTime's pipes.
- ▻ In Chapter 4, we introduce the patterns *pipeline* and *task farm* for parallel systems. Both play a central role for the parallel execution of TeeTime-based P&F architectures.
- ▻ In Chapter 5, we describe the actor model which we use for a distributed execution of TeeTime-based P&F architectures.
- ▻ In Chapter 6, we introduce the concept of domain-specific languages. In particular, we describe their areas of application and present useful tooling for their development. This chapter builds the base for our domain-specific language which allows to define P&F configuration for TeeTime.

▻ Part II presents our generic and parallel P&F framework TeeTime.

- ▻ In Chapter 7, we describe our research scope, research plan, and research methods.
- ▻ In Chapter 8, we present TeeTime's framework architecture. We describe how TeeTime is able to model arbitrary P&F architectures and how it efficiently executes them on parallel and on distributed systems.
- ▻ In Chapter 9, we describe TeeTime's concepts to execute P&F-based applications.
- ▻ In Chapter 10, we introduce the concepts which we provide when developing P&F-based applications with TeeTime.

▻ Part III

- ▻ In Chapter 11, we evaluate the language independence of TeeTime. In particular, we present our two reference implementations in Java and in C++.
- ▻ In Chapter 12, we evaluate TeeTime's feasibility by means of five different case studies.

- ▻ In Chapter 13, we evaluate TeeTime's framework overhead with different configurations and workload on different processor architectures.
- ▻ In Chapter 14, we evaluate TeeTime's reusability and extensibility by means of an in-depth analysis of the framework itself and by two different case studies.
- ▻ In Chapter 15, we present related work concerning the modeling and the execution of P&F-based applications.

▻ Part IV

- ▻ In Chapter 16, we conclude this thesis.
- ▻ In Chapter 17, we present future work.

Part I

Foundations

Chapter 2

The Evolution of the Pipe-and-Filter Style

We start off the foundations part by going through the evolution of the Pipe-and-Filter style. This will give us a detailed overview of the basic entities, different concepts, and various implementation approaches including their pros and cons. Thus, this chapter serves as the basis for the design decisions of our Pipe-and-Filter framework TeeTime.

2.1 The Early Days of Pipes and Filters

The P&F architectural style significantly evolved over the last three decades. Mary Shaw is one of the first authors who explicitly described this software architecture style. In her work [Sha89], she defines it as a useful system organization that consists of filters accepting one stream of inputs and emitting one stream of outputs. An example P&F architecture is shown in Figure 2.1a. It represents Parnas' *Key Word In Context*[1] program [Par72] as P&F implementation [RMJ06]. It reads a text file, produces circular shifts of the lines, alphabetizes these shifts, and finally prints out the results.

Allen et al. propose [AG92] and refine [AG94; All97] the first formalization of the P&F style using the Z specification language [Spi89]. They define a filter by its name (i.e., a unique identifier), its program (i.e., its execution logic), and its input and output ports. Each port is typed such that it accepts or sends data of a particular type only. One execution of a filter proceeds as

[1] The *Key Word In Context* format was the most common format in the pre-digital era for indexing books in libraries.

follows: It reads data from one or more of its input ports, transforms that data, and writes the results to one or more of its output ports.

Hence, ports allow us to go beyond the modeling of filters that are connected in a line. They enable filters to process an arbitrary number of input and output streams. To distinguish both kinds of P&F architectures, Buschmann et al. [BMR+96] call the former *pipelines* and the latter *Tee-and-Join pipeline systems*.

2.2 Dataflow Variations

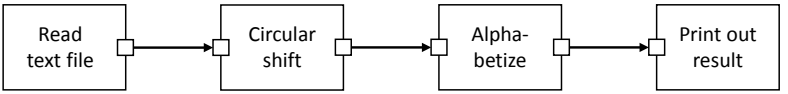
When using ports, data does not only need to flow forward anymore. It may also flow backward to a previous filter forming a feedback loop or to the same filter using a reflexive pipe (example P&F applications are described by Sugerman et al. [SFB+09] and Aldinucci et al. [ADK+14]). Branches are possible, too. The conditional `if-then-else` control structure, for example, can easily be modeled by a filter with one output port for the `true` case and one output port for the `false` case. Figure 2.1b and Figure 2.1c show an example P&F loop and an example P&F branch, respectively.

2.3 Pipes as First-Class Entities

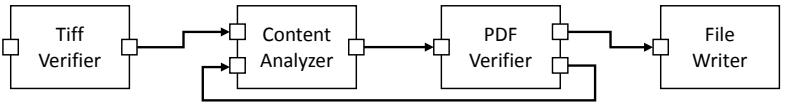
Besides the architectural components *pipes* and *filters* (including ports), Abowd et al. [AAG93] additionally introduce the *configuration* as another key component in P&F architectures. Such a configuration describes a collection of filter instances and pipe instances that are properly connected.

Furthermore, they formalize the encapsulation of a configuration as a higher-level filter. In this way, they introduce hierarchy within the P&F style and thus allow a filter to be composed of several other filters. Shaw [Sha96] calls such a filter a *composite filter* to distinguish it from a *primitive filter*. A detailed formal specification of Abowd et al.'s syntactic and semantic model of the P&F style is given in [AAG95].

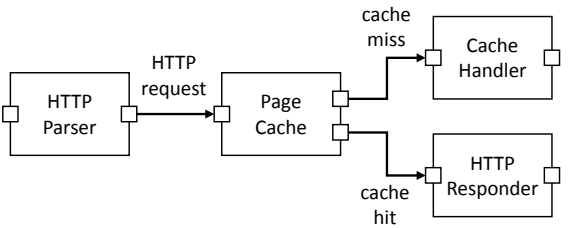
While the first authors [Sha89; AG92; AAG93] of the P&F style and authors of textbooks on software architecture, e.g., Taylor et al. [TMD09], recommend to model a pipe as first-class entity, many real-world P&F



(a) An example pipeline: Parnas' *Keyword In Context* program [Par72] as P&F implementation [RMJ06].



(b) An example feedback loop: the Document Understanding and Analysis System of Gokhale et al. [GY06].



(c) An example branch: a cache stage with an output port for a cache hit and one for a cache miss used by Welsh et al. [WCB01].

Figure 2.1. Variations of P&F dataflow structures.

architectures and frameworks, e.g., FastFlow [ADK+14] or Pipes,[2] do not implement their advice. As a consequence, they typically have one or more of the following disadvantages [Sha96]:

Fixed neighbor filters. If a filter does not only specify its execution logic, but also concrete predecessor and successor filters, it is tightly coupled with them. This approach leads to an unnecessary dispersion of structural information among the whole P&F architecture. Moreover, it reduces the reusability of the filter.

Fixed communication medium. Interaction between filters can have various

[2] `http://tinkerpop.incubator.apache.org`

manifestations. Depending on the scenario, two filters need to communicate via a direct method call, via a queue, or via message passing. However, if a filter is limited to one of these concrete implementations, it cannot be applied to other scenarios without changing the filter's implementation.

Fixed communication protocol. Sharing information among filters can require more than a single step to succeed. In such situations, the communication follows a particular protocol. Without using a pipe, both the source filter and the target filter must know the protocol. Thus, both filters must be changed if the protocol changes.

2.4 The Myth of Pipe Overhead

As discussed in Section 2.3, pipes should be handled as distinct architectural entities. However, one might argue that pipes could introduce a significant, non-negligible runtime overhead since each element must then pass not only the filter entities, but also the pipe entities. Indeed, such an extra level of indirection could potentially lead to a performance decrease. However, most compilers apply method inlining [CH89] that allows to replace the method call by the method's body at compile-time. Thus, the additional call is not present at runtime. Modern just-in-time compilers, as for Java or C#, even perform sophisticated inlining operations automatically at runtime by collecting and using online profile information [Orac]. In fact, we observed this behavior when executing several P&F applications with our framework TeeTime. Moreover, we show in Section 13.1.1 by using an example P&F architecture that TeeTime imposes a negligible runtime overhead compared to a pure Java-based implementation without explicit pipes.

In addition to that, some authors [e.g., BMR+96; AZ05] claim that pipes also lead to a performance degrade because they perform data conversion between the producing filter and the consuming filter. The most familiar example, which performs such a data format conversion, is probably Unix' pipe command. However, we can avoid the resulting overhead if we limit the execution to shared memory systems. If two filters, running in different threads, intend to share data via a pipe, they only need to exchange the pointers pointing to that data. It is not necessary to serialize and deserialize

the data. Moreover, due to recent heavy optimizations in the area of synchronized queues [GMV08; WZT+13; ADK+12], inter-thread communication requires nowadays only a few more nanoseconds than intra-thread communication. In Part II, we provide a detailed description of how we implement and combine our synchronized and unsynchronized pipes to provide a high throughput. In Section 13.1.2, we show that our synchronized pipe performs best compared to all other tested synchronized pipe implementations.

2.5 Sharing State among Filters

The original P&F style defines a filter to be independent of other filters. Each filter may have an internal state which the filter can manipulate. One example is a filter that counts each data element that flows through it. For this purpose, it maintains a counter which is incremented for each processed data element. However, state sharing across non-adjacent filters and consequentially a global state are not intended by the P&F style [Sha89; AG92; GS93; BMR+96].

Nevertheless, there are P&F scenarios where non-adjacent filters do need to share data. One familiar example is the P&F-based, single-threaded compiler architecture described by Garlan et al. [GS93] and by Buschmann et al. [BMR+96]. All of the filters that are responsible for the lexicographical, the syntactic, and the semantic analysis, as well as for the code generation share the same symbol table for modularization and performance reasons. Otherwise, each filter would need to hold a separate local representation of the symbol table leading to the following disadvantages. If one of the filters updates its symbol table, it needs to instruct all the other filters to also update their symbol tables to retain data consistency. In this strict P&F architecture, keeping other filters up-to-date can only be achieved by passing an update notification element through the whole pipeline. This artificial distribution of an actually shared data structure as well as the corresponding update protocol lead to an overly complex and unintuitive approach. Moreover, it requires three times as much memory and introduces a performance overhead. Hence, some authors [AZ05; TMD09] propose to combine the P&F style with data-centered architectures, such as shared

repository or blackboard. In order to satisfy the demand of flexibility of a framework in general and the demand of high-throughput in particular, our framework TeeTime allows to model all variants and combinations of the P&F style.

2.6 Execution of Pipes and Filters

So far, we have described the design of P&F architectures. Now, we address the execution of them. Buschmann et al. [BMR+96] distinguish three scenarios to trigger the activity of a filter *F*. In the first scenario, the filter following *F* pulls output data from *F*. In the second scenario, the filter prior to *F* pushes new input data to *F*. In the third scenario, *F* is in an active loop pulling (or: polling) its input from its previous filter and pushing its output to its subsequent filter. Buschmann et al. call *F* in the first two cases *passive* and in the last one *active*. Here, *F* is assumed to have only one input port and one output port. However, the terminology also applies for more than one input and output ports.

If two active filters communicate with each other, the corresponding pipe handles the synchronization. If only one of these filters is active, the pipe can be implemented by a direct method call from the active to the passive filter. In this way, the pipe also handles the scheduling while otherwise making filter recombination challenging [BMR+96]. In Part II, we describe how TeeTime is able to perform this approach in a fully automatic manner at startup and even at runtime.

2.7 Parallel Execution of Pipes and Filters

Multiple active filters in a P&F architecture can run in parallel to increase the throughput. Allen et al. [AG92] were one of the first authors who propose a formal specification including concurrency aspects. However, they do not address filter scheduling to achieve a low latency. Furthermore, they do not address the synchronization overhead resulting from the pipes. Finally, they omit a discussion about an optimal thread-to-filter assignment strategy to maximize the application's throughput. Some works [GTA06;

GC08; SFB+09] target these open questions by presenting different kinds of parallelization approaches. For example, Suleman et al. [SQK+10] propose to choose at runtime the optimal configuration of active filters in terms of performance and power consumption. The variety of approaches shows that parallelizing P&F-based applications is complex and that there is much potential for optimization.

Nevertheless, it is often claimed that the performance of P&F architectures could easily be increased by parallelizing the filters. Of course, we can simply declare the filters as active to execute them concurrently. However, we always need to ensure that the computation effort is much higher than the communication effort. Otherwise the performance can even become lower than before due to additional synchronization costs. In some cases, two active filters with a low workload can be combined into a single active filter with a medium workload to increase the ratio of the computation and the communication cost [SW02]. This process is also known as *filter fusion* [GTA06]. Figure 2.2 shows an example of two active filters A and B that are combined into the active filter A+B.

Since the throughput of a P&F-based application is limited by the throughput of the slowest filter, optimizing this filter is crucial when tuning a P&F architecture. In Figure 2.2, the active filter D is split into two active filters D1 and D2 both with a lower workload than D had previously. Although D1 and D2 now need to synchronize, they can run concurrently. In addition, the progress in the area of synchronized, lock-free queues [ADK+12; WZT+13; GMV08] has reduced the contention and synchronization cost to a minimum for such single-producer/single-consumer scenarios. In Chapter 3, we give an introduction to these queues and describe in Part II how and when our framework TeeTime takes advantage of them.

An alternative to splitting a filter is to replicate it. In this way, we run two instances of the same filter concurrently [ADK+14; SFB+09]. A schematic illustration is shown by filter E in Figure 2.2. The latter approach is also known as a variant of the *Task Farm* parallelization pattern [Dan00; BDL+13a; WWH16]. We introduce this pattern in Section 4.2 and describe in Section 9.2 how we utilize it for TeeTime to increase the overall throughput.

In summary, finding an efficient, balanced configuration of active filters is not a trivial task and needs further research.

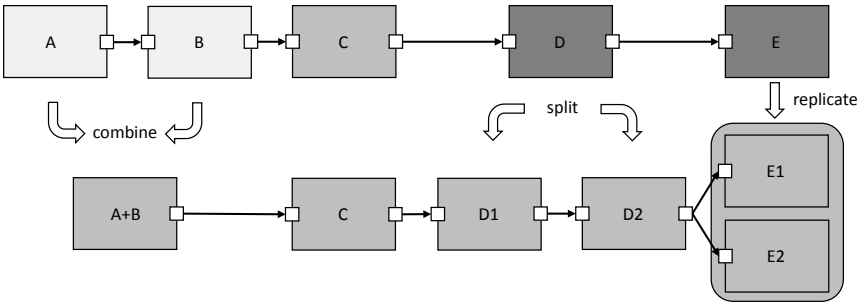


Figure 2.2. Possible operations to balance the workload of active filters in an initially unbalanced P&F architecture. Low, medium, and high workloads are depicted as light, medium, and dark gray, respectively.

2.8 Scheduling of Filters

Another non-trivial task is the efficient scheduling of filters. For example, a filter should only be executed if it can make any progress, i.e., if its input ports have any input data available to consume. Otherwise, the filter execution is a waste of resources and time. Otto et al. [OPT09] use a scheduler that maintains a thread pool and a filter queue. Each time a thread from the pool is idle, it takes a filter from the queue and executes it multiple times to amortize the overhead resulting from the synchronization and the queue operations. Finally, the thread returns the filter to the end of the queue.

Gordon et al. [GTA06] provide the StreamIt compiler which divides a P&F architecture into distinct subsets of filters and assigns each subset to a dedicated thread. These subsets are constructed at runtime by means of *decoupled software pipelining* [ORS+05]. This technique constitutes two distinct schedules: a loop phase and a steady-state loop. The loop phase initializes the execution of the P&F architecture until a steady-state has been reached. Based on this steady-state, load-balanced subsets are dynamically chosen and then executed such that the threads do not stall anymore.

Sugerman et al. [SFB+09] provide the GRAMPS compiler for graphics pipelines which cannot only exploit the CPU, but also the GPU. For this

purpose, their scheduling algorithm distinguishes *thread stages* and *shader stages*. Thread stages are stateful, long-lived, and are executed by the CPU. Shader stages are stateless, data-parallel, and are executed out-of-order by the GPU. An execution of such a stage is represented by a task and stored in a global task queue. The scheduler manages a number of worker processes which fetch these tasks from and put new tasks to the queue.

Sanchez et al. [SLY+11] change the scheduler of GRAMPS to further increase the performance. They avoid a globally shared task queue and assign a local task queue per stage to each worker process instead. In this way, they improve the temporal and the spatial locality. If a task queue for a particular stage of a worker process is empty, the worker process then steals tasks from one of the corresponding task queues of the other worker processes.

Although such work-stealing techniques [NAT+09; SBG+09] seem to be promising to automatically balance the load in P&F architectures, they are also criticized for high-throughput scenarios [SLY+11; DWG+12]. For example, the efficient synchronized single-producer/single-consumer queue, as mentioned in Section 2.7, does not support additional consumers which steal data from it. The queue supports only one single consumer. Adding support for stealing consumers would introduce an additional synchronization overhead and thus would decrease the throughput. Moreover, Kumar et al. [KFB+12] show in a case study that the steals-to-task ratio is often below 0.1%, i.e., steals are extremely rare.

TeeTime enables the integration of arbitrary scheduling strategies, such as a global scheduling algorithm based on a task queue (see Section 9.1.2), or a local algorithm based on work-stealing. Currently, TeeTime's default is to apply a simple, but efficient push-based scheduling approach for intra-thread scenarios (see Section 9.1.1). For inter-thread scenarios, it employs a busy-waiting strategy for an optimized high-throughput execution. Nevertheless, a blocking strategy can individually be configured for each pipe, if desired. We refer to Section 9.1 for a more detailed description.

2.9 Error Handling

Error handling is the most severe problem of the P&F style [BMR+96; TMD09]. Consider the example in which a P&F application has consumed three-quarters of its input, already produced half of its output, and some intermediate filter causes an error. At least, the error should be detected and associated with the filter and the concrete input element to allow for error handling and debugging. Afterwards, the error should be recorded either by a global or a local recorder. A global recorder may, however, merge error messages from different active filters in a non-obvious and unpredictable way [BMR+96]. Depending on the chosen error handling strategy, the P&F application can finally (1) ignore the error and continue, (2) terminate itself, or (3) try to recover to a previous, stable state. In particular, the third strategy is difficult to realize since the state of a P&F-based application is composed of the states of the individual filters. These filters can additionally be distributed among multiple different nodes which makes error handling even more difficult.

In Section 9.1.1, we present novel extensions to detect errors in P&F configurations. In this way, TeeTime is able to identify (1) an invalid assignment of multiple pipes to a single port and (2) an inconsistent definition of active and passive filters. In addition, we describe in Section 9.1.1 how TeeTime automatically detects runtime errors, enriches them with useful debugging information, and allows to handle them in an arbitrary, user-defined way.

Chapter 3

Shared Queues

The overhead of a P&F framework—and thus its performance—heavily depends on the overhead of the pipes. Hence, a P&F framework must provide an efficient way for stage-to-stage communication. Especially for the execution on parallel systems, it must provide a pipe implementation which supports an efficient synchronization across stages executed by different threads. Most available implementations for such a pipe use a shared queue.

A shared queue is a common data structure to exchange data between threads on a multi-core system. It often provides a first-in/first-out (FIFO) semantics and allows for an asynchronous communication by using an internal buffer, e.g., an array or a linked list. Hence, a shared queue perfectly serves as a base for a pipe implementation in a P&F framework.

However, in order to effectively utilize the capabilities of multi-core systems, we need to pay special attention on the usage scenario and the implementation of such a shared queue. There are queues which are optimized for a frequent or an infrequent access. Furthermore, there are queues which are correct for any number of producers or for a single producer only. In order to understand how TeeTime's shared queue achieves a high throughput, we first introduce the major performance affecting aspects of a queue in Section 3.1. Afterwards, we describe how the particular aspect *access rate* can be optimized by discussing lock-based, lock-free, and hybrid queues in Section 3.2-3.4. Finally, we consider the four most effective queue optimization techniques in Section 3.5.

3.1 Performance Affecting Aspects

The implementation of a shared queue heavily affects its performance. For this reason, we discuss the four major aspects below, namely its structure, its size, its access rate, and its number of concurrent participants.

Structure The structure of a queue determines how the queue is represented in memory. If the queue is arranged in a continuous block and accessed linearly, it can efficiently utilize the hardware's cache architecture. Hence, due to its natural sequential locality, an array-based implementation usually performs faster than a list-based implementation.

Size The size of a queue can be limited or unlimited (leaving aside the limited capacity of memory in general). A list-based implementation, for example, can usually hold an unlimited amount of elements. In contrast, an array-based implementation is initially limited to a fixed number of elements. Although there are approaches which allow an array-based implementation to resize automatically on demand, they are at the expense of performance.

Access Rate Depending on whether the shared queue should be accessed frequently or infrequently, it can be optimized to performance or to power consumption, respectively. A lock-free implementation cannot stall a running thread to wait for another thread. It only allows to perform busy-waiting if it should wait for the queue to become non-full or non-empty. In return, the overhead of its alternative synchronization mechanisms is usually much lower than a lock-based implementation. In Section 3.2 and Section 3.3, we describe in more detail how the lock-based queues and, respectively, the lock-free queues work.

Number of Concurrent Participants Some shared queue implementations differentiate between the maximum amount of concurrent producers and consumers. For example, lock-free queues can be categorized into one of

	Single consumer	Multiple consumers
Single producer	Sp/Sc queue	Sp/Mc queue
Multiple producers	Mp/Sc queue	Mp/Mc queue

Figure 3.1. Possible variants of lock-free queues depending on the maximum number of concurrent participants.

four groups depending on the allowed number of producers and of consumers. Figure 3.1 shows these groups in a corresponding matrix. This differentiation is made for the sake of performance. The more participants are allowed to access the queue concurrently, the higher is the synchronization effort for a correct implementation. For this reason, the Sp/Sc queue reaches the highest throughput. In Section 3.3, we describe in more detail what optimizations can be applied to which group of lock-free queues. In Section 13.1.2, we also compare these groups by means of a performance evaluation.

3.2 Lock-based Queues

A lock-based queue uses locks to synchronize the access between the producer threads and the consumer threads (or in short: producers and consumers). Both the producers and the consumers need to acquire a lock before accessing the queue, potentially stalling a thread. Listing 3.1 shows the typical structure of a lock-based queue implementation. A complete example implementation is given by Java's `ArrayBlockingQueue`[1].

The lock-based approach prevents concurrent access, even if different slots are accessed. It always locks the entire queue for each invocation of the `add` and the `remove` operation. Thus, this approach generally makes queues inefficient for high access rates, i.e., for high-throughput scenarios. In Section 13.1.2, we show the performance of these queues in comparison to the lock-free queues from Section 3.3.

[1] `https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/ArrayBlockingQueue.html`

```
1  class LockBasedQueue<E> {
2    // field declaration of 'lock'
3    boolean addToHead(E element) {
4      this.lock.lock();
5      try {
6        // add element to head
7        // signal condition "not empty anymore"
8      } finally {
9        this.lock.unlock();
10     }
11   }
12
13   E removeFromTail() {
14     this.lock.lock();
15     try {
16       // remove element from tail
17       // signal condition "not full anymore"
18     } finally {
19       this.lock.unlock();
20     }
21   }
22 }
```

Listing 3.1. A lock-based queue implementation.

3.3 Lock-free Queues

A lock-free queue [ADK+12; GMV08; WZT+13] uses atomic instructions of the CPU, such as compare-and-swap (CAS), instead of locks to synchronize values across threads. In this way, the queue avoids stalling and awakening its accessor threads which leads to a higher access rate. Moreover, a lock-free queue is often further optimized with respect to cache utilization and synchronization. We refer to Section 3.5 for a brief description of the most effective optimization techniques. Listing 3.2 shows the typical structure of a lock-free queue implementation. A complete example implementation is

given by JCTools's `SpScArrayQueue`.[2]

3.4 Hybrid Queues

A hybrid queue implementation tries to combine the advantages of both the lock-based and the lock-free implementation. Instead of checking the queue's state in a regular interval, i.e., with a constant backoff time, it uses an exponential backoff time. If the awaiting change does not happen within a few iterations, the executing thread is stalled. Eventually, this thread is awakened by the thread on the other side of the queue if the associated condition is satisfied. However, this condition has to be checked upon each queue access and thus causes additional overhead. For this reason, even hybrid queues are not suitable for high-throughput scenarios.

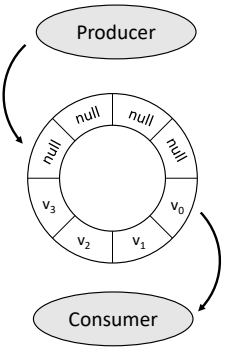
3.5 Optimization Techniques

Shared queues, like lock-free queues, are heavily optimized, especially with respect to cache utilization and synchronization. In the following, we briefly describe the most effective optimization techniques. Further such techniques are summarized by Wang et al. [WZT+13].

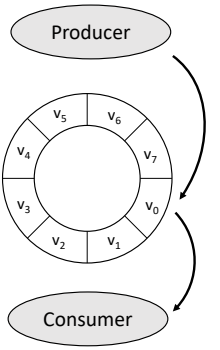
Cache Locality Preservation Most queue implementations use an array (as opposed to a linked list) to preserve cache locality. When a consumer accesses the tail slot of the array, the underlying processor automatically loads some subsequent slots into its cache for the sake of performance. In the case where the consumer reaches the last slot of the array, it continues by accessing the first slot again. The same behavior applies for a producer and is represented by the method `next(..)` as shown in Listing 3.2.

Data/Control Coupling If we limit the usage scenarios of a lock-free queue to a single producer and a single consumer, we can further optimize

[2] `https://github.com/JCTools/JCTools/blob/master/jctools-core/src/main/java/org/jctools/queues/SpscArrayQueue.java`



(a) A `null` value in the slot of the head of the queue indicates that the queue is currently not full.



(b) A non-`null` value in the slot of the head of the queue indicates that the queue is currently full.

Figure 3.2. A producer and a consumer accessing an array-based ring buffer queue in clock-wise order.

its performance. The main idea is to minimize the number of shared variables and instead use already synchronized data elements also as control variables. For example, such a data/control coupling can be realized by avoiding the synchronization of the write and the read indices to detect a full or an empty queue. Instead, `null` in an array slot can be used as special value to represent the state *ready-to-add*. Thus, the producer may only add an element to the slot which represents the head of the queue, if this slot contains the special value `null`. Otherwise, the queue is full and the producer has to wait for an empty slot or skip the insertion. Figure 3.2 illustrates both situations.

The consequence of this approach is that we may not add `null` as element anymore. For this reason, a corresponding `null` check is performed before adding the element to the queue. Listing 3.2 shows an example implementation in Java.

Cache Misses due to False Sharing When the producer and the consumer access array slots which are located closely together in memory, these slots

```
1  class LockFreeSpScQueue<E> {
2    boolean addToHead(E element) {
3      if (null == element) { return false; }
4      if (null != underlyingArray[headIndex]) { return false; }
5      // adds the element to the head
6      underlyingArray[headIndex] = element;
7      headIndex = next(headIndex);
8      return true;
9    }
10
11   E removeFromTail() {
12     // removes the element from the tail
13     E element = underlyingArray[tailIndex];
14     if (null == element) { return null; }
15     // marks the slot as empty
16     underlyingArray[tailIndex] = null;
17     tailIndex = next(tailIndex)
18     return element;
19   }
20 }
```

Listing 3.2. A lock-free queue implementation.

are loaded into the same cache line. As soon as one of the accessors updates its slot, it invalidates the associated cache line such that the other accessor must reload the cache line. In the worst case, producer and consumer always compete with each other for the same cache line. Although they do not necessarily share the same slot, they do share the same cache line, albeit accidentally. Cache misses that result from this situation are called false sharing misses [TLH94].

Temporal slipping is an approach [GMV08] which avoids false sharing. It ensures that the current slots of both queue accessors are far enough away from each other such that their slots are not located on the same cache line. For this purpose, the approach measures the distance between these

two slots by comparing the head and the tail index. If the distance is below a given threshold, e.g., a multiple of the cache line size, the approaching accessor (whether producer or consumer) waits for a particular moment in time. In this way, the approach guarantees no false sharing.

However, for measuring the distance, both accessors need to be able to read the current value of the head and the tail index. For example, whenever the producer needs to additionally read the tail index for computing the distance, it checks whether the tail index in its cache is up-to-date. In a high-throughput scenario, this is usually not the case because the consumer updates the tail index very often. Thus, the producer frequently experiences cache misses. In summary, the more frequently the adjustment routine is applied, the more frequently occur cache misses. Hence, this adjustment routine should not be applied too often, e.g., not on each queue access.

Padding is another approach to avoid false sharing. This technique allocates additional memory between the head and the tail index such that they are not placed on the same cache line. In this way, an update of the head index by the producer does not trigger a cache invalidation for the tail index by the consumer.

Backtracking Wang et al. [WZT+13] present an approach to reduce the number of synchronized `null` checks in the array slots when using data/-control coupling (see Section 3.5 on Page 33). The idea is to let the consumer check for the `null` value in a slot *s** some positions ahead of the tail slot. If *s** is not `null`, all of the slots in between are also not `null`. Thus, elements can be consumed until *s** is reached without further synchronized `null` checks on each slot access. Listing 3.3 shows an example implementation of this approach, but for the producer. For this reason, the implementation uses forward tracking instead of backtracking. Nevertheless, the idea is the same.

Summary With respect to a P&F framework, lock-free queues are suitable in most cases. However, if a stage receives input with a low or irregular frequency, a blocking queue could be the more appropriate choice. Similarly,

```
1 class LockFreeQueue<E> {
2   boolean addToHead(E element) {
3     if (this.producerIndex >= this.producerMaxBatchIndex) {
4       if (!computeNewProducerMaxBatchIndex()) {
5         if (null != this.array[this.producerIndex]) {
6           return false;
7         }
8       }
9     }
10
11    // adds the element to the head
12    this.array[this.producerIndex] = element;
13    this.producerIndex = next(this.producerIndex);
14    return true;
15  }
16
17  E removeFromTail() {
18    // remove element from tail
19  }
20 }
```

Listing 3.3. A lock-free queue implementation based on the B-Queue.

most of the queue optimization techniques are useful for P&F architectures. However, some of them, like temporal slipping and other ones described by Wang et al. [WZT+13], are discussed controversially and require more research. In Part II, we describe why and how we apply lock-free Sp/Sc and Mp/Mc queues for our synchronized pipe implementations.

Chapter 4

Design Patterns for Parallel Systems

A design pattern for parallel systems describes a good solution to a recurring problem in the parallel software domain. Similar definitions for design patterns in general and in particular can be found in [GHJ+95; BMR+96; Ort10]. For an overview of such patterns, we refer to [Ort10]. Beyond that, Mattson et al. [MSM04] propose a generic approach to build up parallel systems based on design patterns. In this thesis, we use the Pipeline pattern and the Task Farm parallelization pattern to increase the throughput of P&F-based applications. In Section 4.1 and Section 4.2, we describe both patterns in more detail.

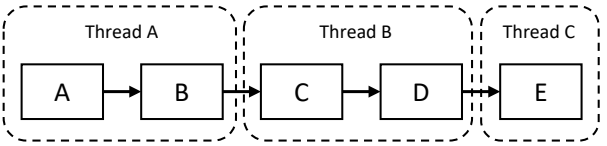
4.1 The Pipeline Pattern

In 1913, Ford presented the world's first moving assembly line[1] which revolutionized the mass production of automobiles. The general idea of an assembly line is to break down the assembly process into a sequence of operations such that the individual steps can be executed simultaneously. Each operation is performed by a dedicated worker on the parts passing from workstation to workstation. This manufacturing process is a good analogy for the pipeline pattern which we briefly describe in the following. Note that we focus on the pattern's capabilities for a parallel execution. For a more detailed description, we refer to [MSM04].

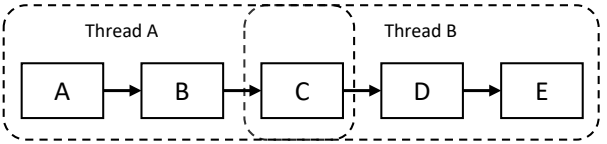
[1] https://www.assemblymag.com/articles/91581-the-moving-assembly-line-turns-100

Context & Problem We intend to perform a sequence of calculation steps on a stream of data elements as fast as possible. For this purpose, we intend to execute these steps in parallel. First, we need to consider how many threads should execute the calculation steps. Second, we need to consider which threads should execute which steps. If two threads share the execution of a step, synchronization must be applied accordingly. Moreover, if the order is important, these two threads must further agree on that order.

Solution The pipeline pattern provides a solution to the problem by assigning the calculations steps to distinct, executable stages which are connected with each other in a unidirectional way. This pattern can thus be seen as a specialization of the more general Pipe-and-Filter style. Depending on the chosen execution strategy, each stage can potentially be run by either all of the given threads or by a dedicated one. The former approach can be realized by a synchronized work queue and a thread pool such that an idle thread takes and executes a stage from that queue (see Section 2.8 for more details). The latter approach, instead, assigns each thread a distinct subset of the available stages for execution. A synchronized work queue and inter-thread coordination are not necessary. As already described in Section 2.7, a prominent, but naïve form of this approach is to execute each stage in a dedicated thread. Alternative, more sophisticated approaches allow to define the number of threads independent of the number of stages. If the number of threads in these approaches is smaller than the number of stages, some threads must execute more than one stage (see Figure 4.1a). If the number of threads is higher, some threads must share the execution of a particular stage (see Figure 4.1b). If such a stage is stateful and can be executed in parallel, the state variables must be properly synchronized, e.g., by a lock or an atomic CPU instruction like compare-and-swap. To preserve a particular order on the output elements, we could forbid the execution of a stage by more than one thread at any point in time. Alternatively, we could maintain ordering in the presence of a parallel execution if the stage has a read-only state or no state at all. In these cases, all of the participant threads take a ticket to consume a contiguous sequence of incoming elements one after the other. The resulting output elements are then sent to the output ports in the same order. In Section 9.1, we discuss the pool approach and



(a) Thread A and B execute two stages each.



(b) Two threads share the execution of stage C.

the assignment approach in more detail with respect to the Pipe-and-Filter style.

4.2 The Task Farm Parallelization Pattern

In 1991, Cole [Col91] introduced so-called *Algorithmic Skeletons*. Such a skeleton is defined as a higher-order function describing a certain computational behavior. One example is the Task Farm parallelization pattern which has been defined by, e.g., Aldinucci and Danelutto [AD99]. Semantically, it describes the identity function of its underlying algorithm. However, the input data stream is parallelized such that the underlying algorithms is concurrently executed. In [ADK+14], the task farm pattern was, among other patterns, implemented for *FastFlow*, a framework for streaming applications where stages have exactly one input port and one output port [FastFlow]. In the following, we briefly describe the task farm parallelization pattern. For a more detailed description, we refer to [AD99] and [MSM04].

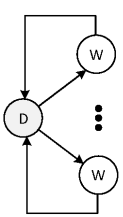
Context & Problem We intend to perform an operation f on each task of a given stream of tasks as fast as possible. For this purpose, we intend to execute f on multiple tasks in a concurrent way. First, a task distributor

is required to distribute tasks to worker processes. Depending on the scenario, different distribution strategies are possible, e.g., round-robin or broad-casting. Second, it is important to determine how many worker processes should run concurrently. Hence, we need to find a balanced ratio between the computation cost of f and the communication cost between the distributor and the workers. Finally, we need to consider what we should do with the results of the worker processes. For example, the results could be fed back to the distributor or merged for later processing.

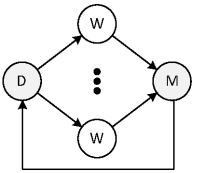
Solution In order to support arbitrary scenarios, the task farm parallelization pattern provides a generic task distributor which can be declared active or passive. In the former case, it autonomously distributes incoming tasks to one or more worker processes according to a user-defined distribution strategy. In the latter case, it serves as a task pool from which worker processes fetch tasks.

Figure 4.2 shows some well-known manifestations of the task farm pattern. For example, Figure 4.2a illustrates a version which is also called the master/worker pattern. Here, the distributor is responsible for both the distribution of the tasks to the workers and the collection of the result from the workers. Figure 4.2b illustrates a more modular version where the collection of results is extracted from the distributor to a so-called merger. Optionally, the merger can still feedback its result to the distributor. Figure 4.2c shows an example for a composite worker. Here, the workers are again composed of a task farm. Similarly, Figure 4.2d shows a another composite worker which is composed of a worker pipeline.

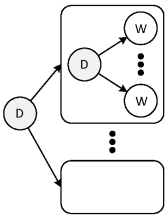
Typically, the number of worker processes is fixed and given in an initialization phase (e.g., see FastFlow [FastFlow]). However, the task farm pattern does not forbid to adapt the number of worker processes at runtime. Cloud environments, e.g., use the master-worker version to scale their instances according to the workload. In Section 9.2.4, we describe how we combine a self-adaptation manager and the task farm pattern to allow a self-adaptive behavior of P&F-based applications.



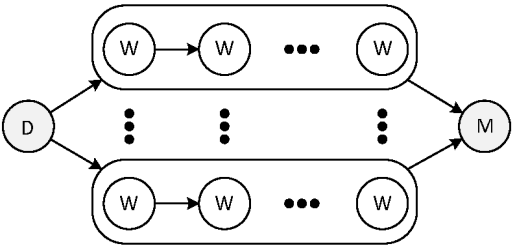
(a) Farm as master-worker.



(b) Farm with a merger and a feedback loop.



(c) Farm composed of a farm per process.



(d) Farm composed of a pipeline per process.

Figure 4.2. Different manifestations of the task farm pattern. (D) means distributor, (W) means worker, and (M) means merger.

Chapter 5

The Actor Model

The actor model [HBS73; Agh86; KSA09] is a model of concurrent computation. In 1973, it was originally designed to model and to experiment with highly parallel computing machines of tens to thousands of microprocessors. Nowadays, the problem of programming scalable multi-core processors has renewed the interest in the actor model. We use the actor model within TeeTime to realize distributed P&F architectures. Please refer to Section 8.2.3 on Page 70 for more information on our realization.

In the following, we briefly introduce the basic concepts of the actor model (in Section 5.1), describe the resulting properties (in Section 5.2), and name some prominent example implementations (in Section 5.3).

5.1 Components

The central units of computation are so-called *actors*. They are concurrent, autonomous entities which communicate with each other via asynchronous message passing. Each actor can send messages to other actors, create further actors, and perform tasks which depend on the consumed messages and its current internal state.

To send a message to a target actor, the source actor needs to know the name of the target actor. The name serves as an abstract kind of address to where messages can be sent. Depending on the implementation, names of actors can be represented, e.g., by e-mail addresses, IP addresses, or process IDs. In the actor model, a message is always sent asynchronously such that the source actor does not need to wait for the message to be consumed. Moreover, the model ensures that each message is transmitted after a finite time. When the message finally arrives at the target actor, it is placed in

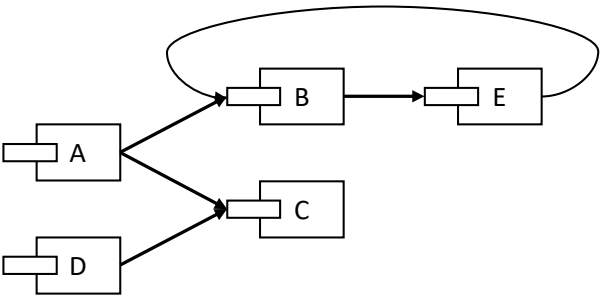


Figure 5.1. Possible communication ways between actors.

the so-called *mailbox* of the target actor. This mailbox serves as a buffer of incoming messages which the target actor consumes one after another in FIFO order. Figure 5.1 shows some possible communication ways between actors. Actor A sends messages to two different actors, actor C receives messages from two different actors (A & D), and actor B forms a cycle with actor E.

When an actor consumes a message, it determines the type of the message and reacts accordingly based on its internal implementation logics. For example, the actor can send new messages to itself or to other actors, as described above. However, a further valid reaction to a consumed message is to create new actors. Often, these new actors are then treated as child actors such that a termination of the parent actor triggers the termination of its child actors. Inversely, if a child actor terminates—either planned or unexpectedly—it notifies its parent actor such that the parent actor can react appropriately. Another possibility to react to a consumed message is to update internal state attributes. In this way, an actor can react differently to the same type of messages.

5.2 Properties

The actor model is characterized by the following properties: concurrent computations, fault tolerance, and hardware abstraction.

The definition of an actor naturally allows to model and to perform concurrent computations. Actors are declared independent of each other and are loosely coupled by exchanging messages. A message is sent asynchronously and is buffered in a synchronized mailbox. Hence, source actors and target actors can run concurrently.

Fault tolerance is supported by the parent-child relationship between actors. A parent actor supervises its children actors in order to react gracefully to a failure caused by one of its child actors. For example, the parent actor can restart a faulty child actor or try to resend a message to a child actor until a maximum number of retries. Most actor model implementations (see Section 5.3) provide many more fault tolerance strategies based on the parent-child relationship.

Hardware abstraction is provided by using logical names instead of hardware addresses to identify individual actors. Thus, messages are sent to named actors. The underling actor model implementation resolves the names to actual addresses in a transparent way. The application developer does not need to consider whether a target actor runs on the same node as the source actor or on a different node. Moreover, the resolution strategy is configurable such that names can be mapped to arbitrary addresses.

5.3 Example Implementations

There are many manifestations of the actor model these days. For example, the actor model is an integral part of the functional, concurrent, and distributed programming language *Erlang* [AVW+93; Arm13]. Each lightweight process is internally represented by an actor. Messages are exchanged by the native language constructs `!` (for send) and `receive`. Another example is the functional programming language *Scala* [CB14; Gul15]. It is also known for its sophisticated support for the actor model. Similar to Erlang, it provides the messaging operations `!` and `receive`. However, Scala's primary kind of processing units are threads, not actors. Hence, an actor is defined by declaring a class which extends the pre-defined class `Actor`.

Besides programming languages, the actor model is also provided as

frameworks. Some more prominent examples are Theron[1], Celluloid[2], and Akka [All13; RBW15; Wya13]. Theron is a C++ concurrency library which is based on the actor model. Celluloid is a library to build fault-tolerant concurrent programs in Ruby. Akka, as already mentioned in Section 1.1, is an implementation of the actor model on the JVM. It allows for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala. Hence, we choose Akka as framework for our actor-based realization of distributed P&F architectures.

[1] http://www.theron-library.com
[2] https://celluloid.io

Chapter 6

Domain-Specific Languages

As opposed to general purpose languages (GPLs), such as Java or C#, domain-specific languages (DSLs) are designed or optimized for a particular application domain [MHS05; Bet16]. For example, we use one DSL for specifying P&F architectures (see Section 10.2) and another one to enable the instrumentation for our visualization approach (see Section 10.3.1 on Page 118). In return for their specialized areas of application, they provide several advantages compared to GPLs. For example, the syntax of DSLs often has a more compact notation and, as a result, allow to write code in a faster and less error-prone way. Moreover, DSLs often provide a higher level of abstraction such that aspects irrelevant for the particular domain are hidden from the user. In this case, the DSLs handle such aspects in a predefined way.

In general, DSLs can be specified in a graphical or textual way. For example, the Unified Modeling Language (UML) is a graphical DSL whereas the Structured Query Language (SQL) is a textual one. All of the DSLs, which we have developed in the context of this thesis, are textual ones. Hence, we focus on this kind of DSLs. In addition, DSLs are categorized into internal or external ones [Fow10]. An internal DSL is embedded into a GPL often in the form of a library with a fluent interface. As opposed to that, an external DSL defines its own language constructs and therefore requires a dedicated parser. For example, we provide an internal and an external version of our Configuration DSL (see Section 10.2). The internal version is realized as a Java library on top of TeeTime such that it can be used without additional tooling and knowledge about another language. Instead, the external version represents a new, independent language and thus requires a dedicated compiler to integrate with TeeTime. However, it allows to define

P&F architectures in a more compact and abstract form. In this way, it is independent of any specific programming language constructs.

For our DSLs, we use Xtext [Bet16], a framework for the development of programming languages and domain-specific languages. We define a grammar for each DSL and Xtext generates an associated full infrastructure including a parser, a linker, a typechecker, and a compiler as well as editing support for Eclipse. Since Xtext is based on the Xbase framework [EEK+12], it also provides the Java type system and some predefined data types like integer, float, and string. In this way, DSL development is simplified enormously.

Part II

The Pipe-and-Filter Framework TeeTime

Chapter 7

Research Design

In this chapter, we present our research design for targeting the problems concerning the modeling and the execution of P&F architectures described in Section 1.1. We begin in Section 7.1 by presenting our research scope including our research goal. Afterwards, we describe how we realize our research goal by presenting our research plan in Section 7.2. Finally, in Section 7.3, we summarize the research methods which we apply in the course of our research.

7.1 Research Scope

The scope of our research comprises the development and the execution of P&F-based applications. As motivated in Chapter 1, we see the necessity and the potential to ease the engineering and to improve the performance of such applications. Hence, our research goal is to provide a new P&F framework which is generic and parallel. Below, we describe both properties in more detail.

With a generic P&F framework, we address a broad applicability. The framework must support the modeling and the execution of not only a particular kind of P&F architectures, but of all possible variations. Moreover, it must target practitioners as well as researchers in order to provide a usage and an experimentation platform for the P&F style. This requirement in turn depends on the framework's extensibility. Hence, it must also provide well-defined interfaces which allow to extend or to replace available functionality. Additionally, we address reusability in the sense such that predefined filters can be used in several, different fields of applications. Finally, we address language independence so that the framework can be implemented and

used not only in one, but in several object-oriented programming languages.

With a parallel P&F framework, we address the parallel execution of filters. We will go beyond the naïve approach which executes each filter in a dedicated thread. In addition, the framework must execute these filters in an efficient way, especially in scenarios where the workload is not evenly distributed among the filters. Hence, we address a high throughput for various kinds of parallel P&F configurations. Nevertheless, the framework must also execute sequential and distributed P&F configurations in an efficient way.

7.2 Research Plan

As mentioned in Section 7.1, our research goal is to provide a new generic and parallel P&F framework to enhance the development and the execution of P&F-based applications. With that goal in mind, we define the following *Goal, Question, Metric*-based [SB99; SBC+02] research plan:

Q1: How to Model Arbitrary P&F Architectures?

To answer this question, we will construct different, representative P&F architectures. These representatives must cover both, various technical and various domain-specific aspects. For example, they have to represent linear, branched, and loop-based P&F instances. Moreover, they should represent different use cases. By implementing these P&F architectures with our envisioned framework, we will show its generic character and its broad applicability. We refer to Chapter 12 for the corresponding evaluation. In addition to that, we will show that the capability to model arbitrary P&F architectures does not depend on a single, particular implementation of our framework. We refer to Chapter 11 in which we present two implementations of the framework written in Java and in C++, respectively. Our associated metrics are therefore:

- **M1.1** Number and kind of different, representative P&F architectures.
- **M1.2** Number of implementations in different programming languages.

Q2: How to Provide Reusability and Extensibility by the Framework?

To answer this question, we will identify which entities of the frameworks have to be fix, extensible, and exchangeable. Primarily, the required grade of changeability will decide on whether a particular entity should be fully exchangeable or whether it is sufficient to only provide an extensibility mechanism. We will evaluate this grade by modeling different P&F architectures with different points of view from both research and industry. Thereby, we will also consider which entities should be encapsulated by the framework and which entities should be accessible for reuse. For the resulting framework architecture, we refer to Chapter 8. For a reusability and extensibility evaluation, we refer to Chapter 14. Our associated metrics are therefore:

- **M2.1** Number of fixed, extensible, and exchangeable framework entities.
- **M2.2** Number and kind of public and private entities of the framework.

Q3: How to Execute Arbitrary P&F Architectures?

To answer this question, we will consider different, representative P&F architectures similar to how we do so for answering **Q1**. For example, multiple ports per stage and a graceful termination may not be a problem. Moreover, such representatives must be executable with one and with multiple threads as well as in a distributed setting. We refer to Chapter 12 for a corresponding feasibility evaluation with several, different case studies. Our associated metrics are therefore:

- **M3.1** Number and kind of different, representative P&F architectures.
- **M3.2** Number of threads and nodes.

Q4: How to Efficiently Parallelize Arbitrary P&F Architectures?

To answer this question, we will evaluate different existing parallelization approaches for the P&F style. In particular, we will consider and evaluate approaches which parallelize single filters. Furthermore, we will develop

and evaluate a new efficient, parallel scheduler for the execution of filters on multi-core systems. Finally, we will improve the profiling of parallel P&F configurations by providing tools that support in identifying bottlenecks and concurrency issues. We refer to Chapter 13 and Chapter 15 for the corresponding evaluations. Our associated metrics are therefore:

- **M4.1** Number and kind of different parallelization approaches.
- **M4.2** Number and kind of different parallel scheduling algorithms.
- **M4.3** Execution time and speedup.

7.3 Research Methods

Besides the Goal, Question, Metric approach [SB99; SBC+02], we employ the following, additional research methods to answer the research questions mentioned in Section 7.2. These research methods follow the general research methods in software engineering by Wohlin et al. [WRH+12] and Juristo and Moreno [JM10] as well as the guidelines for empirical software engineering by Kitchenham et al. [KPP+02].

- Literature Review: We conduct literature reviews to identify and to compare existing approaches and concepts of other researchers.
- Proof-of-Concept Implementation: We develop proof-of-concept implementations to evaluate the technical feasibility of single framework entities and framework concepts. Simultaneously, they are a prerequisite for our lab experiments.
- Lab Experiment: We conduct lab experiments to investigate and to compare the performance of multiple different implementations of the same entity, for example, of pipes and schedulers.
- Case Study: We conduct case studies to show the generic character and the broad applicability of TeeTime in real contexts.

Chapter 8

Framework Architecture

As discussed in Chapter 2, there are still some open challenges for the P&F architectural style in general and for a P&F framework in particular. In Chapter 7, we described that we address these challenges by providing a generic and parallel P&F framework called TeeTime. In this chapter, we introduce TeeTime by presenting its software architecture. First, in Section 8.1, we describe our main architectural drivers which have heavily influenced the framework design. Afterwards, in Section 8.2, we present each of the framework's entities, including its responsibilities and its design decisions. In the following, we will use the term *stage* as generalization for *data sources*, *filters*, and *data sinks*, as categorized by Buschmann et al. [BMR+96].

In this chapter, we introduce the structural concepts of our software architecture. For a description of TeeTime's execution concepts, we refer to Chapter 9. Moreover, we refer to Chapter 10, where we show several different ways to define and to debug stages as well as configurations with TeeTime. Finally, we refer to Chapter 11 for a description and an evaluation of our two proof-of-concept implementations in Java and in C++.

8.1 Architectural Drivers

Our architectural drivers incorporate those requirements and constraints which focus on a broad applicability, on a high throughput, and on a great level of extensibility and reusability. In the following, we describe each of our major architectural drivers in more detail.

Requirement (REQ_{all}): Modeling and Execution of all P&F Variants We successfully used the P&F style in research and industrial projects (see Part III for some examples). Each of these projects targets a different domain of application and thus requires a different kind of P&F architecture. Hence, we expect from our P&F framework that is able to model and to execute all kinds of P&F architectures. As described in Chapter 2, these kinds include simple linear pipelines, more complex branched structures, and also feedback loops.

Supporting all of these P&F architectures is one of the major lacks which current frameworks, such as StreamIt [TKA02], FastFlow [ADK+14], and GRAMSP [SFB+09] have.

Requirement (REQ_{thrpt}): High-Throughput Execution One of the main advantages of the P&F style is its potential for a high-throughput processing. For example, while the first stage in a P&F configuration processes the next incoming data element, the second stage can work on the output of the first stage. This throughput-oriented processing requires both a parallel stages execution and an efficient stage scheduling.

Especially with regard to the expected increase of cores per processors in the near future [Bor07; DL16], a single cache-aware multi-core system provides an immense potential for a parallel execution of stages. Hence, in order to be able to utilize the capacity of current commodity hardware, we expect from our P&F framework that it is able to execute stages in parallel. Nevertheless, the framework must still be able to execute a given P&F configuration by a single thread only. In addition, it should allow for a distributed execution. However, we do not focus on these two issues (see Constraint CON_{mcore}).

Concerning an efficient scheduling, the framework should ideally schedule a stage for execution only if this stage will make progress, i.e., if it consumes some input, changes its internal state, or produces some output. Moreover, the scheduler and its associated threads should communicate with each other in an efficient way, for example, by causing as few synchronization overhead as possible.

We use Kieker's internal P&F framework [HWH12] for analyzing monitoring data. Although it supports parallel execution, its scheduler as well as

unnecessarily coarse-granular synchronization points effectively sequentialize the execution.

Requirement (REQ_{ext}): Extensible Entities As mentioned in Chapter 2, there are multiple different ways to schedule stages in a multi-threaded context. Suleman et al. [SQK+10] propose to use a global scheduler, Aldinucci et al. [ADK+14] implemented a local scheduler for each thread in their tool, and Sanchez et al. [SLY+11] suggest to use a hybrid approach. So far, it seems that an efficient scheduling depends on the scenario. In short, finding an efficient scheduling is still an active research topic. Hence, a P&F framework must be able to integrate and to apply new scheduling approaches. In general, the framework has to be open for extensions such that it supersedes writing another, custom P&F framework. In this way, it can effectively reduce the implementation effort. In FastFlow [ADK+14; FastFlow], for instance, it is not possible to exchange the scheduling or the pipe implementations without major effort.

Requirement (REQ_{reuse}): Reusable Entities Another main advantage of the P&F style is its potential for a high modularity. Stages do not directly depend on each other. Instead, they communicate only via their ports and potentially via a shared state. This modularity allows to reuse stages by others and in many different scenarios. Moreover, pipe implementations and schedulers should also be reusable in order to provide a flexible experimentation platform. Hence, it is essential for a P&F framework to support reusability at several different layers.

In particular, it must support not only primitive, but also composite stages. Such stages provide a higher level of abstraction and thus improve reusability. A P&F user does not need to know which stages to declare and how to connect them properly. Instead, he or she can just use a corresponding predefined composite stage. One major challenge of composite stages, however, is to execute them in an efficient way—preferably without any runtime overhead. Kieker's internal P&F framework [HWH12], for instance, does not support composite stages at all.

Another aspect of reusability is to provide the framework as a whole for reuse by others. Although there are free web platforms for providing open

source software, like sourceforge and github, they are still rarely used in research. For example, the research tools presented and evaluated by [Bor07; SFB+09; SQK+10; SLY+11] are not provided officially for download.

Requirement (REQ_{tsafe}): Type-Safe Connection of Stages Besides the ability to model arbitrary P&F architectures, a P&F framework should also support the modeling process itself. With this requirement, we focus on the proper connection of stages in a P&F configuration. A P&F framework should automatically detect and report a connection whose source port and target port are not compatible. For example, if the source port sends string values, but the target port accepts integer values only, the framework should ideally recognize this invalid connection at compile-time. In cases where compile-time checking is not available or not possible, the framework must at least validate each connection at run-time. For example, run-time validation is necessary if the used programming language does not provide support for compile-time checking or if a P&F configuration is built dynamically via reflection. Hence, the earlier a P&F framework recognizes an invalid connection between stages, the lower is the implementation effort. Thereby, the user experience and the acceptance increases. Since the Akka framework [Lig] follows the actor model, it naturally cannot ensure a valid connection between actors at compile-time. Even worse, the run-time type check of each incoming message needs to be programmed manually by the Akka user. The framework does not provide support for automatic type checking. Similarly, Kieker's internal P&F framework [HWH12] is also not able to ensure type-safety at compile-time, although it does so at least at run-time.

Constraint (CON_{oo}): Object-Oriented Framework Architecture Due to the broad acceptance of object-oriented programming languages and the intrinsic object-based structure of the P&F style, we focus on an object-oriented framework architecture. This allows us to reason about a framework architecture in terms of architectural entities with a state, a behavior, and a relationship to other such entities. Moreover, we can assume the existence of type parameters, interfaces, and concepts like data encapsulation and inheritance.

Constraint (CON_{mcore}): Multi-Core Systems In our opinion, too little attention is paid to the parallel execution on multi-core systems. Instead, we see a huge trend to distribute computations among multiple nodes. The range of corresponding matured open source tools, such as Kafka [Foua], Apache Spark [ZXW+16; Foub], RabbitMQ [Piv], Akka [Lig], Apache Storm [Fouc], and Kubernetes [Foud], confirm our observation. Furthermore, distributed architectures, such as REST, the cloud, and micro services, gain more and more popularity. Nevertheless, there is still much work to do concerning an efficient parallel execution on a single node. As mentioned above, the number of cores per node will continue to increase such that we need to find ways to exploit such immense potential of processing power. A single multi-core system can achieve a throughput that is much higher than the maximum bandwidth between two nodes. In particular, the execution of P&F-based applications on such systems has not yet been discussed and evaluated sufficiently. Hence, our focus is on providing support for an efficient execution and for a research platform on multi-core systems.

8.2 Framework Entities

Our proposed framework architecture follows the original definition by Shaw [Sha89] and Allen and Garlan [AG92; AG94]. It incorporates all of the four first-class entities: pipes, stages, ports, and configurations. This equally applies to our Java and C++ implementations. We describe them in more detail in Chapter 11. Besides these basic entities, our framework architecture includes some more, essential entities which are derived from our architectural drivers mentioned in Section 8.1. In the following, we describe each entity in a structured way: we give them a name, describe their responsibilities, and justify their design by explaining our design decisions. Figure 8.1 gives an overview of our major entities.

8.2.1 Entity: Stage

A stage represents a certain kind of computation. For this purpose, the stage may declare an arbitrary number of input ports, i.e., zero, one, or

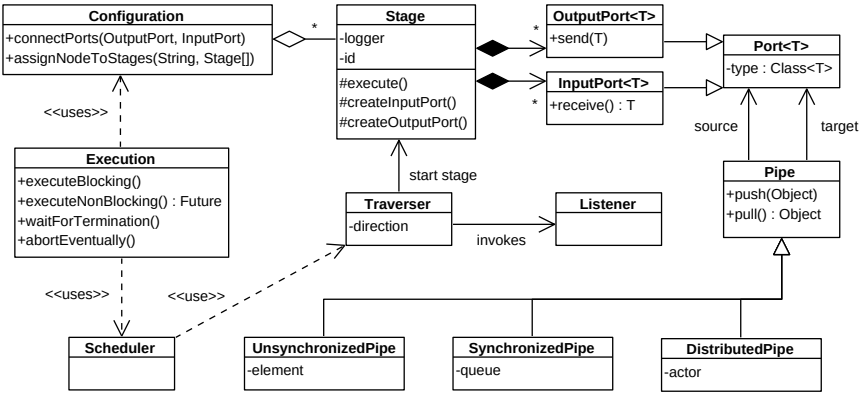


Figure 8.1. An excerpt of the TeeTime framework architecture showing its four first-class entities *configuration*, *stage*, *port*, and *pipe* and their relationships.

more. If it is not stateless, it is stateful exactly as defined by Shaw [Sha89] and Allen and Garlan [AG92; AG94]. This means, the stage can compute its output based on its input only, or additionally based on internal state variables. However, the stage in general and its computation in particular may not depend on any other stage. Beyond that, the stage can also have state variables which are shared among other stages. Note that this is not intended by the original P&F style. However, as described in Section 2.5, it can improve the modularization and the performance of the whole P&F architecture. While executing, the stage itself is responsible for which input ports it reads in what frequency and in what order.

Similar to its input ports, a stage can declare an arbitrary number of output ports. An execution of the stage may, therefore, lead to some output at its output ports. Hence, our framework architecture allows to model data sources (without any input port), filters (with both input and output ports), and data sinks (without any output port). Note that it is up to the stage whether it sends its result to none, one, or multiple of its output ports. Beyond that, an execution can also cause a change in its state if the state is not read-only. If the stage is a data source, i.e., if it has no input ports, it must additionally declare a termination condition. In this way, it

can signal the scheduler when it has finished its work and thus should not be rescheduled anymore. Furthermore, a stage may add and remove its input and output ports while being executed. Thus, the number of ports is generally not fixed. One use case is shown in Section 9.2, where we dynamically adapt the number of ports (and also of stages) in order to react on workload which changes during runtime.

To distinguish two stage instances of the same type, each instance has a unique identifier. In addition, each stage instance has a unique logger which can be turned on or off individually. In this way, we enable a focused debugging of the runtime behavior. Furthermore, both attributes are used if an error occurs within the execution. The stage's identifier is passed to an exception handler which in turn may use the stage's logger to report the error, e.g., to the console.

At runtime, a stage is either active or passive depending on whether it is executed by a dedicated thread or not. However, this *activity state* is not defined in the stage, but in a P&F configuration where the stage is declared. It depends on the particular configuration and on the used scheduler whether a stage should be active or passive. Thus, to employ a stage in an arbitrary context, its definition cannot prescribe its activity state.

For the same reason, a stage should not contain any synchronization mechanisms. It depends on the used scheduler whether synchronization is necessary or not. For example, if two stages, which are connected with each other, are always executed by the same thread, synchronization is not necessary at all. Thus, to execute a stage in the most efficient manner, its definition should not use any synchronization.

A stage may be primitive or composite. If it is composite, it is composed of predefined child stages. Each of these child stages can in turn be a composite stage possibly forming various levels of hierarchy. However, a composite stage is only a wrapper and does not exist at runtime. It is flattened to the P&F configuration which is represented by its child stages. That means, it does not have a dedicated execution logic and cannot be declared active. In this way, we keep its design simple leading to a minimized potential for programming errors. Simultaneously, we thus ensure that a composite stage does not incur *any* runtime overhead.

Conversely, if a composite stage could be declared active, we would

observe a reduced performance. Consider, for instance, the following two possible implementations:

1. The first implementation executes the composite stage in a dedicated thread such that each incoming data element is delegated to one of the child stage. This delegation process must be synchronized if the child stage is active, too. However, synchronization always comes with a loss of performance and thus causes a potentially significant runtime overhead.

2. The second implementation does not execute the composite stage in a dedicated thread. Instead, all of its child stages are declared active. This implementation could lead to an undesired high amount of active stages and thus decrease the overall performance due to excessive contention.

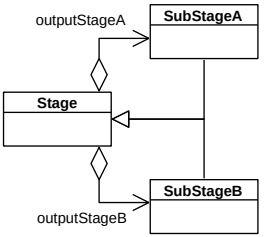
In sum, we disallow a composite stage to be declared active.

In Section 13.1.3, we show in a performance experiment for various levels of hierarchy that a TeeTime-based composite stage comprising a set of stages has the same throughput as the set of stages in isolation.

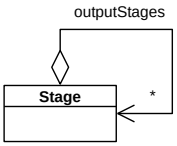
8.2.2 Entity: Port

A port represents an entry point to or an exit point from a stage. Such an input port or, respectively, output port provides a method to read from or, respectively, to write to another port. In this sense, a port is a part of the interface of the stage to which the port belongs. Thus, it is responsible for decoupling its stage from other stages.

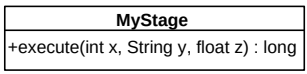
Conversely, if a stage would directly use stages instead of ports, the stage would commit itself to particular types of stages (see Figure 8.2a) or use the abstract super type of stages (see Figure 8.2b). Either way, both approaches require that the stage directly invokes its successor stages to pass output elements. Thus, a stage would also take on the role of a scheduler. If a stage would use pipes instead of ports, the stage could use the whole interface of a pipe. In particular, it could push and pull elements to and, respectively, from each pipe independent of whether the pipe represents an entry or an exit point. Hence, this approach bears the risk of an improper use by the framework users. If a stage would use the parameters and the return



(a) Using particular types of stages instead of ports



(b) Using the super type of stages instead of ports



(c) Using the `execute`'s method signature instead of ports

Figure 8.2. Alternatives to using ports in order to connect stages with each other.

value of its `execute` method instead of ports, the stage could have not more than one single output target (see Figure 8.2c). Multiple output targets in general and branches in particular are not possible with this approach. For these reasons, we decided to incorporate ports as first-class entity into our framework architecture.

Besides stage decoupling, a port is also responsible for a type-safe connection between stages. Its type enables a type-safety analysis at compile-time and at run-time. For this purpose, we model the type of a port in the following two ways. For compile-time validation, we attach a type parameter on the port. If two ports are incompatible with each other, the compiler detects this issue and outputs an appropriate error message. Since type parameters are removed in the compilation process (at least in Java), we need a different approach to access the type of a port at run-time. For this reason, we add a `type` attribute to the port. Unlike type parameters, attributes are accessible at run-time in all object-oriented programming languages. Within the validation phase, TeeTime is thus able to detect incompatible port connections.

8.2.3 Entity: Pipe

A pipe represents a unidirectional connection between an output port and an input port. It provides two methods which allow to push and to pull data

elements into and, respectively, from the pipe. Hence, a pipe is responsible for transmitting data elements using a particular transport medium and a transport format. Depending on the format, the pipe needs to perform some kind of data conversion, if necessary. In such a case, it serializes incoming elements into the transport format and deserializes outgoing elements into the target format. Moreover, a pipe is responsible for buffering incoming data elements. Buffering can help in situations where the amount of pushed elements is *temporarily* greater than the amount of pulled elements. Depending on the pipe implementation, such a buffer is represented by a single ordinary value or by a more complex data structure, such as a queue. Furthermore, a pipe is responsible for the scheduling of its source stage and its target stage. It either directly executes the target stage or it returns to the source stage. For performance reasons, the pipe by default does not notify the scheduler upon each new incoming and outgoing element. Instead, the scheduler reads the state of the pipes in a regular time interval or when it needs to assign a free thread to a stage. It depends on the scheduler strategy how the scheduler behaves (see Section 8.2.7 for more information). Finally, a pipe is responsible for the synchronization between its ports. Similar to buffering, the actual synchronization depends on the pipe implementation. Currently, TeeTime provides three different implementations for the three usage scenarios: unsynchronized communication, synchronized communication, and distributed communication. In the following, we describe the concept and the area of application of each implementation in detail. However, the user does not manually need to choose the implementation for each pipe between two ports. Instead, the scheduler undertakes this task by selecting the most appropriate implementation automatically (see Section 8.2.7 for more information).

Unsynchronized Communication

The pipe for unsynchronized communication between a source stage and a target stage consists of a buffer with a capacity of one single element. Figure 8.3a shows a schematic overview of the pipe. This small capacity is sufficient because temporal fluctuations in transmission cannot occur in single-threaded scenarios. Hence, this pipe does not need to buffer elements.

In TeeTime, we implemented this buffer by a simple field declaration. When the source stage pushes a new element to the pipe, the element is stored in this buffer. Afterwards, the pipe directly executes the target stage which in turn pulls the element from the pipe. Thus, this pipe implements the back-pressure policy: an element is first pushed through the whole P&F architecture until the next element is processed.

Concerning data conversion, this pipe does not serialize incoming elements. As opposed to inter-process and inter-node communication, data conversion is not necessary within the same thread. The pipe simply transfers all of the incoming elements by passing not their actual data, but their corresponding pointers (also known as *references* in object-oriented programming languages). In this way, we avoid an expensive overhead due to data conversion. Since both the source stage and the target stage share the same thread, communication does not need to be synchronized. Hence, this pipe completely avoids synchronization. Moreover, the pure use of direct method calls without any synchronization mechanisms allows modern just-in-time compilers, as for Java or C#, to perform sophisticated inlining operations automatically at runtime. By collecting and using online profile information [Orac], these compilers specifically increase the performance of those method calls which are invoked most frequently. In fact, we observed this behavior when executing several P&F applications with our framework TeeTime. Our evaluation in Chapter 13 shows that TeeTime imposes a negligible runtime overhead compared to a pure Java-based implementation without explicit pipes.

In summary, the unsynchronized pipe avoids synchronization and data conversion, and enables strong compiler optimizations which both contribute to a high throughput. We use this pipe implementation in combination with the scheduler described in Section 9.1.1.

Synchronized Communication

The pipe for synchronized communication between a source stage and a target stage consists of a buffer with a customizable capacity. Figure 8.3b shows a schematic overview of the pipe. In TeeTime, we implemented this buffer by a synchronized, lock-free queue. For more information on the

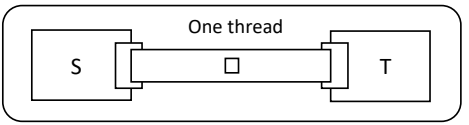
Java and C++ implementation, we refer to Chapter 11. Similar to the unsynchronized pipe, any incoming element is stored into the buffer. However, the pipe does not execute the target stage, but returns control to the source stage. Hence, this pipe does not implement the back-pressure policy. It is up to the scheduler when the target stage should be scheduled and, especially, by which thread. Concerning data conversion, this pipe does not serialize incoming elements either. As with the unsynchronized pipe, the synchronized pipe stores pointers in its buffer. Thus, data conversion is superfluous. As opposed to the unsynchronized pipe, this time, synchronization is necessary since communication extends across two different threads. As mentioned above, the pipe uses a lock-free queue for this purpose. In this way, it reduces its synchronization overhead to a minimum (see Chapter 3 for more information). In Section 13.1, we show by a comparative evaluation of various synchronized pipe implementations that TeeTime's synchronized pipe implementation achieves by far the highest throughput. We use it in combination with the schedulers described in Section 9.1.1 and Section 9.1.2. Moreover, we employ an instrumented version of it in our task farm stage to monitor its throughput. We refer to Section 9.2 for more information on this runtime monitoring extension.

Distributed Communication

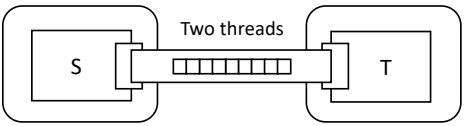
The pipe for distributed communication between a source stage and a target stage consists of two actors which communicate with each other either by TCP or UDP. Figure 8.3c shows a schematic overview of the pipe. Since both actors run on different nodes, they are separated via the network. Hence, technically, the distributed pipe does not consist of a single pipe instance. Instead, there is a sender instance with an actor and a receiver instance with another actor which together form the logical pipe. However, the actor-based implementation and communication are completely transparent to the P&F user. The pipe hides all of its internals just like the pipes do for unsynchronized and synchronized communication. Type-safety is maintained as usual by the source and the target port, so there is also no problem with the untyped mailbox used for the communication between both actors. In TeeTime, we implemented the actors by using the

Akka [Lig] framework which follows the actor model [Agh86; Hew10]. We chose Akka since it provides efficient and secure network transmission, automatic discovery of actors in the network, and fault tolerance. In this way, we do not need to reinvent the wheel, but rather rely on matured open-source software. Since, after all, we mainly focus on generic modeling and parallel execution. For more information on the Java implementation, we refer to Chapter 11. Currently, our C++ implementation does not support distributed P&F architectures.

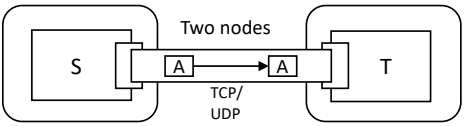
The pipe's behavior is as follows. When pushing elements to the distributed pipe on the sender node, they are stored in an internal buffer similar to the synchronized pipe. This buffer is shared with the associated actor such that the actor can concurrently pull elements from it and send them to the receiver actor. The receiver actor pushes incoming elements to the internal buffer of the associated pipe in order to let the target stage pull them from the pipe. Since communication takes place via the network, outgoing elements are serialized to a byte sequence of a particular format. Moreover, the byte sequence is wrapped by a message header with some meta data because the actor model is message-driven. We keep the corresponding data conversion overhead as small as possible by using, again, a matured open-source high-performance serialization/deserialization framework which is called Kryo [Sof]. For more information on the Java implementation, we refer to Chapter 11. As mentioned above, a stage and an actor communicate with each other by means of an internal buffer. This buffer is represented by the same lock-free queue which is also used by the synchronized pipe. Hence, we ensure a minimal synchronization overhead. Our design of the distributed pipe is based on a comparative evaluation which considers different frameworks for distributed processing and different P&F integration approaches. For a detailed description, we refer to the work of Echternkamp [Ech17]. Since the scheduler schedules stages and not pipes, it does not take the actual communication medium into account. Hence, there is no need for a special scheduler for distributed P&F architectures. Similar to the synchronized pipe, we use the distributed pipe implementation in combination with the schedulers described in Section 9.1.1 and Section 9.1.2.



(a) Intra-thread communication with an unsynchronized, single-element pipe.



(b) Inter-thread communication with a synchronized, bounded single-producer/single-consumer pipe.



(c) Distributed communication with an actor-based pipe.

Figure 8.3. Efficient pipe-based communication between stages in TeeTime.

8.2.4 Entity: Configuration

A configuration represents a set of stages which are interconnected by a set of pipes. Thus, such a configuration forms a P&F architecture which can be executed by TeeTime. The actual execution is handled by the *execution* entity described in Section 8.2.5. Since TeeTime supports the addition and removal of ports and associated stages while executing a given P&F configuration, note that the architecture may change at runtime. Basically, a configuration provides the method `connectPorts` which connects an output port with an input port. Hereby, it does not instantiates one of the three pipes described in Section 8.2.3. Instead, it saves the mapping by using a placeholder pipe. Later, at startup, the scheduler decides individually for each connection which pipe implementation should be used. Further versions of `connectPorts` accept additional parameters. For example, it is

possible to customize the pipe's capacity. Moreover, by passing a pipe factory, the scheduler does not automatically create a pipe by its own, but uses the factory to do so for this particular connection. In this way, TeeTime is open for custom pipe implementations. If a configuration represents a distributed P&F architecture, it also declares the nodes and associates them to distinct sets of stages. For this purpose, TeeTime provides the method `assignNodeToStages`. If the IP addresses of the nodes should not be hard-coded into the configuration, nodes can be named instead. The actual IP addresses are then passed as console arguments and the assignment takes place by their names. In this way, the configuration can be reused for different node setups without recompiling. Finally, the configuration provides another predefined overloaded version of the method `connectPorts`. This version allows to overwrite the default transport protocol (TCP) to either TCP over SSL or UDP. With these additional configuration parameters, TeeTime is able to automatically deploy and start the stages on the specified nodes. More on this topic can be found in Section 9.3. Nevertheless, if the declaration of stages, connections, and nodes is not sufficient for some schedulers, further information can be added to a configuration. For example, the scheduler described in Section 9.1.1 additionally requires to declare each stage either active or passive in order to assign threads to stages properly.

8.2.5 Entity: Execution

An execution represents the runtime environment of the framework. It is responsible for the proper execution of a configuration using a given scheduler. It triggers the initialization, validation, execution, and termination process provided by the particular scheduler. To run a configuration, the *execution* entity provides a blocking and a non-blocking method. The non-blocking method is used to allow the user to cancel a running configuration prematurely.

8.2.6 Entity: Traverser

A traverser represents a service which is responsible for traversing through a P&F configuration. Beginning at an initial set of stages, it follows their connections until all stages have been visited. Depending on its settings, a traverser traverses either along or against the direction of the pipes. Similarly, it visits successor stages using either a breadth-first or a depth-first strategy. It can be applied before, after, and even while running a configuration. Its areas of application are versatile. For example, the scheduler described in Section 9.1.1 uses the traverser to validate the type-safety, to instantiate the best-matching pipe between each pair of ports, and to determine the set of owning stages for each individual thread. Moreover, the traverser is used to propagate signals through the P&F architecture. For example, the start signal initiates the execution of a configuration while the termination signal prepares the termination of the same. In this way, the framework ensures that a data source begins or finishes its work before a subsequent filter or data sink does so.

8.2.7 Entity: Scheduler

A scheduler represents the execution strategy of the framework. Thus, it is responsible for the execution of P&F architectures. On the one hand, it initializes and validates the given configuration. This includes the creation of threads as well their assignment to stage instances. On the other hand, the scheduler chooses the pipe implementation which matches best to each connection between two ports. Finally, it schedules the stages according to its execution model (see Section 9.1 for more details). The way of how the actual scheduler performs these tasks is up to the scheduler's implementation.

Chapter 9

Executing P&F Systems

After describing the framework components in isolation in Chapter 8, we now explain their interaction among each other. In Section 9.1, we introduce our new scheduling approach for executing generic P&F architectures. Furthermore, we describe our implementation of an alternative scheduling approach which is commonly used for streaming applications. Afterwards, in Section 9.2, we present a modular, incremental, and generic approach to introduce parallelism in arbitrary P&F architectures. In Section 9.3, we describe how our framework executes distributed P&F architectures. Finally, we focus on a P&F problem in Section 9.4 which is seldom made a subject of discussion: the execution and, especially, the graceful termination of P&F-based applications which contain (feedback) loops.

9.1 Execution Models

TeeTime provides support for arbitrary scheduling strategies. In the following, we present two particular strategies: one novel strategy (see Section 9.1.1) that is able to execute generic P&F configurations, and one existing strategy (see Section 9.1.2) that is often applied in streaming applications. For each of the schedulers, we describe in detail how it fulfills the tasks which we have defined for the scheduler component in Section 8.2.7.

9.1.1 Parallel Push/Pull Scheduling

In the following, we present a new scheduling approach for P&F architectures. The basic idea of this strategy is to avoid using a global scheduler.

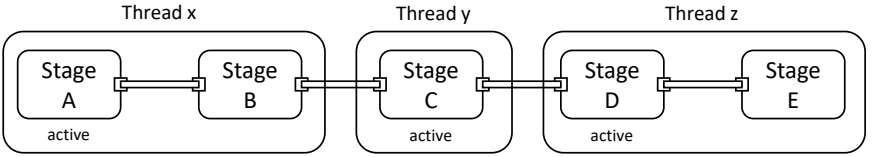
Instead, it delegates any scheduling responsibility to its threads. In this way, it prevents additional synchronization and coordination effort.

We call our new scheduling approach the *parallel push/pull scheduling model*. It extends the naïve approach where each thread executes a unique stage of a given P&F configuration. In this way, a stage pulls input and pushes output in parallel with all of the other stages. As opposed to this naïve approach, our parallel push/pull model is not limited to one stage per thread. Instead, each thread can manage more than one stage. However, a stage still belongs to one single thread only. A stage is therefore not shared among multiple threads. As a result, a change of state does not require any synchronization if the state is not shared.

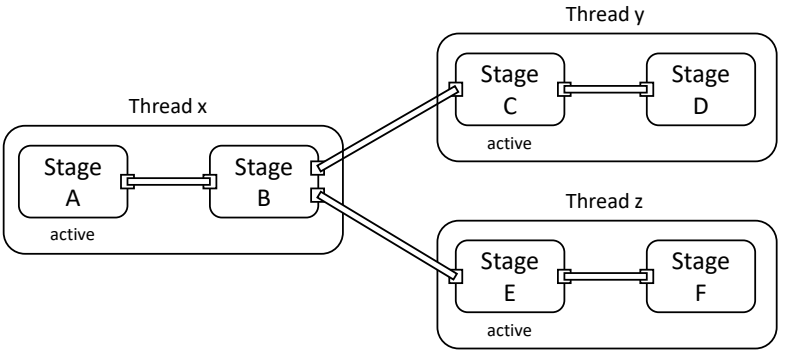
In the following, we describe the scheduler in detail. In particular, we describe how it fulfills its three tasks defined in Section 8.2.7: thread management, pipe instantiation, and execution. First, the scheduler starts a traverser (see Section 8.2.6) to collect all of the stages which are declared active in the given P&F configuration. These active stages represent the entry points for the threads of the scheduler. That means, a thread is responsible for its active stage and all its passive successor stages. Figure 9.1a shows an example P&F configuration with a linear pipeline. The stages `A`, `C`, and `D` are declared active. For this reason, the scheduler assigns one thread to `A`, one thread to `C`, and one thread to `D`. All of the remaining stages are passive, i.e., `B` and `E`. Hence, each of the passive stages is executed by the thread which executes the nearest, previous active stage. Consequently, `thread x` also executes `B` and `thread z` also executes `E`.

Similarly, each of the stages `A`, `C`, and `E` in Figure 9.1b is executed by a dedicated thread. The stages `B`, `D`, and `F` are passive and are thus executed by the threads of the active stages as described above. Once the assignment of each thread has been determined by the scheduler, it creates the threads and assigns them to the corresponding stages in a fully automatic and transparent way.

However, a thread assignment can also be invalid: either if there are conflicting declarations of active stages or if a stage is not assigned to any thread. Figure 9.2 illustrates these two types of conflicts. To detect such invalid thread assignments, the scheduler employs an adapted graph coloring algorithm to find conflicting declarations of active stages. It interprets



(a) Example P&F configuration with a linear pipeline. Stage `A` is declared active and is thus executed by a dedicated thread `x`. The next active stage is `C` so that thread `x` also executes the passive stage `B`. Similarly, the thread `z` of the active stage `D` takes over the execution of the passive stage `E`.



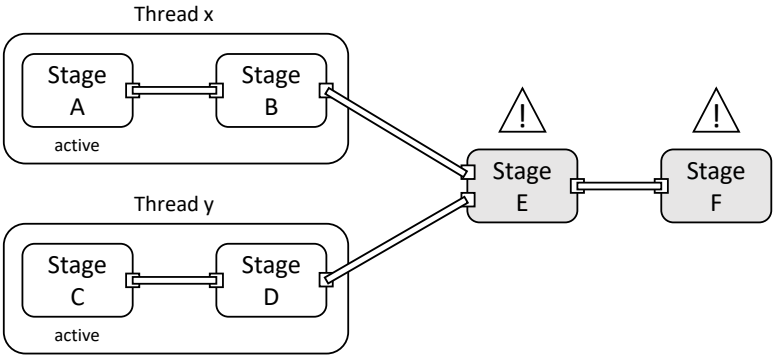
(b) Example P&F configuration with a branch. Stage `A` is declared active and is thus executed by a dedicated thread `x`. The next active stages are `C` and `E` so that thread `x` also executes the passive stage `B`. Similarly, the threads `y` and `z` of the active stages `C` and, respectively, `E` take over the execution of the passive stages `D` and, respectively, `F`.

Figure 9.1. Two example P&F configurations with their active stages and their corresponding thread assignments.

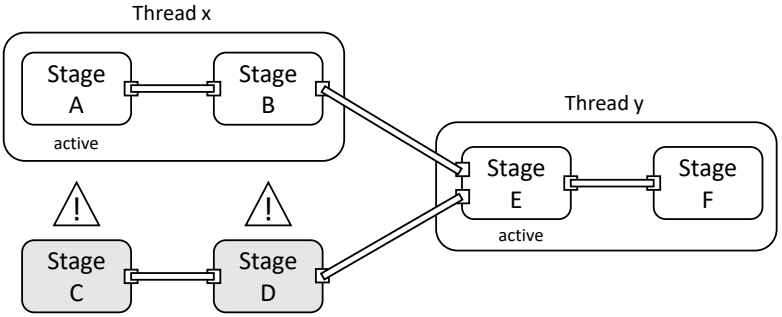
a P&F architecture as P&F graph such that each node represents a stage and each edge represents a pipe. For each active stage, our algorithm works as follows: First, it assigns a unique color c to the active stage. Then, it traverses the whole P&F graph and colors all passive successor stages with the same color c. It starts from the active stage and ends when it reaches another active stage. If a passive stage should be colored although it was already colored before, the algorithm reports a conflict (see Figure 9.2a). Finally, if the traversal has been finished, but a stage was not colored at all (see Figure 9.2b), the algorithm also reports a conflict. Hence, with this error detection approach, we effectively avoid an inconsistent P&F configuration.

After checking for a valid thread assignment, the scheduler traverses the P&F configuration again in order to replace the pipe placeholder instances by actual pipe instances (cf. Section 8.2.4). It decides individually for each connection which pipe implementation should be used. If the node of the source stage is different to the node of the target stage, the scheduler creates a distributed pipe. If the target stage is active, the scheduler chooses a synchronized pipe. Otherwise, that is, if the target stage is passive, the scheduler instantiates an unsynchronized pipe. In this way, the scheduler automatically chooses the most efficient pipe implementation for each individual connection. Moreover, by means of unsynchronized pipes, it also implements the back-pressure policy: it first propagates an element through the whole P&F architecture before producing a new one. Finally, there are stages which do not always require to connect each output port with an input port. For example, consider a stage that outputs its incoming element either via the matching or via the mismatching output port depending on whether it matches a particular condition or not. In some P&F architectures, only the matching port is relevant for the following dataflow such that the mismatching port does not need to be connected. In such cases, the scheduler creates a dummy pipe and assigns it to the corresponding output port.

Afterwards, the scheduler starts all of its threads which first wait for the validation and the start signal. They do not start with the actual execution of their stages yet. The validation signal is immediately sent by the scheduler after starting the threads. It is propagated through the whole P&F configuration beginning by the data source stages. Upon reception, the



(a) Conflict: it is unclear whether thread `x` or `y` should execute the passive stages `E` and `F`.



(b) Conflict: stage `C` and `D` are not assigned to any thread.

Figure 9.2. Two examples for conflicting declarations of active stages.

particular stage performs its individual validation logic. In addition, the scheduler checks for type-safety between the stage and all of its successor stages. It verifies for each output port whether the type is the same or a subtype of the associated input port. This runtime validation mechanism ensures a detection of invalid user configurations early enough before the actual execution begins.

Subsequently, if the user starts the execution (by invoking the method `executeBlocking` or `executeNonBlocking`), the scheduler sends the start signal to all of the (active) data source stages. From there on, the signal is passed through the whole P&F configuration via its pipes. Active data sources now begin with producing elements. Simultaneously, active filters and data sinks start consuming elements by polling their input ports. The scheduler uses polling by default instead of a blocking read mechanism because it assumes a high throughput of elements. Pausing and resuming threads would cause too much delay compared to busy-waiting when expecting hundred of thousands elements per second. Nevertheless, this behavior can be configured individually to adapt the scheduler to the particular scenario.

Besides the initialization and the execution, the scheduler also handles runtime errors. For this purpose, each stage is surrounded by the error catching statement of the used programming language (cf. Java's `try-catch` statement[1]). This allows to identify the location and the type of a potentially occurring error. As soon as an error occurs, it is enriched with the causing stage and passed to an error handler. Since it depends on the given P&F configuration whether the error can be tolerated or needs to terminate the execution, the actual error handler implementation is user-specific. For example, possible handler implementations could ignore or compensate the error. Depending on the handler's return value, the framework either terminates or continues the execution. In this way, it is possible to define arbitrary error handling strategies. Hence, like the (non-)blocking behavior of consumers mentioned above, the error handler represents another part of the framework which is intended to be adaptable for individual usage scenarios.

[1] `https://docs.oracle.com/javase/tutorial/essential/exceptions/catch.html`

Besides the termination by an error handler, the scheduler allows for aborting the execution by the user via the method `abortEventually`. However, the P&F configuration usually triggers its termination by itself when all of the data sources have finished their work. A data source indicates its end of processing by invoking a particular termination method. Since the associated termination condition is very specific for the actual data source, the corresponding stage developer must explicitly define when the data source should terminate itself. The following termination process is then handled automatically by the framework. If a data source has finished its work, it propagates a termination signal through the whole P&F configuration, just like with the validation and start signal. Upon reception of the termination signal, the particular stage performs its individual termination logic. For example, a stage could close a connection to a remote database. If all of the stages of a thread have executed their termination procedure, the corresponding thread terminates itself. In this way, the scheduler ensures an automatic and graceful termination. In contrast, Akka [Lig] does not provide an automatic termination approach. Other frameworks, like FastFlow [ADK+14] and Java's Streaming API [Oraa], which do support automatic termination, forgo supporting arbitrary loops in return. We describe in Section 9.4 what termination and loops have in common and how our framework is able to support both together.

In summary, the scheduler takes over all three tasks defined in Section 8.2.7. The user does not need to care for thread management, pipe instantiation, and the execution itself. Moreover, the scheduler detects invalid thread assignments and provides runtime error handling.

9.1.2 Global Task Pool Scheduling

The following scheduling approach is commonly used to execute several tasks concurrently by multiple threads. This approach uses a global task pool which holds stage executions as tasks. An idle thread can take one task out of the pool and executes the corresponding stage multiple times. Afterwards, it puts the stage back to the pool and begins anew by pulling the next stage.

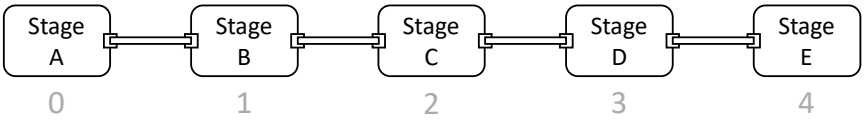
We call this scheduling approach the *global task pool scheduling*. As op-

posed to the parallel push/pull scheduling, a user does not need to declare a stage active or passive. Instead, all of the stages are potentially executed by all of the scheduler's threads. Hence, a stage is not bound to a single thread, but can be executed by several threads. However, a stage is still always executed only by a single thread at each moment in time.

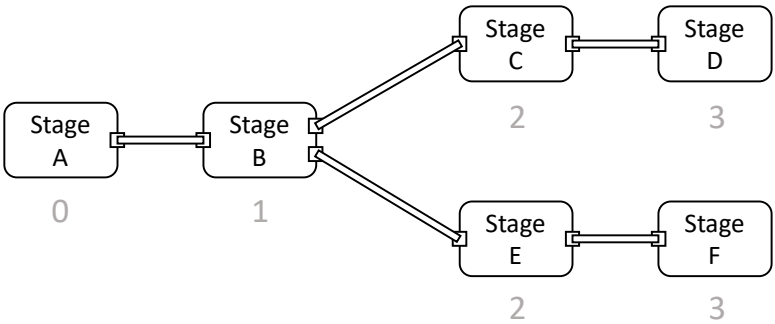
In the following, we describe the scheduler in detail. In particular, we describe how it fulfills its three tasks defined in Section 8.2.7: thread management, pipe instantiation, and execution. Additionally, we describe how we adapt the scheduler (1) to allow arbitrary stage implementations and (2) to enable back-pressure. Usually, a stage may output only a small, fixed number of elements per execution in order to schedule other stages as well. Our improvement allows to output an arbitrary and possibly varying number of elements per execution without neglecting to schedule the remaining stages. Simultaneously, our scheduler implements a variant of the back-pressure policy: it prioritizes the execution of those stages which have a larger distance to data sources than other ones. In this way, the scheduler first propagates elements through the whole P&F architecture before triggering data sources to produce new ones. We expand on both of our improvements after describing the basics of the scheduler itself.

The initialization phase of the scheduler is as follows. First, the scheduler starts a traverser (see Section 8.2.6) to determine the level index of each stage. The level index represents the depth of a stage relative to the data source stages in the whole P&F architecture. Figure 9.3 shows four example P&F configurations with the level indexes of their stages. We use the level index later for implementing the back-pressure policy.

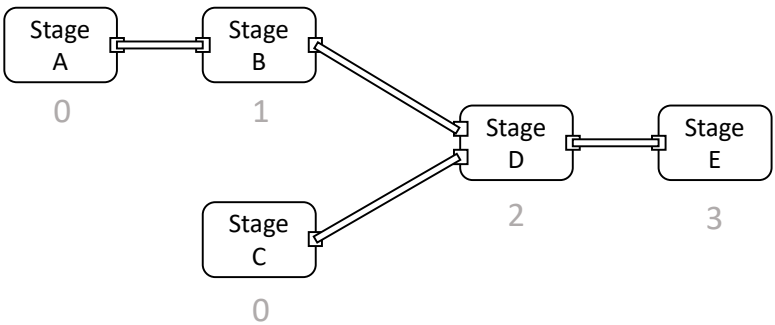
Second, we initialize the task pool, the central data structure for communication between our threads. It stores stage instances as tasks such that a thread, which takes one task out of the pool, executes the given stage instance multiple times. Since we demand that a stage in TeeTime does not require any synchronization, we must forbid a parallel execution of the same stage instance. For this reason, the task pool reserves only one slot per stage instance. Nevertheless, if one particular stage instance has a high workload and would benefit from a parallel execution, we propose our task farm stage (see Section 9.2) to also parallelize such kind of stages. Finally, after initializing the pool, we pass all of the data source stages to the pool



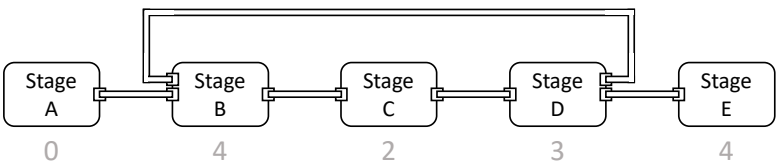
(a) Level indexes of an example pipeline.



(b) Level indexes of an example P&F configuration with a branch.



(c) Level indexes of an example P&F configuration with a merge.



(d) Level indexes of an example P&F configuration with a loop.

Figure 9.3. Level indexes of four example P&F configurations.

such that the threads can later start with executing them to produce initial elements.

As opposed to the previous parallel push/pull scheduling, the global task pool scheduling does not require a thread assignment validation phase. Since each thread can potentially execute each stage instance, there is no invalid thread assignment. Hence, the scheduler directly passes over to the pipe initialization phase. It traverses the P&F configuration in order to replace the pipe placeholder instances by a particular kind of synchronized pipes which supports multi-threading. This type of pipe consists of a synchronized lock-free multiple-producer/multiple-consumer queue. It allows to add and to remove elements by multiple different threads. Although there is always only one executing thread at a time which lets the stage add elements to the pipe (same for removing), this thread may change while running the P&F configuration. In order to properly synchronize the threads with each other, all pipes are thus synchronized in that way.

Subsequently, the scheduler starts the request number of threads. Similar to the parallel push/pull scheduling, it initiates a validation signal and a start signal. Thereupon, each stage performs its individual validation and starting logic. Moreover, the scheduler checks for type-safety by comparing the types of two interconnected ports. The only difference in that phase compared to the previous scheduler is that it additionally starts a number of backup threads. A backup thread is necessary in case where a regular thread has been paused. We will come back to this point and describe why and how the scheduler pauses threads.

If all of the stages are ready to be executed, each regular thread performs the following work in a loop: It starts by taking one of the stage instance from the global task pool. It continues by executing the stage instance multiple times according to the value passed by the user. Finally, it puts the stage instance back to the pool if the stage has not terminated. A stage terminates if it is a data source and has finished its work, or if it receives the termination signal and has no remaining input left. However, this process is not sufficient to execute all stages in the P&F configuration. So far, the threads would only execute the data sources since no stages other than the initial ones are added to the pool. Hence, adding an element via an output port also schedules the corresponding target stage on each x insertions

where x can be chosen by the user. With $x > 1$, we effectively reduce the pressure on the task pool and, more important, ensure that the target stage has enough input when it is executed multiple times. However, if $x > 1$ and the source stage takes some time to produce x elements, the target stage is not scheduled until then. For high-throughput scenarios, this case is uncommon and thus not implemented in our reference implementations. Nevertheless, there are approaches which can handle such a case. For example, we could use an additional thread which periodically schedules stage instances in order to drain remaining elements from the pipes. We refer to the work of Wang et al. [WZT+13, Section 2.6] for an overview of these approaches.

Back-pressure is achieved by prioritizing those stages which have a higher level index. We therefore adapt the common task pool to the structure illustrated by Figure 9.4. The task pool is represented by a list which in turn comprises one pool per list entry. If a thread now wants to add a stage instance to the task pool, the stage's level index is used to access the correct list entry. The stage is then added to the associated pool if the stage has not already been added to it. We use a synchronized, lock-free implementation for the list and a synchronized, lock-free hash set for ensuring uniqueness. If a thread is about to take one stage instance from the task pool, it chooses a stage from the pool which is not empty and associated with the highest level index currently available. We use for each of the pools the same synchronized, lock-free multiple-producer/multiple-consumer queue which we use for the pipes. With this modified task pool, we are able to achieve back-pressure even for P&F configurations which change at runtime.

As already mentioned above, a stage may usually output only a small, fixed number of elements per execution in order to schedule other stages as well. Figure 9.5a shows the execution logic of an example data source stage which outputs one element per execution. This stage manages an iterator and accesses it once per execution (Line 2-3). If the iterator returns an element, this element is used to compute and send a single result via the output port (Line 4-5). Otherwise, the data source signals the scheduler that it has finished its work (Line 7). This implementation allows the scheduler to decide for the thread after each execution whether it should re-execute

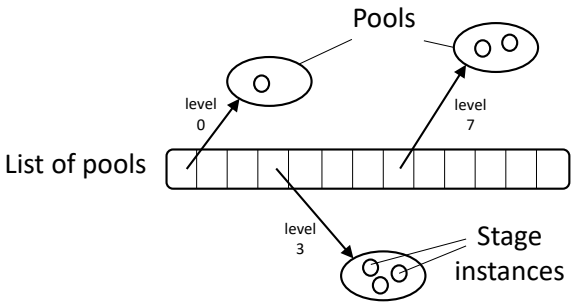


Figure 9.4. A schematic overview of our global task pool

```
1 void execute() {
2   if (iterator.hasNext()) {
3     element = iterator.next();
4     result = compute(element);
5     outputPort.send(result);
6   } else {
7     workCompleted();
8   }
9 }
```

(a) An example data source which outputs one element per execution

```
1 void execute() {
2   while (iterator.hasNext()) {
3     element = iterator.next();
4     result = compute(element);
5     outputPort.send(result);
6   }
7
8   workCompleted();
9 }
```

(b) An example data source which outputs all elements in the first execution

Figure 9.5. Two example data sources which output one or, respectively, multiple elements per execution

the data source or whether it should execute another stage. In contrast, the data source shown in Figure 9.5b outputs all elements of the iterator at once (Line 2-5). This implementation often overfills the pipe associated with the output port and prevents the executing thread to switch the execution to another stage. If, for example, the scheduler pauses the thread in case of reaching the pipe's maximum capacity, the data source implementation could even cause a deadlock.

Our global task pool scheduler is able to handle arbitrary stage implementations, especially both of which are shown by Figure 9.5. In order to

allow producing an arbitrary and possibly varying number of elements per execution, our scheduler is notified on each element insertion and on each capacity violation. On element insertion, it schedules the target stage on each xth element as already described above. If the pipe is full, i.e., if a capacity violation has occurred, it proceeds as follows. First, it schedules the target stage in order to drain the pipe. Then, it pauses the source stage by stalling the currently executing thread t. In this way, the current position of the execution is saved. In order to eventually awake the source stage (and t) again, it is scheduled directly after the target stage. If, then, another thread is about to execute the source stage, it awakes the sleeping thread t and stalls itself afterwards without executing the stage. In return, t executes the stage by continuing the execution from its previous position. Since this approach reduces the number of available threads by one on each capacity violation, the scheduler compensates this loss each time by consulting an additional thread from a pool of backup threads. As soon as a backup thread is stalled again, it is put back to the thread pool. For each stage, at most one backup thread is necessary. Thus, the number of required backup threads is limited by the number of stages in the P&F configuration. In this way, the scheduler is able to support stages which produce a varying number or all of its elements in one single execution. Runtime error handling is supported in the same way as described for the parallel push/pull scheduler. The same applies for the termination of a P&F configuration. Only the termination of the threads is different. The threads terminate not until all stages of the P&F configuration have terminated, i.e., the termination signal must pass all stages beforehand.

In summary, the scheduler takes over all three tasks defined in Section 8.2.7, just like the parallel push/pull scheduler does. The user does not need to care for thread management, pipe instantiation, and the execution itself. As opposed to the parallel push/pull scheduler, there is no invalid thread assignment since the assignment is handled automatically by the scheduler and not by the user. In return, the implementation is much complexer. Moreover, it requires from the user to find and set the optimal value for the number of stage executions per task to compensate the higher synchronization effort. Apart from that, it also provides runtime error handling. Due to our improvements, the scheduler additionally implements a

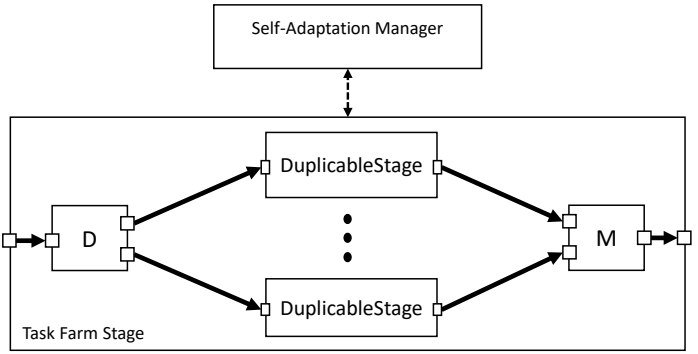


Figure 9.6. Architecture of our task farm stage. `D` represents the distributor, `M` represents the merger, and the `DuplicableStages` represent the instances of the stage to be parallelized.

variant of back-pressure and allows producing an arbitrary, varying number of elements per stage execution. We refer to Section 13.2 for a comparative performance evaluation of both schedulers.

9.2 Parallelization via the Task Farm Stage

In this section, we present our approach to increase the throughput of P&F architectures. Its description is mainly taken from our previous work [WWH16]. Our main idea is to provide a composite stage that is wrapped around an existing stage in order to parallelize it. We call this composite stage the Task Farm Stage (TFS) since it utilizes the Task Farm Parallelization Pattern [AD99]. The structure of the TFS is illustrated in Figure 9.6. Although our approach can also be applied to distributed systems, we have only implemented and evaluated it for the execution on multi-core systems, so far.

The TFS is a composite stage with additional parallelization functionality. Its child stages are the Dynamic Distributor (shown as *D* in Figure 9.6), the Dynamic Merger (shown as *M* in Figure 9.6), and the Duplicable Stage. The

Duplicable Stage represents the stage which should be parallelized. Usually, there are multiple instances of the Duplicable Stage at runtime, each running in a dedicated thread. These instances may vary dynamically during the execution as a result of workload changes. Additionally, a self-adaptation manager is used to monitor the TFS and to maximize its performance (see Section 9.2.4).

In general, the task farm parallelization pattern can be applied to all P&F architectures. However, our TFS currently defines the following constraints regarding the P&F architecture to limit the complexity:

- The TFS in general and the Duplicable Stage in particular may not contain feedback loops. For example, output ports of a Duplicable Stage must not lead to the distributor of its own TFS. This limitation allows us to measure the pipe throughputs more precisely, which is necessary for the self-adaptation manager (see Section 9.2.4).

- Each Duplicable Stage has exactly one input port and one output port. More input and output ports would possibly require more input and output ports for the TFS itself, leading to a higher complexity.

Nevertheless, our TFS covers the task farm pattern variants 4.2b-4.2d from Figure 4.2. It is able to parallelize not only primitive stages, but also composite ones, since the Duplicable Stage may be a composite stage.

The workflow of a typical traversal of an incoming data element at the input port of the TFS is as follows. At first it arrives at the Dynamic Distributor. The distributor chooses which instance of the Duplicable Stage (worker stage) is going to process the element according to a specified distribution strategy. Afterwards, the distributor sends the element to the output port leading to the chosen worker stage. The worker stage then processes the data element and finally sends it to the Dynamic Merger. The task of the merger is to merge incoming elements to a single output stream. The concrete behavior of the merger and the order of the elements in the output stream depends on the specified merging strategy.

In the following sections, we discuss the components of the TFS. In Section 9.2.1, we explain the behavior of the Dynamic Distributor and the Dynamic Merger as well as some of their strategies. Section 9.2.2 addresses

the definition of the Duplicable Stage. Section 9.2.3 explains the process of adding and removing a worker stage at runtime.

9.2.1 Dynamic Distributor & Dynamic Merger

The Dynamic Distributor and the Dynamic Merger have to process all data elements entering the TFS. Therefore, an efficient implementation of these stages is a key requirement to an efficient TFS. Furthermore, as we intend the self-adaptation manager to be able to dynamically add or remove worker stages, the distributor and merger have to be able to add and remove ports at runtime.

As mentioned above, the task of the **Dynamic Distributor** in the TFS is the distribution of each data element to a worker stage. We can define an arbitrary strategy for the distributor to define its exact behavior. In the following, we discuss three possible distribution strategies.

CloneStrategy

The distributor duplicates each incoming data element according to the current number of worker stages, before sending an exact copy to each worker stage. Therefore, all worker stages produce the same output elements which are then merged and passed to the output port of the TFS. Although this strategy trivially increases the overall throughput, it also violates the original semantics of the Duplicable Stage.

BlockingRoundRobinStrategy

This strategy causes the distributor to send the current input element to the next worker stage selected in round-robin order. Hence, each worker stage has approximately the same workload, assuming the processing time is the same for each data element. However, as the capacity of pipes is often bounded, a distribution strategy must also handle the case of a full pipe. This strategy waits until the pipe is free again, which can waste time if other worker stages have non-full input pipes.

NonBlockingRoundRobinStrategy

This strategy behaves almost exactly like the BlockingRoundRobinStrategy. However, if the input pipe of the chosen worker stage is full, it searches for another worker stage in round-robin order. Therefore, this distribution strategy provides the best performance.

Although the task of the **Dynamic Merger** is the opposite of the task of the Dynamic Distributor, it uses very similar concepts regarding its merging strategy. In the following, we discuss two possible merging strategies.

BlockingRoundRobinStrategy

This strategy behaves almost analogous to its counterpart of the Dynamic Distributor. The only difference is that this strategy skips input ports which are no longer in use (e.g., if the corresponding worker stage has been removed). Otherwise, we would cause a livelock or, respectively, a deadlock depending on whether this strategy is implemented in a busy-waiting or in a blocking-read way.

NonBlockingRoundRobinStrategy

This strategy behaves analogous to its counterpart of the Dynamic Distributor. It searches for the next non-empty input port in round-robin order and passes the corresponding element to the merger's output port. Due to the absence of any blocking mechanism, this merging strategy provides the best performance.

Additionally, the Dynamic Distributor and the Dynamic Merger need a way to dynamically add and remove output and input ports, respectively. Therefore, we introduce two port actions which enable the distributor and merger to provide this functionality. The first action triggers the distributor or the merger to add a new port to itself and to connect it with the new worker stage. The second action triggers the distributor or the merger to remove an existing port from itself.

Both the distributor and the merger provide a port action interface to the self-adaptation manager. In this way, the self-adaptation manager can

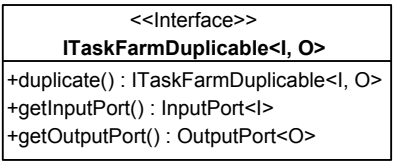


Figure 9.7. If a stage implements `ITaskFarmDuplicable`, our TFS is able to duplicate it and to execute it in parallel.

dynamically add or remove worker stages from the TFS depending on the workload.

9.2.2 Duplicable Stage

The TFS can parallelize an arbitrary (composite) stage if the stage is a Duplicable Stage, i.e., if it implements the interface `ITaskFarmDuplicable` shown in Figure 9.7. The interface requires to implement three methods.

The two methods `getInputPort` and `getOutputPort` are necessary to retrieve the input port and the output port of the Duplicable Stage. The TFS requires access to these ports to properly connect the worker stages to the Dynamic Distributor and the Dynamic Merger.

The `duplicate` method is a crucial part of the interface. It generates an additional worker stage from the corresponding instance of the Duplicable Stage. Thus, the user of the TFS can freely implement the duplication behavior of the Duplicable Stage, providing a way to implement most use cases. The duplication method is called whenever the self-adaptation manager decides to add another worker stage to the TFS.

The programming effort to migrate an existing stage to a duplicable stage is low. It only needs to implement the interface `ITaskFarmDuplicable`. First, the methods `getInputPort()` and `getOutputPort()` must return one input port and, respectively, one output port of the existing stage. Second, the method `duplicate()` must return a new copy of the existing stage. If the stage is stateless, an invocation of the constructor is sufficient. Otherwise, the implementation must also ensure that the internal and shared state attributes are copied in a correct and thread-safe way.

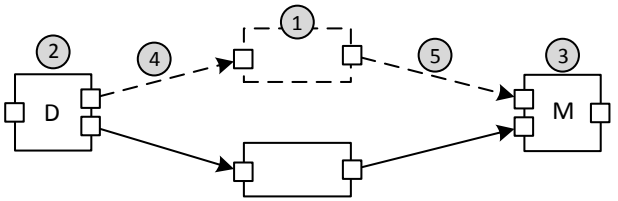


Figure 9.8. Duplicating a stage: (1) the new worker stage, (2) the distributor, (3) the merger, (4) the new worker stage's input pipe, and (5) the new worker stage's output pipe.

9.2.3 Addition and Removal of Worker Stages

The TFS achieves its parallelization by dynamically adding and removing worker stages. The self-adaptation manager monitors the workload and decides whether further parallelization is reasonable. For this purpose, the TFS requires to implement some logic for the dynamic addition and for the dynamic removal of the worker stages.

The process to duplicate a worker stage is illustrated in Figure 9.8. The first step is to create a new worker stage (1) by using the duplication method on an arbitrary existing worker stage inside the TFS. To prevent race conditions between the new worker stage and its not yet existing input and output pipes, it does not yet process data elements. To connect the Dynamic Distributor (2) and the Dynamic Merger (3), we then create the worker stage input pipe (4) between the distributor and the worker stage. As discussed in Section 9.2.1, we use the CreatePortAction to connect the pipe with a new output port of the distributor. Similarly, the worker stage output pipe (5) between the worker stage and the merger is created. The pipe is then also connected to the merger via the CreatePortAction. Finally, the new worker stage (1) is started in a dedicated thread and begins processing elements. In this way, it increases the total throughput of the TFS.

The process to remove an existing worker stage is illustrated in Figure 9.9. The only step required to remove an existing worker stage is to deactivate the distributor's output port which is connected to the worker stage's input pipe (1). Afterwards, the distributor will no longer send data elements

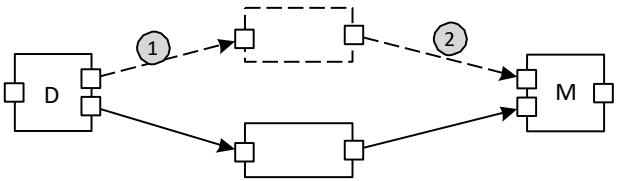


Figure 9.9. Removing a duplicated stage: (1) the distributor closes its output port, and afterwards (2) the worker stage eventually closes its output port.

to the removed worker stage, allowing it to safely process all buffered elements. If the removed worker stage has no further elements to process, it terminates itself and closes its output port (2). Since the merger uses the BlockingRoundRobinStrategy (see Section 9.2.1), it detects the closed port and proceeds with another worker stage.

It is vital that the removal of a worker stage allows the removed stage to work off all remaining elements buffered by the input pipe. Otherwise, the TFS might loose some elements whenever a worker stage is removed.

Because a TFS has multiple worker stages at runtime, it has to decide which available stage should be removed. We choose the strategy to always remove the worker stage whose input pipe has the least number of buffered data elements. Since the lower the number of buffered elements, the less is the time to wait for the stage to be actually removed. Hence, this strategy usually provides a low latency for removing worker stages.

9.2.4 Structure of the Self-Adaptation Manager

The task of the Self-Adaptation Manager (SAM) is to control the addition and removal of worker stages in its corresponding TFS. Therefore, each TFS inside the P&F architecture requires a dedicated SAM. The design of the SAM is based on the general design of an adaptable software system described by Massow et al. [MHH11] and is shown in Figure 9.10.

The SAM consists of three components. The Monitoring component monitors the throughput of the pipes which connect the Dynamic Distributor and the worker stages with each other. In this way, we measure the performance of each worker stage. Afterwards, the Analysis component

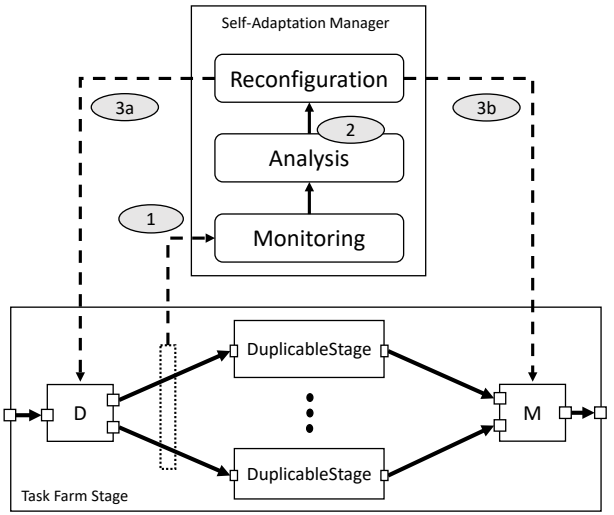


Figure 9.10. The SAM and how it interacts with the TFS: the Monitoring component measures the throughput of the pipes (1), the Analysis component computes a corresponding throughput score (2), and the Reconfiguration component sends actions to the distributor (3a) and to the merger (3b) to adapt the TFS.

analyzes the measurements of the Monitoring component. It calculates how much the throughput of the TFS has changed since the last few measurements. The last component is the Reconfiguration component, which takes the result of the Analysis component and decides whether the TFS should add or remove a worker stage. The cycle as shown in Figure 9.10 is then completed and starts again with the Monitoring component after a user-defined delay (our default is 50 ms). In Section 9.2.5 to 9.2.7, we introduce each of the three components in more detail.

9.2.5 Monitoring Component

As mentioned above, the Monitoring component measures the throughput of the pipes between the Dynamic Distributor and the worker stages (see (1) in Figure 9.10). We compute the throughput of a single pipe by n/td where

n is the amount of elements pulled from the pipe since the last SAM cycle and td is the time difference between the current and the last SAM cycle. This throughput definition directly reflects the actual productivity of the corresponding worker stage without being influenced by element buffering or similar issues.

Afterwards, we compute the sum and the average of the throughputs of the worker stages. We save these values for later use to make informed decisions on whether we can further optimize the TFS performance by adding or removing a worker stage. As we collect these measurements in every SAM cycle, we construct a history of measurements.

9.2.6 Analysis Component

The Analysis component analyzes the most recent measurement and the measurements of the Monitoring component to calculate a so-called *throughput score*. The throughput score serves as action indicator so that the Reconfiguration component can decide whether it should add a worker stage, remove a worker stage, or do nothing.

Let $v \in \mathbb{N}$ be the most recent measurement. Let $p \in \mathbb{N}$ be a calculated predicted throughput based on a number of recent history measurements. The throughput score $ts \in \mathbb{Q}$ is then defined by

$$ts = \frac{v - p}{p}.$$

For example, let $v = 4$ and $p = 2$, i.e., the throughput has increased since the last measurement. The corresponding throughput score is $ts = \frac{v-p}{p} = \frac{4-2}{2} = 100\%$. Thus, the measured throughput is 100% higher than expected. A negative score, however, indicates that the measured throughput is smaller than expected. For example, let $v = 1$ and $p = 2$. Then, the corresponding throughput score is $ts = \frac{v-p}{p} = \frac{1-2}{2} = \frac{-1}{2} = -50\%$. Thus, the measured throughput is 50% smaller than expected. In general, the throughput score represents the throughput increase relative to p.

To calculate p, we can choose one of multiple prediction algorithms, each providing some unique characteristics regarding their forecasting behavior.

Mean Algorithm

The mean algorithm uses a number of previous throughput measurements of the TFS and calculates the average value of it. This value will then be interpreted as the expected value of the current point in time.

This is one of the least complex algorithms that can be used to predict values. While it is very fast due to its low computational effort, it does not produce acceptable results for our use case. If the throughput is constant, the average of the last few measurements is always a correct prediction, leading to correct forecasts. However, the algorithm looses this advantage for every other runtime behavior and cannot accurately predict future throughput measurements.

Weighted Algorithm

The weighted algorithm is a variation of the mean algorithm. It also computes the average of a certain number of measurements, but it additionally adds weights in such a way that more recent measurements have more impact on the prediction.

For our use case, this algorithm is more usable than the mean algorithm since it reacts faster on changes of the throughput. However, the algorithm does not behave well for linearly and exponentially growing throughputs since it does not extrapolate any behavior of the throughput measurements. For an irregular runtime behavior, it in turn yields comparatively good results since extrapolation is not possible at all.

Regression Algorithm

The regression algorithm uses a statistical regression algorithm to predict the throughput at the current point in time. It uses at least two previous measurements to construct a straight line $y = ax + b$, where x is a point in time, y the throughput at that time, and $a, b \in \mathbb{R}$. If more than two measurements are used, a straight line is found that corresponds best to all provided data points. The prediction can be obtained by solving the equation by setting x to the current point in time.

This algorithm behaves very well for any nearly linear runtime behavior of the TFS. Exponential and other regular behavior can also be accurately predicted by using a lower amount of data points for the line construction. This is possible since exponential functions are mathematically nearly linear for a small interval. However, since the regression algorithm assumes a linear function in the runtime behavior, it can yield very inaccurate predictions for irregular behavior.

9.2.7 Reconfiguration Component

The Reconfiguration components directly controls the TFS. It decides, depending on the throughput score calculated in the Analysis component, whether a worker stage should be added, removed, or kept running. For this purpose, the component communicates with the distributor and with the merger by sending actions to them (see (3a) and (3b) in Figure 9.10). There are the following three different actions.

The *add action* represents the addition of a new worker stage, resulting in a higher degree of parallelization of the Duplicable Stage. This action is chosen if the throughput score was positive and above a given positive throughput boundary, i.e., the addition of the last worker stage gained a performance boost.

The *remove action* represents the removal of a worker stage. It is triggered in two situations: either (1) when the workload has decreased after some time, or (2) when the workload is non-decreasing, but the last addition of a new worker has not sufficiently increased the throughput. The TFS recognizes the first situation by means of a negative throughput score which, in addition, has to be below a given negative throughput boundary. The second situation is detected by means of a non-negative throughput score which is below the positive throughput boundary. By removing a worker stage, the associated processing unit is released. In this way, we avoid an inefficient usage of the processing units. Moreover, we reduce the communication overhead of the TFS' internal components.

The third and last action is the *no-op action*. It is triggered whenever the current throughput score is between the negative and the positive throughput boundary.

In summary, our task farm stage provides a generic and modular way to introduce parallelism into P&F architectures. Based on the measured overall throughput, it autonomously changes the number of its duplicable stages on demand. For this purpose, we integrated a runtime monitoring facility into the synchronized pipe which records the push and the pull throughput. This feature enables an efficient runtime adaptation of P&F architectures with respect to varying workloads.

9.3 Distributed Execution of Stages

Besides a parallel execution on multi-core systems, TeeTime also provides support for a distributed execution of P&F configurations. By declaring a single distributed configuration, the framework is able to automatically deploy the stages on the associated nodes and connect them with each other via distributed pipes (see Section 8.2.3). IP addresses are not hard-coded into the configuration, but are abstracted by node identifiers, so that the configuration can be reused in various different execution environments. For this purpose, TeeTime provides automatic discovery of nodes in the network. In addition, it offers fault tolerance in the sense of fault detection, fault isolation, and graceful termination as opposed to an uncontrolled crash. Apart from that, a distributed configuration behaves the same way as a non-distributed configuration. Each node uses its own scheduler (see Section 9.1) and executes its stages either sequentially or in parallel. A distributed configuration can be declared in two ways: either directly in the target programming language or indirectly with our domain-specific language for TeeTime configurations. We refer to Section 10.2 for more information about the two notations. The remote deployment and the distributed execution of the configuration utilize the actor model. In the following, we describe how both of them work in detail.

9.3.1 Remote Deployment

A distributed configuration is deployed among multiple nodes by using actors. Each node takes on the role of the master or of a worker. The master

node represents the control node from where the execution is started by the user. Simultaneously, the master node can take on the role of a worker, too. A worker node represents a distinct part of the whole distributed configuration. Thus, all worker nodes together form the complete P&F configuration. The master node starts by executing the given distributed configuration. However, the execution does not start the scheduler and let the threads run the stages. Instead, it triggers a master actor which waits for the registration of worker nodes. A worker node registers itself by starting a worker actor and let it send a corresponding registration message. If the master actor receives at least as many registration messages as nodes are declared by the distributed configuration, it assigns each worker actor a distinct part of the configuration. Our deployment approach assumes that the whole distributed configuration is deployed on all of the worker nodes in advance. There is no transmission of the configuration or of a single part via the network. The only purpose of the assignment process is to instruct the worker actors to execute their particular configuration parts. Nevertheless, this approach could be extended to also send configuration parts via the network. Figure 9.11 shows a corresponding schematic overview.

9.3.2 Distributed Execution

As soon as a worker actor has received its assignment, it initializes its configuration part just like any other non-distributed configuration. Thereby, the distributed pipes automatically create and connect their sender actors and, respectively, receiver actors (cf. Section 8.2.3). Afterwards, the worker actor sends an acknowledge message to tell the master actor that it is ready to start its configuration part. When the master actor has received such a message from all of its workers, it instructs them to start their individual executions.

The execution of each configuration part proceeds just like the execution of any non-distributed configuration. If a data source has finished, it triggers a termination signal which is propagated through the whole distributed configuration via the distributed pipes. If the execution of any configuration part terminates unexpectedly, for example, due to an error, the whole distributed configuration is terminated, but in a controlled, graceful manner.

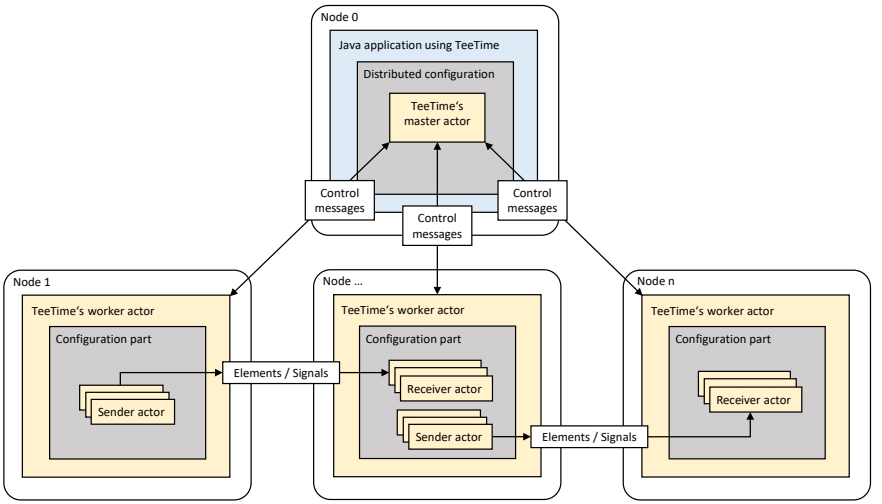


Figure 9.11. Schematic overview of the remote deployment and the remote communication in TeeTime (based on [Ech17]). TeeTime configuration entities are colored in gray; TeeTime network communication entities are colored in yellow.

For this purpose, TeeTime uses the following fault handling strategy. It puts the master actor partially in charge of handling failures by declaring it as system monitor and fault observer. All of the workers periodically send a heartbeat to the master such that the master is able to detect and to react on an unavailable worker node. In such a case, the master triggers the termination of each individual configuration part by propagating standard termination signals. If a worker has caused an error, but is still available, it autonomously informs the master about the error. Afterwards, it shuts down without waiting for a response from the master.

TeeTime intentionally does not save the current state of the whole execution as it would require to monitor and to log each element and each stage's state on each participating node. Besides the monitoring effort, this approach would require a periodical, system-wide halt to persist the nodes running parallel to each other at the same point in time. Although we wish to recover from a failed state and to resume the execution from there

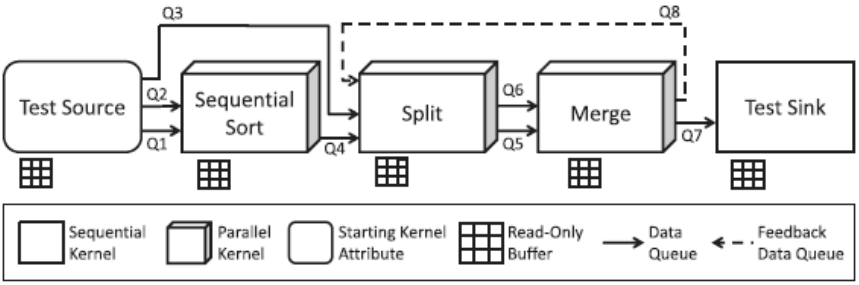


Figure 9.12. A P&F configuration from [ME15] implementing the merge sort algorithm.

on, the synchronization overhead would be too high. Moreover, the error could occur again after restoring if it depends on an invalid input element. Hence, TeeTime only performs a graceful shutdown and lets the user decide whether the P&F configuration should be restarted or not.

9.4 Feedback Loops in P&F Configurations

Feedback loops provide an easy way to implement recursive functions in P&F architectures. Figure 9.12 shows a corresponding P&F architecture which utilizes a loop for the recursive behavior of the well-known merge sort algorithm. The `Merge` stage either outputs the merged array to Q7 or to Q8 depending on whether the array has already been merged completely or whether the array requires further merge operations.

Another loop example is shown in Figure 9.13. The stage `Counter` counts the number of elements passed from input port `I1` to output port `O1`. For this purpose, it increments the current number at its second input port `I2` by one and passes the result to its second output port `O2`. Since `O2` and `I2` are connected with each other by a feedback pipe, the stage effectively uses the pipe as state container for its counter variable.

However, loop-based P&F configurations in general and both examples in particular have some problems. For example, a deadlock or a livelock

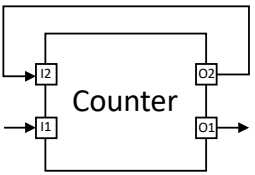


Figure 9.13. A P&F stage which uses a reflexive pipe to represent state outside of the stage itself.

could occur during the execution. If a stage A produces elements for a another stage B which in turn produces elements for A via a feedback pipe, then A could wait for B and vice versa in certain situations. Such mutual dependencies are difficult to identify, especially for static scheduling algorithms [KM08; PD10]. A P&F framework would have to guarantee for a given pair of a scheduler and a configuration that both together are free of deadlocks. So far, our framework does not support such a complex analysis. We recommend to design stages in a defensive manner such that they do not consume input in a particular order.

A deadlock can also occur in terms of the automatic termination of P&F configurations. Usually, a stage terminates if it has received a termination signal from all of its input ports. Thereby, it passes the termination signal to its output ports. The `Split` stage from Figure 9.12, however, would receive a termination signal from its feedback pipe only if it had already passed a termination signal to the `Merge` stage. Hence, this termination strategy leads to a deadlock or a livelock and therefore does not work for loop-based P&F configurations.

An alternative termination strategy shuts the whole P&F configuration down, if all data sources have terminated, none of the stages are currently executed, and all pipes are empty. In such a situation, all of the remaining stages would not produce any element anymore because there is no input left to consume. For this reason, the whole P&F configuration can be terminated safely. Although this strategy works with the configuration from Figure 9.12, it does not work with the configuration from Figure 9.13 because the feedback pipe is never empty.

The actual problem, however, is not the scheduler or the termination strategy, but the concept of feedback pipes itself. By definition, the P&F style prescribes that a stage should be declared in isolation, that is, independent of other stages and particular configurations. However, as soon as a P&F configuration contains a loop, it also requires a termination condition which usually depends on one or more stages. In the case of the merge sort configuration from Figure 9.12, the termination condition is implemented within the `Merge` stage which breaks the concept of isolation for stages. The `Merge` stage cannot be implemented anymore without also considering the `Split` stage and the merge sort algorithm as a whole.

Hence, in general, we recommend to avoid using feedback loops in P&F configurations. Nevertheless, we allow to declare loops with TeeTime, especially for research reasons. However, we note that defining a particular loop-based P&F configuration requires careful consideration in terms of scheduling, termination, and stage design.

Chapter 10

Developing P&F Systems

Besides the ability to model arbitrary P&F architectures, a P&F framework should also support the modeling process itself. On the one hand, it could provide support by following matured guidelines and best-practices to ease its handling for the application developer. On the other hand, it could additionally be shipped with a set of tools which widens and facilitates its applicability. In this section, we present such supporting elements for TeeTime. In Section 10.1, we describe how TeeTime eases the definition of stages. Afterwards, we explain in Section 10.2 how TeeTime supports the definition of P&F configurations. Finally, in Section 10.3, we present some tools and techniques to improve the debugging and the profiling of P&F configurations.

10.1 Definition of Stages

TeeTime supports the definition of stages by providing a minimal, but sufficient application programming interface (API) to develop arbitrary stages. The framework allows a stage to declare not more than its ports, its execution logic, and optionally its handlers for the initialization and the termination event. It follows the principle of information hiding by applying encapsulation using, for instance, visibility modifiers like `protected` and `private`. In this way, internal data and behavior are not exposed to the application developer. Moreover, the way of how to declare the entities mentioned above is prescribed by particular, statically named methods. For example, the application developer is forced by the compiler to implement the method named `execute` in order to define the execution logic for a stage. That means, in particular, the framework does not use any kind of runtime

resolution mechanisms like Java's Reflection API. In summary, we paid special attention on the design of TeeTime's API such that the application developer has only few possibilities to declare an erroneous stage.

We have also implemented and experimented with a domain-specific language (DSL) for declaring TeeTime stages. However, we came to the conclusion that its development and its maintenance are too laborious. Our main goals were (1) the reduction of errors by the application developer, (2) a more intuitive syntax, and most importantly (3) a unified stage definition language. The latter goal should allow us to generate the stage in arbitrary target programming languages. However, such a stage DSL (independent of our particular implementation[1]) has more disadvantages than benefits compared to a general purpose language like Java or C++. For example, consider Listing 10.1 and 10.2 which show an example stage named `SimpleStage` written with Java and with our stage DSL, respectively. The DSL effectively reduces the amount of code by hiding the visibility modifiers, the inheritance of TeeTime's base type `Stage`, and the instantiation of ports. In this way, it offers less potential for errors and provides a more intuitive syntax. However, the major and most critical code part, i.e., the execution logic and the event handlers, cannot be written in a more compact or more general way. For this purpose, the DSL ought to provide a syntax and a semantics which cover all of the existing object-oriented programming languages. Although there are approaches, like XBase [EEK+12], which provide a unified expression language for multiple different languages, none of them support arbitrary languages. The development and testing of such a DSL would result in an enormous engineering effort. Even worse, libraries cannot be used at all if they are not written in the syntax of the DSL. Since, otherwise, the DSL could not be used to generate code in arbitrary programming languages. Code generation would be limited to those programming languages for which an implementation of the library is available. Hence, our latest state of development uses a hybrid approach. For the declaration of the ports and the events, it uses the syntax of the DSL. For the implementation of the events, it uses the syntax of the target programming language by linking to the file which contains the language-specific code (see Line 15-17

[1] `https://build.se.informatik.uni-kiel.de/teetime/Teetime-Stage-DSL`

Listing (10.1) The `SimpleStage` written with Java.

```
1  package a.b.c;
2
3  import x.y.z.MyTypeFromLibrary;
4
5  public class SimpleStage extends Stage {
6    private InputPort<String> inPort = super.createInputPort();
7    private OutputPort<Integer> inPort = super.createOutputPort();
8
9    protected void onStarting() {...}
10
11   protected void execute() {
12     String element = inPort.receive();
13     // ...
14   }
15
16   protected void onStarting() {
17     test.path.AnotherHandler.run(); // static method
18   }
19 }
```

Listing (10.2) The `SimpleStage` written with our domain-specific language.

```
1  package a.b.c {
2    stage SimpleStage {
3      import x.y.z.MyTypeFromLibrary
4      input port String inPort
5      output port Integer outPort
6
7      on starting {...}
8
9      on executing {
10       // language-specific code makes no sense in the DSL
11     }
12
13     on terminating {
14       package test.path
15       class AnotherHandler
16       method run
17     }
18   }
19 }
```

in Listing 10.2). However, this approach still requires one implementation for each programming language which should be supported by the DSL. For these reasons, we do not recommend any stage DSL.

Nevertheless, TeeTime provides further support to the application de-

veloper. It offers predefined stages which are frequently required for many different scenarios. For instance, it provides abstract stages for defining data sources, filters, and data sinks. Another abstract stage eases the definition of composite stages. Moreover, TeeTime makes the stages *distributor* and *merger*, which have already been used by the task farm stage in Section 9.2, available for reuse.

In our opinion, the definition of a stage also includes the specification of associated tests. Remember that a common unit test checks the behavior of a method by comparing the expected return value with the actual return value. As opposed to that, a stage cannot run outside a P&F configuration and may have more than one return value, i.e., multiple output ports. Consider Listing 10.3 which shows an example unit test for a stage called `Incrementer`. Let us assume this stage increments each incoming number by one and sends the result to its output port. In order to test this behavior, we must define a P&F configuration (Line 1-14) which consists of a data source, the incrementer itself, and a data sink. The data source passes each input element from the test input to the incrementer. The data sinks collect all outgoing elements from the incrementer so that we can compare them against our expected elements. In the test, we first instantiate the configuration as shown in Line 18. Afterwards, we pass it to an execution and start it (Line 20-21) in order to finally compare the actual result with the expected one (Line 23). The structure of this test reveals two major disadvantages. First, the configuration must in general provide one getter-method per output port of the test stage (here: `getOutputElements()`) to allow accessing the collected output elements from within the unit test. Second, the stage itself does not appear in the test at all which makes it difficult to grasp the intention of the test. For these reasons, we propose a different approach.

Our solution is (1) to hide the configuration and the execution from the test, and (2) to focus on the input and the output of the stage. For this purpose, we developed an internal domain-specific language as a library in Java. It automatically creates and executes a configuration with the amount of data sources and data sinks necessary to test the given stage. Listing 10.4 shows the same test from Listing 10.3 written with the new stage testing DSL. In Line 3, we create an instance of the stage to be tested, that is, of

Listing (10.3) A unit test for an example stage `Incrementer` written with TeeTime's default entities.

```
1  class IncrementerConfig extends Configuration {
2    IncrementerConfig(List<Integer> inputElements) {
3      dataSource = new DataSource(inputElements)
4      incrementer = new Incrementer();
5      dataSink = new DataSink();
6
7      connectPorts(dataSource.getOutputPort(), incrementer.getInputPort());
8      connectPorts(incrementer.getOutputPort(), dataSink.getInputPort());
9    }
10
11   List<Integer> getOutputElements() {
12     return dataSink.getCollectedElements();
13   }
14 }
15
16 @Test
17 public void shouldExecuteIncrementerCorrectly() throws Exception {
18     IncrementerConfig config = new IncrementerConfig(1, 13, 42);
19
20     Execution<IncrementerConfig> execution = new Execution<>(config);
21     execution.executeBlocking();
22
23     assertThat(config.getOutputElements(), is(2, 14, 43));
24 }
```

Listing (10.4) A unit test for an example stage `Incrementer` written with TeeTime's stage testing DSL.

```
1  @Test
2  public void shouldExecuteIncrementerCorrectly() {
3    Incrementer incrementer = new Incrementer();
4    List<Integer> outputElements = new ArrayList<>();
5
6    test(incrementer).and()
7      .send(1, 13, 42).to(incrementer.getInputPort()).and()
8      .receive(outputElements).from(incrementer.getOutputPort())
9      .start();
10
11     assertThat(outputElements, is(2, 14, 43));
12 }
```

the `Incrementer`. In Line 4, we create the result list which will contain the output elements of the `Incrementer`. Line 6-9 show the actual application of the stage testing DSL. The call to `test(..)` is the entry point to the DSL. It explicitly declares the stage to be tested. Developers different to the tester should now be able to understand the intention of the test more easily than before. The DSL allows to pass elements to each input port which a stage provides. Line 7 shows the two associated calls `send(..)` and `to(..)` for the input port of the `Incrementer`. Similarly, the DSL allows to individually access the elements passed to each output port. Line 8 shows the two associated calls `receive(..)` and `from(..)` for the output port of the `Incrementer`. The final call to `assertThat` in Line 11 then performs the comparison using the result list created previously and filled within the execution. We consequently used our own stage testing DSL for all of the stages which we provide with the Java reference implementation.[2]

10.2 Definition of P&F Configurations

We provide three approaches to define a P&F configuration with TeeTime. All of them are based on the framework architecture described in Section 8.2. The most verbose approach is the object-oriented, programmatic one which uses the syntax and semantics of the underlying programming language. An alternative approach employs an internal DSL similar to the one for writing stage tests (cf. Section 10.1). The last approach uses an external DSL and allows to generate the configuration into arbitrary target programming language syntax. In the following, we describe each approach in detail.

Listing 10.5 shows an example configuration defined with the programmatic approach using Java. This P&F configuration processes a set of sentences which contain words separated by whitespaces. In Line 1, we declare the `WordCounterConfiguration` by extending the class `Configuration` provided by TeeTime. We declare the stages in Line 5-9 and their interconnections in Line 11-14. First, an initial list of sentences is passed to the stage `DataSource` (Line 5) which outputs each sentence one by one to its output port (Line 11).

[2] `https://github.com/teetime-framework/TeeTime/tree/master/src/test/java/teetime/stage`

```
1  public class WordCounterConfiguration extends Configuration {
2
3      public WordCounterConfiguration(final List<String> sentences) {
4          final DataSource<String> producer = new DataSource<>(sentences);
5          final ToLowerCase toLowerCase = new ToLowerCase();
6          final SentenceSplitter splitter = new SentenceSplitter();
7          final WordCounter wordCounter = new WordCounter();
8          final Printer<CountingMap<String>> printer = new Printer<>();
9
10         this.connectPorts(producer.getOutputPort(), toLowerCase.
               getInputPort());
11         this.connectPorts(toLowerCase.getOutputPort(), splitter.
               getInputPort());
12         this.connectPorts(splitter.getOutputPort(), wordCounter.
               getInputPort());
13         this.connectPorts(wordCounter.getOutputPort(), printer.
               getInputPort());
14     }
15
16 }
```

Listing 10.5. An example configuration in TeeTime defined with the programmatic approach in Java.

A less verbose approach is presented by Listing 10.6. It makes use of an internal domain-specific language to connect ports in a shorter and more concise way for common scenarios. The DSL is supplied as a library in Java and allows to build linear pipelines by calling the methods `from(..)`, `to(..)`, and `end(..)`. For this purpose, the stages have to satisfy some requirements. The method `from(..)` accepts a stage which provides an output port accessible by `getOutputPort()`. The method `to(..)` accepts a stage which provides an input port and an output port accessible by `getInputPort()` and `getOutputPort()`, respectively. The method `end(..)` accepts a stage which provides an input port accessible by `getInputPort()`.

If the stages are not used elsewhere in the configuration, they can also be passed directly to the three methods of the DSL. In this way, the configuration can be written even shorter. Listing 10.7 shows the result of this refactoring. Either way, the DSL introduces a new abstraction, namely the concept of a linear pipeline, which avoids connecting ports by the application developer. Previously, the application developer could connect two ports with each other which are compatible according to their types.

```
1  public class WordCounterConfiguration extends Configuration {
2
3    public WordCounterConfiguration(final List<String> sentences) {
4      final DataSource<String> producer = new DataSource<>(sentences);
5      final ToLowerCase toLowerCase = new ToLowerCase();
6      final SentenceSplitter splitter = new SentenceSplitter();
7      final WordCounter wordCounter = new WordCounter();
8      final Printer<CountingMap<String>> printer = new Printer<>();
9
10     super.from(producer).to(toLowerCase).to(splitter).to(wordCounter).end
           (printer);
11   }
12 }
```

Listing 10.6. The example configuration from Listing 10.5 defined with our internal configuration DSL.

```
1  public class WordCounterConfiguration extends Configuration {
2
3    public WordCounterConfiguration(final List<String> sentences) {
4      super.from(new DataSource<>(sentences))
5          .to(new ToLowerCase())
6          .to(new SentenceSplitter())
7          .to(new WordCounter())
8          .end(new Printer<>());
9    }
10 }
```

Listing 10.7. The example configuration from Listing 10.6 defined with our internal configuration DSL using an even shorter notation.

However, ports could still be connected in a wrong way. For example, one single output port could be connected more than once with different input ports. The DSL abstracts from this issue and thus represents a less error-prone approach for building a configuration.

The most compact approach is presented by Listing 10.8. It makes use of an external DSL to declare not only ports, but whole configurations in a shorter and more concise way. More specifically, our DSL allows to declare the configuration class and the configuration constructor by means of a single statement (see Line 1). Moreover, stages are instantiated implicitly such that constructor calls, like `new` in Java, do not need to be written anymore (see Line 2-6). Furthermore, our DSL allows to build pipelines by

```
1 WordCounterConfiguration(List<String> sentences) {
2   DataSource<String> producer(sentences)
3   ToLowerCase toLowerCase()
4   SentenceSplitter splitter()
5   WordCounter wordCounter()
6   Printer<CountingMap<String>> printer()
7
8   producer->toLowerCase->splitter->wordCounter->printer
9 }
```

Listing 10.8. The example configuration from Listing 10.5 defined with our external configuration DSL.

connecting stages with the arrow symbol `->` (see Line 8).

We recommend to use this approach if a generator for the target programming language and a corresponding IDE plugin is available. Otherwise, we recommend to use the internal DSL and the programmatic approach in a hybrid way. On the one hand, the internal DSL is good at defining linear P&F structures. On the other hand, the programmatic approach is flexible enough to define branches and loops.

We provide the grammar and an associated Eclipse plugin for the external DSL as open source.[3] In this way, one can easily make extensions and adaptations to the DSL. For example, Listing 10.9 shows the custom `active` keyword necessary for the parallel push/pull scheduling (see Section 9.1.1). This keyword declares the associated `distributor` as an active stage. In the case of a branch and, especially, of a loop, our DSL additionally provides an alternative notation for connecting ports. Listing 10.9 shows an example where the `distributor` is connected with the two stages `counter0` and `counter1` by using the variable names of the ports. For example, `newOutputPort` is generated to `getNewOutputPort()` and `inputPort` is generated to `getInputPort()`.

For declaring a distributed configuration, our DSL provides the keyword `Node(..)`. Listing 10.10 shows two different examples of its applicability. In Line 5, the `producer` is assigned to a node named "producer-node" whereas, in Line 6, the `consumer` is assigned to an anonymous node. In addition to that, our DSL allows to specify the network protocol to use between two

[3] `https://build.se.informatik.uni-kiel.de/teetime/teetime-distributed-dsl`

```
1   active Distributor<String> distributor()
2   Counter<String> counter0(), counter1()
3
4   distributor.newOutputPort->counter0.inputPort
5   distributor.newOutputPort->counter1.inputPort
```

Listing 10.9. TeeTime's configuration DSL also allows to declare a stage *active* and to connect stages based on particular ports.

```
1 ExampleDistributedConfiguration(){
2   InitialElementProducer producer()
3   CollectorSink consumer()
4
5   Node(producer-node) producer
6   Node consumer
7
8   producer-TCP->consumer
9 }
```

Listing 10.10. An example distributed configuration (excerpt) written with TeeTime's confguration DSL.

stages located on different nodes. In Line 8, the `producer` is declared to communicate with the `consumer` using the Transmission Control Protocol.

In summary, we provide three different kinds of notations for P&F configurations. Each of them has its own advantages in terms of flexibility, compactness, and portability. While the programmatic approach offers the most flexible notation, so offers the external DSL approach the most compact and portable notation. Although we only have a generator to Java so far, the latter approach provides the opportunity to generate configurations in many different other programming languages. The internal DSL approach, however, represents a compromise between flexibility and compactness, and thus serves as a bridge between the two other approaches. In particular, it does not require any additional tooling like a generator or an IDE plugin.

10.3 Debugging and Profiling of P&F Configurations

When developing software, it is crucial to have supporting tools and techniques which ease in debugging the software. For this reason, we equipped each stage in TeeTime with an individual logger (see Section 8.2.1). Depending on the configured log level, it protocols each step of the stage at runtime in order to help the developer in verifying the stage's behavior. However, during the development and the application of TeeTime in various different domains of applications (cf. Chapter 12), we noticed that this simple debugging support is not sufficient to identify performance bottlenecks and concurrency issues in P&F configurations. Thus, we integrated application performance monitoring into TeeTime and, based on that, built a live visualization for running P&F configurations as Eclipse Plugin. In Section 10.3.1 and Section 10.3.2, respectively, we describe both aspects in detail.

10.3.1 Application Performance Monitoring

Similar to the definition of P&F configurations in Section 10.2, we present a programmatic approach and a DSL-based approach to monitor the performance of TeeTime-based applications. Both of them have their advantages and disadvantages and thus cover different areas of application.

The programmatic approach relies on probes which are hard coded into the base stage of TeeTime. In this way, all of the stages, which are implemented with TeeTime, can automatically be monitored without any additional effort by the application developer. A flag allows to enable or to disable the monitoring for a particular execution. The goal of the programmatic approach is to differentiate and thus to measure the execution phase, the waiting phase, and the trigger phase of each stage.

A stage is in the execution phase if it is being executed and if it is not in any other phase. As soon as the stage accesses one of its ports, it could enter two alternative phases, namely the waiting phase or the trigger phase. It enters the waiting phase if the associated pipe implementation blocks or performs busy waiting. This situation can, for example, occur in the synchronized pipe implementation when it is currently empty or

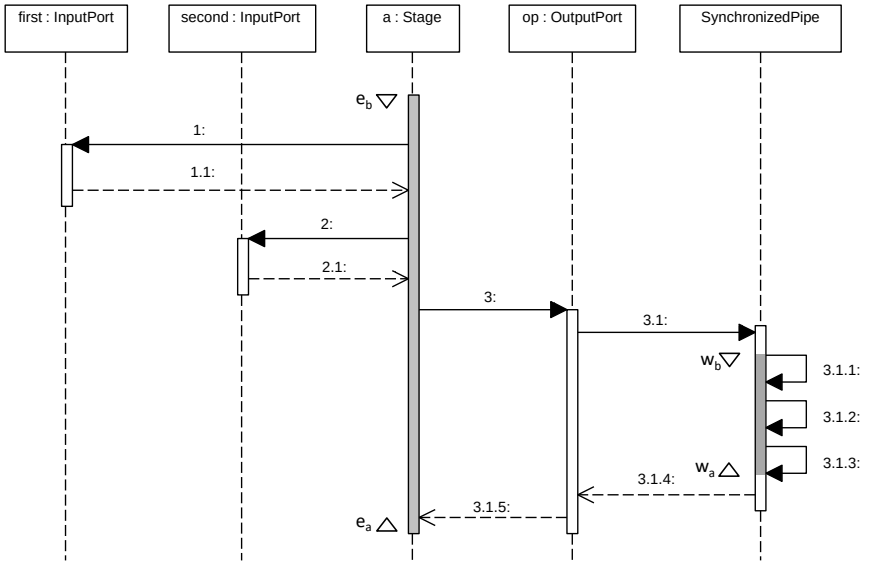


Figure 10.3. In this scenario, the buffer of the output pipe is full. For this reason, the execution of stage a leads to busy waiting three times in a row. During these transmission attempts, stage a is in the waiting phase. The triangles e_b and e_a represent the probe locations for the before-event (see subscript b) and, respectively, the after-event (see subscript a) of stage a's execution. The triangles w_b and w_a illustrate the probes to monitor the waiting phase.

full, respectively (see Section 8.2.3 on Page 69). Figure 10.3 illustrates this scenario. Alternatively, a stage enters the trigger phase if the associated pipe implementation immediately triggers the execution of the target stage. This situation typically occurs in the unsynchronized pipe implementation (see Section 8.2.3 on Page 68) and is shown in Figure 10.4.

With these three phases, it is possible to identify how long each stage has run and waited, respectively. The necessary probes are placed at the beginning and the end of the method `Stage.execute()` and wherever waiting phases can occur (see TeeTime's framework architecture in Figure 8.1 on Page 64). Figure 10.3 and 10.4 show the probe locations for two example

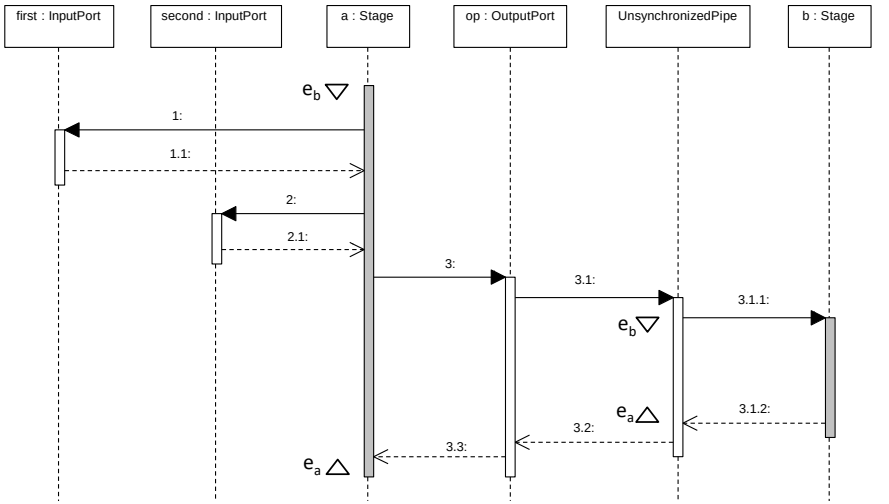


Figure 10.4. In this scenario, the execution of stage a triggers the execution of stage b. During that execution, a is in the trigger phase. Similar to Figure 10.3, the triangles e_b and e_a illustrate the probes to monitor the individual stage executions.

scenarios. Each probe stores the phase name, the current time stamp, and the executing thread in a stream of phase entry events. These events can then be used to compute the time spent in the execution phase, in the waiting phase, and in the trigger phase of each stage. Equation 10.3.1 to 10.3.3 show the corresponding calculation rules $e_t(s)$, $w_t(s)$, and $t_t(s)$ for a single stage execution s. The set $wphases(s)$ represents all of the waiting phases of s, while $succs(s)$ represents s's successor stages.

$$e_t(s) = \Big(e_a(s) - e_b(s)\Big) - w_t(s) - t_t(s) \tag{10.3.1}$$

$$w_t(s) = \sum_{wp_i \in wphases(s)} (w_a(wp_i) - w_b(wp_i)) \tag{10.3.2}$$

$$t_t(s) = \sum_{s_i \in succs(s)} \Big(e_a(s_i) - e_b(s_i)\Big) \tag{10.3.3}$$

Hence, the programmatic approach allows to measure not only the execution time, but also the waiting time of each stage. In this way, it enables a fine-grained analysis of concurrently running stages. An example application of the approach is shown by Pegler [Peg16]. He conducted a case study in which he compares two versions of TeeTime to identify the root cause of a performance loss.

As opposed to the programmatic approach, the DSL-based approach relies on probes which are instrumented via aspect-oriented programming (AOP). This allows us to also instrument applications which were not written with TeeTime. Moreover, we do not even need access to the source code to monitor the application's behavior. The approach only requires a declaration of components including their input and output ports. However, many applications are written in programming languages which do not directly support these concepts. For this reason, the approach provides a DSL which allows to describe such applications by *components* and *ports*. In this DSL, a component is declared by a unique name and by a list of port declarations. A port declaration includes the port's type, the port's unique name, and the port's representation in the target programming language. The port's type is specified by the keyword `in` for an input port, or by the keyword `out` for an output port. The port's name is surrounded by quotation marks and must be unique in the context of the associated component. The port itself is represented by a method of the application. This method is indicated by its fully qualified name in the target programming language. Listing 10.11 shows an example instance of the DSL which illustrates the instrumentation of three components. This example also clarifies that the DSL is not aware of TeeTime in general or stages in particular. Instead, the DSL uses the more general notion of a component which enables the instrumentation of arbitrary component-based applications.

In addition to the component declarations, it is also possible to declare which ports should be connected with each other. The corresponding notation of the DSL is shown by example in Line 15 and 16. Although the indication of port connections is not necessary for the instrumentation, it allows to visualize them in our two live visualizations described in Section 10.3.2.

```
1  FileHandler {
2    in "ip" fileHost.plugins.encrypt.FileHandler.open(fileHost.
         plugins.encrypt.data.File)
3    out "op" fileHost.plugins.encrypt.FileHandler.send(fileHost.
         plugins.encrypt.data.FileWrapper)
4  }
5
6  FileEncrypter {
7    in "ip" fileHost.plugins.encrypt.FileEncrypter.accept(
         fileHost.plugins.encrypt.data.FileWrapper)
8    out "op" fileHost.plugins.encrypt.FileEncrypter.saveFile(
         fileHost.plugins.encrypt.data.EncryptedFile)
9  }
10
11 FSWriter {
12   in "ip" fileHost.plugins.encrypt.FSWriter.save(fileHost.
         plugins.encrypt.data.EncryptedFile)
13 }
14
15 FileHandler.op -> FileEncrypter.ip
16 FileEncrypter.op -> FSWriter.ip
```

Listing 10.11. Example components declaration written with the instrumentation DSL.

The main use case of the DSL is the generation of probes which collect runtime data in terms of components and their communication between each other. The result artifacts comprise three probes and an associated configuration file. One probe records the start time while another probe records the end time of a component's execution. The last probe records the exception if the execution has failed. The configuration file associates the probes with the ports by using the declared method names. These generated artifacts then serve as the basis for the two new component-based visualizations described in Section 10.3.2. So far, the DSL is not intended to record data from a lower level of abstraction such that classes, fields, and

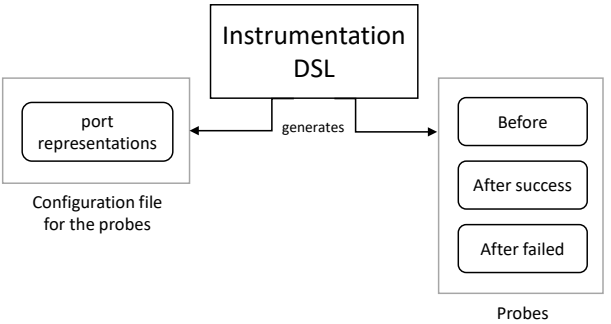


Figure 10.5. The whole probe generation process of the DSL-based instrumentation approach.

methods can be monitored.

Currently, the DSL assumes that an execution of a component is triggered by its input port and ends at its output port. Other or additional execution paths are not instrumented and thus cannot be monitored. Furthermore, it generates probes only in Java so far. The probes are implemented as aspects with the AOP framework AspectJ [Asp] and produces records defined with Kieker's instrumentation record language (IRL). [JW16] Figure 10.5 illustrates the whole probe generation process.

Hence, the DSL-based approach is less intrusive, has a more compact notation, and allows to monitor component-based applications. Besides the constraints mentioned above, it can however not distinguish waiting times from execution times. An example application of the approach is shown by Tavares de Sousa [Tav16]. He conducted a case study in which he instruments a TeeTime-based application with the DSL-based approach. The data collected in this way is then used to visualize the runtime behavior of the application.

10.3.2 Live Visualization

Based on the runtime information which we collect with the DSL-based instrumentation approach from Section 10.3.1, we now present two new

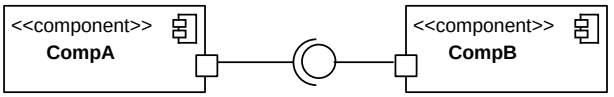


Figure 10.6. Example scenario which is used to introduce our two new visualizations: `ComponentA` uses `ComponentB`.

visualizations for the runtime behavior of component-based applications. Both of them support in identifying performance issues and concurrency issues. They do so while the application is running, i.e., both visualizations are live visualizations. They serve as a complement to our 3D visualization [WWF+13] for identifying and analyzing bottlenecks and deadlocks.

We introduce our visualizations by means of the example scenario shown in Figure 10.6. We intend to visualize the runtime behavior of the components `CompA` and `CompB`. For this purpose, we define these components including their ports and interconnection with the instrumentation DSL in order to let it instrument them. Afterwards, when executing the components, we can watch their current and past behavior. Figure 10.7 shows the components at a particular point in time using our first visualization: the execution behavior diagram. This diagram represents the components similar to Figure 10.6. Large boxes illustrate components and small boxes illustrate ports. The interconnection is drawn as an arrow to represent the data flow direction. The new feature of this diagram is the usage of different colors and of a timeline. Components are colored according to a mapping specified in advance. The mapping assigns a color to a range of values of a particular metric. Example metrics are the component's execution time or the number of processed elements within a given time interval. Both the color mapping and the metric are defined by the user. For example, Figure 10.7 colors `CompA` with a higher saturation than `CompB` because `CompA` ran longer than `CompB` in the currently selected time interval. A similar color encoding is used by Marwede et al. [MRH+09] to identify performance anomalies. The time interval is displayed in red on the timeline at the bottom. The diagram shows the components' behavior either from the current point in time, or from a past point in time. Figure 10.8 shows the timeline representation in more detail. Both the time interval and the offset can be

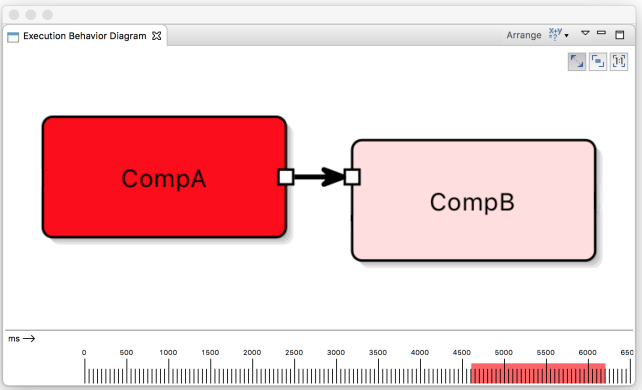


Figure 10.7. The execution behavior diagram showing the example application from Figure 10.6.

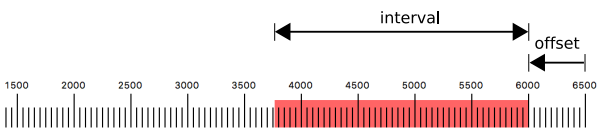


Figure 10.8. An example timeline of the execution behavior diagram.

adjusted by the user as desired. In this way, the behavior of the components can be analyzed in terms of an arbitrary metric at every point in time.

Figure 10.9 shows our second visualization: the timing behavior diagram. As opposed to the execution behavior diagram, this diagram represents the individual executions of the components in a way similar to a UML sequence diagram. Components are displayed side by side at the top. Their individual executions are illustrated as bars. In this way, the user can read off the start and the end of an execution by means of the timeline shown at the left side. The exact values of an execution are displayed when the mouse is hovered over the execution. During the execution of the components, the timing behavior diagram automatically scrolls down

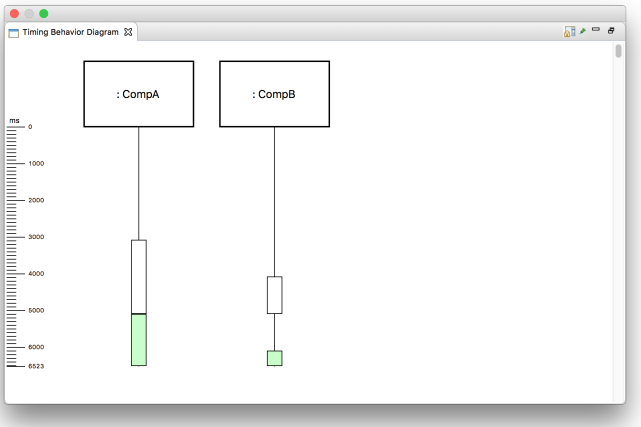


Figure 10.9. The timing behavior diagram showing the example application from Figure 10.6.

to the current point in time and colors all of the components which are currently being executed in green. Thus, the timing behavior diagram highlights simultaneous executions of different components and, in this sense, supports the identification of concurrency issues.

Both diagrams share almost the same live processing architecture. The associated data flow is shown by Figure 10.10. The black arrows illustrate the data flow which is common to both visualizations. In the first step, the runtime data collected from the instrumented application is written to a record log. Afterwards, the data is transformed into a runtime model. For example, the execution time of a component execution is computed based on the start time and the end time. If the end time is not yet available, because the component is currently being executed, the end time is set to the current time. The execution behavior diagram then queries the runtime model and stores the query result in a temporary runtime model. This is done according to the currently selected time interval in the timeline. Subsequently, the temporary runtime model is transformed into the view

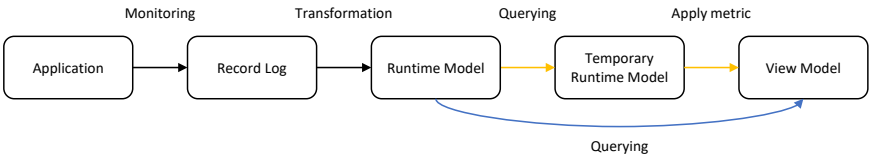


Figure 10.10. Data flow of our live visualizations. Black arrows illustrate the common data flow; colored arrows illustrate the data flow specific to the individual visualizations. The yellow path represents the data flow of the execution behavior diagram; the blue path represents the data flow of the timing behavior diagram.

model by applying the user-defined metric and color mapping. The result is finally visualized by the behavior execution diagram. Conversely, the timing behavior diagram queries the runtime model without any intermediate transformations since it exclusively visualizes the current activity of each component. For more information about technical details, we refer to the work of Tavares de Sousa [Tav16].

For the instrumentation process, we use our DSL described in Section 10.3.1 in combination with Kieker [HWH12] and AspectJ [Asp]. Furthermore, we also use Kieker for writing and storing the collected runtime data. For the common runtime model as well as for the view model of both visualizations, we use standard Java classes. Since the transformation is composed of multiple consecutive steps, we use TeeTime for its realization. The visualizations are implemented as a single Eclipse plugin and utilize the visualization framework Klighd [SSH13]. The plugin is available as open source.[4]

[4] `https://build.se.informatik.uni-kiel.de/teetime/teetime-live-visualization-plugin`

Part III

Evaluation of TeeTime

Chapter 11

Language Independence Evaluation

We start the evaluation of TeeTime by examining its framework architecture for language independence. For this purpose, we consider our two reference implementations in Java and in C++. We show that both of them successfully realize the general language-agnostic framework architecture presented in Chapter 8, albeit they partly use different techniques for this. In the following, we consider their similarities and compare their differences by taking a closer look at their implementations. Moreover, we provide both the Java implementation[1] and the C++ implementation[2] as open source.

11.1 Methodology

The goal of this evaluation is to show that TeeTime's framework architecture is not bound to the features of a particular programming language. As long as the language supports the object-oriented concepts as described by our architectural constraint CON_{oo} in Section 8.1 on Page 62, we make no further restrictions. Since Java and C++ fulfill these requirements, we were able to implement TeeTime in both languages using Java 8 and C++11. Below, we consider these two implementations in more detail to understand how they realize the individual framework entities described in Section 8.2. Hence, with this evaluation, we provide answers to our research question Q1 (*How to Model Arbitrary P&F Architectures?*) mentioned in Section 7.2.

[1] https://github.com/teetime-framework/TeeTime

[2] https://github.com/teetime-framework/TeeTime-Cpp

Each of both implementations represents a different programming language as covered by the metric M1.2. In Section 11.2, we provide a first, rough overview of both implementations by providing descriptive statistics about their size and their complexity. Afterwards, in Section 11.3, we describe their state of development by comparing which entities they have realized with which techniques.

11.2 Statistical Overview

Table 11.1 shows a descriptive statistics about the size and the complexity of our two TeeTime reference implementations. It shows that the Java implementation is more complex than the C++ one with regard to the number of source files, the overall size on the file system, and the lines of code. This larger complexity has several reasons. On the one hand, the Java implementation serves as the main experimentation platform for us. Unlike the C++ implementation, it implements all of the concepts which we present in Part II. On the other hand, it provides more stages for reuse. For this reason, it also contains a larger number of unit, integration, and performance tests. In Section 11.3, we discuss the similarities and the differences of both implementations in more detail.

11.3 State of Development

Table 11.2 shows TeeTime's framework entities and its associated entities in the Java reference implementation. Since each TeeTime entity has a Java equivalent, the Java implementation realizes all of the entities which our framework architecture prescribes. Entities with type parameters, like `Port<T>`, are implemented as generic types. However, due to constraints in Java, type parameters in generic types are only accessible at compile-time. In order to enable runtime validation of port connections anyway, we additionally store the type parameter `T` in the field `type` of `Port<T>` (see TeeTime's framework architecture in Figure 8.1 on Page 64). For achieving a high throughput and a high scalability in multi-threaded scenarios, it is important that the synchronized queue has a low and constant overhead.

Table 11.1. Descriptive statistics about the size and the complexity of our two TeeTime reference implementations (taken from the master branch on 08.01.2019).

Metric	Result for Java impl.	Result for C++ impl.
Source files	392[a]	102[b]
Overall size of source files	943,465 bytes[a]	698,502 bytes[b]
Lines of code[c]	14822[d]	12448[d]
↳Type declarations	50[e]	134[f]
↳Stage declarations	59[g]	12[g]
↳Tests	247[h]	36[i]

[a] `Get-ChildItem -Recurse -File -Include "*.java" | Measure-Object -Property Length -Sum`
[b] `Get-ChildItem -Recurse -File -Include "*.h","*.cpp" | Measure-Object -Property Length -Sum`
[c] Without comments and without blank lines
[d] `Measured with cloc from https://github.com/AlDanial/cloc`
[e] Measured with the Eclipse plugin metrics2
[f] Measured with cccc from `http://cccc.sourceforge.net`
[g] `Non-abstract stages measured by hand`
[h] `Get-ChildItem -Recurse -File -Include "*.java" | Select-String -pattern "@Test" -CaseSensitive -SimpleMatch | Measure-Object`
[i] `Get-ChildItem -Recurse -File -Include *.h,*.cpp | Select-String -pattern "TEST(" -CaseSensitive -SimpleMatch | Measure-Object`

We use the lock-free `SpScArrayQueue` of the JCTools library[3] which is similar to the queues described by [ADK+12; GMV08; WZT+13]. This lock-free queue imposes only a very low overhead by using light-weight synchronization mechanisms, such as memory barriers. Moreover, it achieves a constant overhead by placing data of different threads on different cache lines to minimize cache contention. Finally, the queue allows access from a single producer and a single consumer only. In this way, it avoids expensive inter-producer and inter-consumer coordination. For achieving similar performance in distributed scenarios, we use the message-driven, actor-oriented framework Akka [Lig]. It offers facilities for an efficient network transmission and for fault tolerance. In particular, it utilizes the message serializer `AkkaSerializer` from Twitter which is based on the Kryo library.[4] Kryo is well-known for its efficient object serialization algorithm. The Java implementation supports arbitrary schedulers. Such a scheduler only needs

[3] `https://github.com/JCTools/JCTools`
[4] `https://github.com/EsotericSoftware/kryo`

Table 11.2. TeeTime's framework entities and its associated entities in the Java reference implementation.

TeeTime Entity	Java Equivalent
Stage	teetime.framework.AbstractStage
Port<T>	teetime.framework.AbstractPort<T>
↳OutputPort<T>	teetime.framework.OutputPort<T>
↳InputPort<T>	teetime.framework.InputPort<T>
Pipe	teetime.framework.IPipe<T>
↳UnsychronizedPipe	teetime.framework.pipe.UnsynchedPipe<T>
↳SychronizedPipe	teetime.framework.pipe.BoundedSynchedPipe<T>
↳SpSc Queue	org.jctools.queues.SpscArrayQueue<E>
↳DistributedPipe	teetime.framework.pipe.DistributedPipe
↳Actor	akka.actor.UntypedActor
↳Serialization	com.twitter.chill.akka.AkkaSerializer
Configuration	teetime.framework.Configuration
Execution	teetime.framework.Execution
Scheduler	teetime.framework.TeeTimeScheduler
↳Push/Pull Scheduler	teetime.framework.scheduling.pushpullmodel.PushPullScheduling
↳Task Pool Scheduler	teetime.framework.scheduling.globaltaskpool.GlobalTaskPoolScheduling
Traverser	teetime.framework.Traverser
↳Listener	teetime.framework.ITraverserVisitor
CompositeStage	teetime.framework.CompositeStage
Task Farm Stage (TFS)	teetime.stage.taskfarm.StaticTaskFarmStage<I, O, T>
Adaptive TFS	teetime.stage.taskfarm.DynamicTaskFarmStage<I, O, T>

to implement the Java interface `TeeTimeScheduler`. Then, it can be passed together with a `Configuration` to an `Execution`. Currently, the Java implementation provides the two schedulers presented in Section 9.1.1 and 9.1.2. Furthermore, it includes the task farm stage and its associated self-adaptive manager introduced in Section 9.2.

Table 11.3 shows TeeTime's framework entities and its associated entities in the C++ reference implementation. As some C++ equivalents are not yet available, the C++ implementation provides only a subset of the features of its Java pendant. For example, it does not support distributed P&F architectures, so far. Moreover, it only implements the Parallel Push/Pull Scheduler (see Section 9.1.1). Furthermore, it is limited to primitive stages. However, these limitations do not result from C++ itself, but are just a matter of implementation effort. In addition to that, some TeeTime entities are implemented by the same C++ type. For example, `teetime.Configuration`

Table 11.3. TeeTime's framework entities and its associated entities in the C++ reference implementation.

TeeTime Entity	C++ Equivalent
Stage	teetime.AbstractStage
Port<T>	teetime.AbstractPort
↳OutputPort<T>	teetime.OutputPort<T>
↳InputPort<T>	teetime.InputPort<T>
Pipe	teetime.Pipe<T>
↳UnsychronizedPipe	teetime.UnsynchedPipe<T>
↳SychronizedPipe	teetime.SynchedPipe<T>
↳SpSc Queue	teetime.SpscValueQueue<T>
↳DistributedPipe	n/a
↳Actor	n/a
↳Serialization	n/a
Configuration	teetime.Configuration
Execution	teetime.Configuration
Scheduler	n/a
↳Push/Pull Scheduler	teetime.Configuration
↳Task Pool Scheduler	n/a
Traverser	n/a
↳Listener	n/a
CompositeStage	n/a
Task Farm Stage (TFS)	n/a
Adaptive TFS	n/a

realizes the configuration, the execution, and the scheduler. This is, again, not due to C++ limitations, but rather for simplicity reasons. As soon as further schedulers are added to the C++ implementation, we will completely adapt it to our TeeTime framework architecture. For realizing entities with type parameters, the C++ implementation uses class templates. Hence, if a configuration includes two differently parameterized `OutputPorts`, then the compiler generates a new class for each of them. Since the type of the port is, therefore, directly represented in the generated class, the C++ implementation does not require the `type` field in the `Port<T>` entity. High throughput is ensured by an own implementation of a single producer/single consumer array queue. The design of the queue is again based on the queues described by [ADK+12; GMV08; WZT+13]. However, we further optimized the performance by utilizing C++'s move semantics[5], custom

[5] `https://en.cppreference.com/w/cpp/language/move_constructor`

memory alignment[6], and efficient atomics[7]. These features are not yet available in Java. Thus, the C++ implementation achieves a higher performance than the Java one. For a performance comparison of state-of-the-art queue implementations and our one, we refer to the work of Ohlemacher [Ohl16]. In particular, it shows that our C++ queue implementation is the fastest in most scenarios. For a corresponding evaluation of our Java-based queue, we refer to Section 13.1.2. Although the C++ implementation currently provides only one scheduler, this scheduler has the additional ability to set CPU affinity for each thread. In this way, the scheduler allows to place different threads on different cores. If the underlying hardware has more than one CPU, threads can also be placed on the same CPU or on different CPUs. Again, this feature is not yet available in Java and thus further increases the distance to the performance of the Java implementation. For a performance comparison of our two implementations including different CPU affinity configurations, we refer to the work of Ohlemacher [Ohl16].

11.4 Summary

This evaluation shows that our TeeTime framework architecture is not bound to the features of a particular programming language. Although our two implementations differ in some technical details, both implement the same, general, language-independent framework architecture introduced in Chapter 8. We leave missing implementations of C++ entities as future work. From now on, we use the Java implementation in all of the following evaluations for consistency.

[6] https://en.cppreference.com/w/cpp/language/alignas
[7] https://en.cppreference.com/w/cpp/atomic

Chapter 12

Feasibility Evaluation

In this chapter, we evaluate the feasibility of our P&F framework TeeTime. We conduct several case studies which altogether cover both various technical and various domain-specific aspects. They show that TeeTime can model and execute several different P&F architectures, including linear pipelines, branches, and loops. Moreover, they show that TeeTime can be employed in several different fields of application. The entities and data associated with our case studies can be found in our Git repository.[1]

Besides these case studies, research projects, such as ExplorViz[2], iObserve [HHJ+13], and RadarGun [HWH17; Bar18], as well as some Github projects[3] make also use of TeeTime.

12.1 Case Study: Trace Reconstruction

In this case study, we use a P&F architecture built with TeeTime that contains linear pipelines, branches, and composite stages. It reconstructs and aggregates program traces from monitoring events in order to generate various graphs represented in different formats, such as DOT[4] or GraphML.[5]

Hence, this case study has two goals: one technical and one domain-specific goal. The technical goal is to show that TeeTime is able to model and to execute linear, branched, and composite P&F structures. The domain-specific goal is to show that TeeTime can be applied to the domain of

[1] https://build.se.informatik.uni-kiel.de/chw/phd-case-studies

[2] https://www.explorviz.net

[3] https://github.com/teetime-framework/TeeTime/stargazers

[4] https://www.graphviz.org/doc/info/lang.html

[5] http://graphml.graphdrawing.org

```
1 $0;1412763178835942258;1.9;KIEKER-SINGLETON;SE;1;false;0;NANOSECONDS;1
2 $1;1412763178834633375;3881283897249497088;1;<no-session-id>;SE
      ;3881283897249497088;-1
3 $2;1412763178849813012;1412763178849798259; 3881283897249497088;0;public
      static void kieker.examples.monitoring.aspectj.BookstoreStarter.main
      (java.lang.String[]);kieker.examples.monitoring.aspectj.
      BookstoreStarter
4 ...
5 $3;1412763178864687960;1412763178864684753; 3881283897249497088;5;public
      static void kieker.examples.monitoring.aspectj.BookstoreStarter.main
      (java.lang.String[]);kieker.examples.monitoring.aspectj.
      BookstoreStarter
```

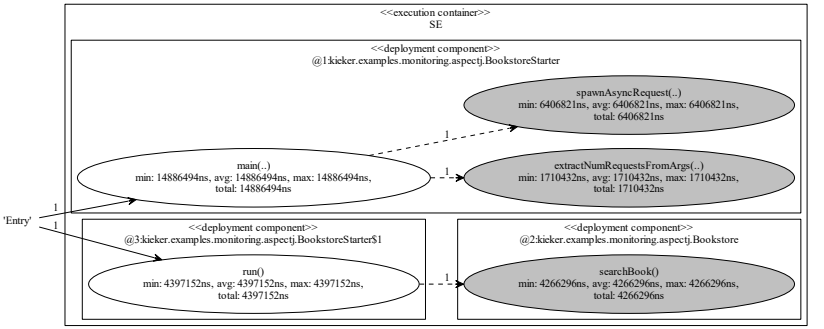
Listing 12.1. Excerpt of an example Kieker log file. Each line represents a monitored event whose attributes are separated by semicolons.

program trace reconstruction.

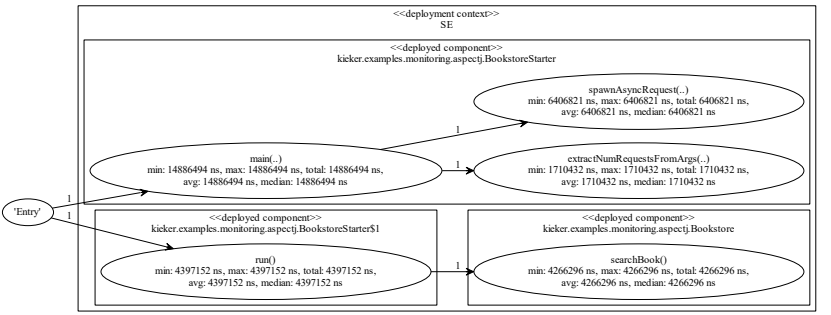
We study the P&F architecture and its execution of the trace reconstruction and visualization approach performed by Kieker's analysis component. [Kie] As input, the component receives a path to a directory which contains log files recorded by Kieker's monitoring component. As output, it produces multiple trace graphs and an aggregated dependency graph of the monitored method invocations stored in the log files. An excerpt of the input and a corresponding dependency graph are shown by Listing 12.1 and Figure 12.1a, respectively.

The P&F architecture, which we consider below, represents one of multiple different, representative P&F architectures covered by the metrics M1.1 and M3.1. Hence, with this case study, we provide answers to our research questions Q1 (*How to Model Arbitrary P&F Architectures?*) and Q3 (*How to Execute Arbitrary P&F Architectures?*) mentioned in Section 7.2.

Methodology We show that this case study fulfills the technical goal by providing a new TeeTime-based P&F architecture of Kieker's trace reconstruction approach. Since the architecture and thus the implementation represent an algorithm from the domain of program trace reconstruction, we simultaneously show with this implementation that the domain-specific goal is also fulfilled. Our procedure is as follows. We let a team of two students reimplement the approach as a TeeTime-based P&F architecture.



(a) Output of Kieker's trace reconstruction approach. It represents an operation dependency graph reconstructed from the input shown in Listing 12.1.



(b) Output of Kieker's trace reconstruction approach implemented with TeeTime. It equals almost completely the output shown in Figure 12.1a.

Figure 12.1. Output of Kieker and TeeTime in comparison. Both graphs were generated from the input shown in Listing 12.1.

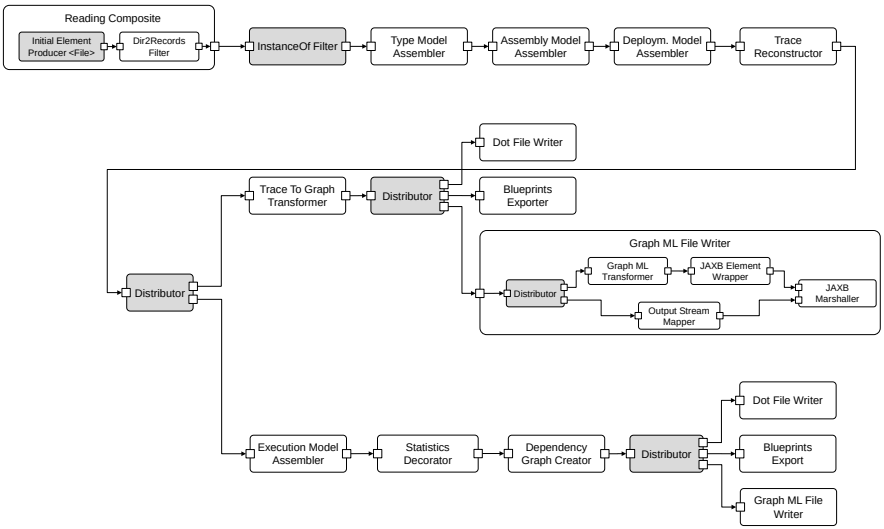


Figure 12.2. The P&F architecture used in the case study comprising linear, branched, and composite P&F structures. Stages from the domain of program trace reconstruction are illustrated in white.

Subsequently, we let them execute the architecture with the input of monitoring logs shown by Listing 12.1. We verify the execution by comparing the output graph with the one shown by Figure 12.1a. Moreover, we provide the input, the implementation, and the output as artifacts at our Git repository for an in-depth inspection for the interested reader.

Results and Discussion Figure 12.2 shows a schematic view of the new TeeTime-based P&F implementation. It consists of 19 general (gray) and domain-specific (white) stages where two of them are composite. The `Reading Composite` consists of two stages and the `Graph ML File Writer` consists of five stages. Most of the stages are connected with each other in a row such that they form linear pipelines. The remaining stages either split the data flow into two and three branches (see the `Distributors`) or join the branches again (see the JAXB Marshaller). Hence, this TeeTime-based

P&F implementation shows that TeeTime is able to model linear pipelines, branches, and composite stages.

A corresponding execution of this implementation produces the output graph shown by Figure 12.1b. It mostly equals the output graph of Kieker shown by Figure 12.1a. We only observe a few visual differences which are, however, intended by the author. In addition, the TeeTime-based P&F implementation adds the median execution time to each operation. Hence, the output graph shows that TeeTime is able to also execute linear pipelines, branches, and composite stages. Moreover, this case study shows that TeeTime can successfully be applied to the domain of program trace reconstruction.

12.2 Case Study: Quicksort

In this case study, we use a P&F architecture built with TeeTime that contains a loop. It receives a stream of integer arrays and sorts each of them in ascending order as it is described by the quicksort algorithm.

This case study has two goals: one technical and one domain-specific goal. The technical goal is to show that TeeTime is able to model and to execute feedback loops. The domain-specific goal is to show that TeeTime can be applied to the domain of sorting algorithms.

We study the P&F architecture and its execution of the iterative quicksort algorithm shown by Listing 12.2. The algorithm uses a stack to save the ranges of unsorted areas in the given array. After initializing the stack (Line 2-8), it enters a loop (Line 11). Within this loop, it partitions the elements in the array (Line 18) as long as the stack has no ranges left to partition.

The P&F architecture, which we consider below, represents one of multiple different, representative P&F architectures covered by the metrics M1.1 and M3.1. Hence, with this case study, we provide answers to our research questions Q1 (*How to Model Arbitrary P&F Architectures?*) and Q3 (*How to Execute Arbitrary P&F Architectures?*) mentioned in Section 7.2.

Methodology We show that this case study fulfills the technical goal by providing a new TeeTime-based P&F architecture of the quicksort algorithm.

```
1   void quickSort(int arr[]) {
2       // create the auxiliary stack
3       int stack[] = new int[arr.length];
4       // initialize top pointer of the stack
5       int top = -1;
6       // push initial range to the stack
7       stack[++top] = l;
8       stack[++top] = h;
9
10      // keep popping elements until stack is not empty
11      while (top >= 0) {
12
13          // pop the range of a yet unsorted area of the array
14          h = stack[top--];
15          l = stack[top--];
16
17          // set pivot element at its proper position
18          int p = partition(arr, l, h);
19
20          // If there are elements on left side of pivot,
21          // then push left side to stack
22          if ( p-1 > l ) {
23              stack[ ++top ] = l;
24              stack[ ++top ] = p - 1;
25          }
26
27          // If there are elements on right side of pivot,
28          // then push right side to stack
29          if ( p+1 < h ) {
30              stack[ ++top ] = p + 1;
31              stack[ ++top ] = h;
32          }
33      }
34  }
```

Listing 12.2. A Java implementation of the iterative quicksort algorithm.[a]

[a] https://www.geeksforgeeks.org/iterative-quick-sort

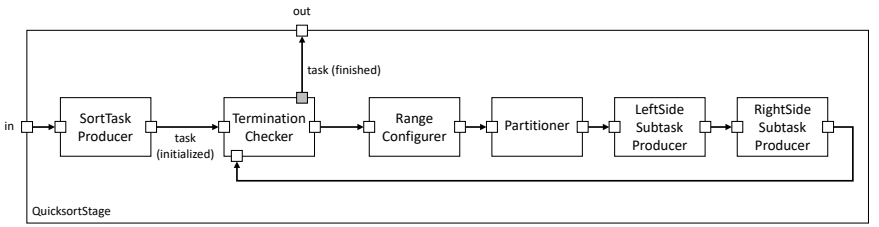


Figure 12.3. A P&F implementation of the iterative quicksort algorithm.

Since the architecture and thus the implementation represent an algorithm from the domain of sorting algorithms, we simultaneously show with this implementation that the domain-specific goal is also fulfilled. Our procedure is as follows. We let a student implement the algorithm as a TeeTime-based P&F architecture. We verify the execution by comparing the expected output with the actual one with unit tests. Furthermore, we provide the input, the implementation, and the output as artifacts at our Git repository for an in-depth inspection for the interested reader.

Results and Discussion Figure 12.3 shows a schematic view of the new TeeTime-based P&F implementation. The quicksort algorithm is represented as a composite stage with a single input port and a single output port. It consists of six domain-specific stages which implement the six code blocks shown in Listing 12.2. The `SortTaskProducer` creates a new sort task which holds the array and the associated stack. The `TerminationChecker` receives this task and passes it to the `RangeConfigurer` if the stack is not empty. The same applies for incoming subtasks which are produced by the corresponding producers in the feedback loop. If the stack of a task is empty, the task is passed to the second output port of the `TerminationChecker` (illustrated in gray) and thus leaves the composite quicksort stage. In this situation, the `TerminationChecker` additionally checks whether there are some further tasks left. If all tasks have been processed and if the input port which receives new tasks has received a termination signal, then the `TerminationChecker` passes the signal to the `RangeConfigurer`. In this way, the `TerminationChecker` eventually receives this signal again from its second

input port via the loop. Finally, the `TerminationChecker` can then terminate itself and pass the signal to its second output port (illustrated in gray). Hence, this implementation and an execution of the corresponding unit tests show that TeeTime has successfully been used to model and to execute a loop-based P&F architecture. Moreover, it shows an example application of TeeTime in the domain of sorting algorithms.

12.3 Industrial Case Study: Extraction of User Behavior Profiles

In this case study, we use two P&F architectures built with TeeTime that extract in combination the user behavior profiles of an industrial real-world application called b+m bAV-manager developed by the b+m Informatik AG. This application serves as an administration software for customer and calculation data in the field of company pension schemes. In cooperation with a third-party calculation engine, it creates expert opinions for several valuation and accounting regulations. Currently, the developers only know in an insufficient way how the users of the b+m bAV-manager work with the graphical user interface (GUI). For a planned modernization process of the GUI, this knowledge is, however, necessary in order to prioritize frequently used screenflows and to skip the migration of unused screenflows.

Hence, the primary goal of this case study is the identification of actual screenflows by extracting and visualizing user behavior models from the b+m bAV-manager. We refer to our previous work [DW16] for more information about the associated evaluation. However, in this section, we focus on the secondary goal of this case study, namely to show that TeeTime is applicable to the domain of user behavior profiles. We do not define an explicit technical goal for this case study since the P&F architectures, employed here, do not use any new P&F structures besides those already presented in Section 12.1. We study the P&F-based approach illustrated by Figure 12.4. It consists of the two P&F architectures already mentioned above. Both of them are implemented as a small Java application. They communicate with each other by a session log storage. As input, the first P&F architecture receives runtime records from the b+m bAV-manager via

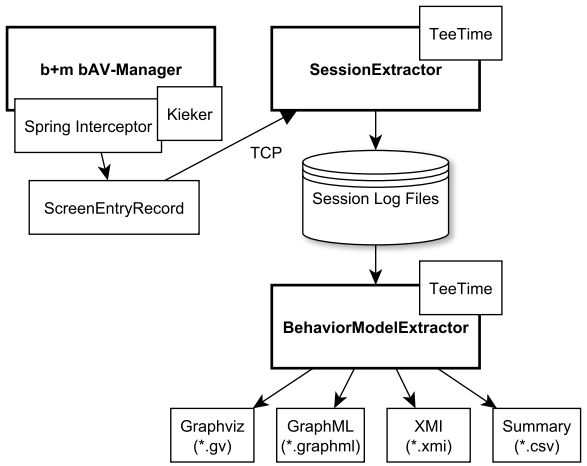
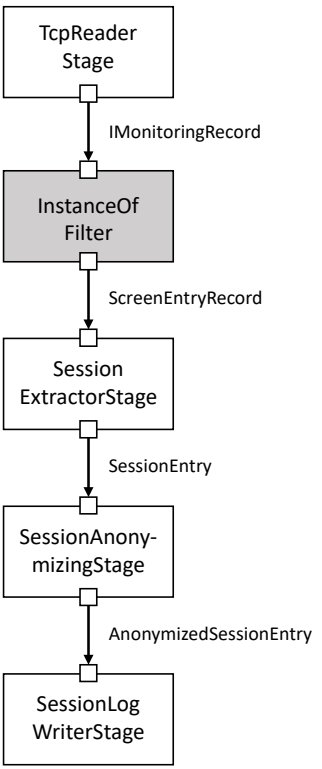


Figure 12.4. Overview of our P&F-based approach to extract and to visualize user behavior profiles.

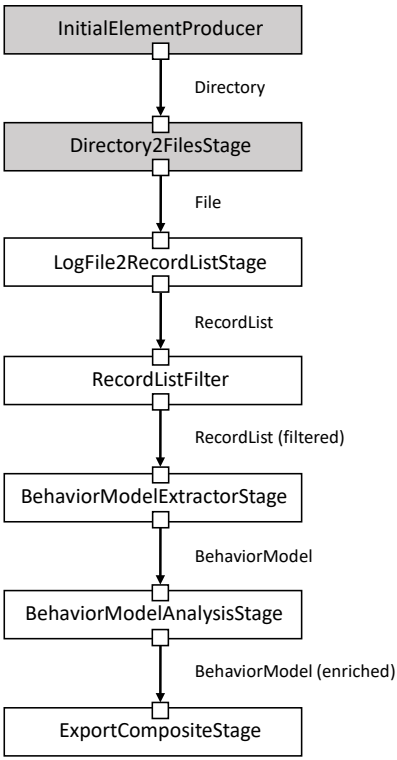
TCP using Kieker. By transforming these records in a pipeline of five stages, it outputs anonymized session logs of the bAV-manager users. Figure 12.5a shows the P&F architecture in more detail.

As input, the second P&F architecture receives the path to the session log directory. Then, it collects all of the session log files from that directory and passes each of them to the `LogFile2RecordListFilter`. This stage groups the entries of each incoming log file to a record list such that the subsequent `RecordListFilterStage` can filter individual entries on demand to focus on a specific area in the application or on a particular time interval. The next `BehaviorModelExtractorStage` uses the record list to build up a corresponding behavior model consisting of states (screens) and transitions (user activities). After performing some analyses by the `BehaviorModelAnalysisStage` to enrich the model with hierarchy information, the resulting model is exported to multiple file formats by the `ExportCompositeStage`.

The P&F architectures, which we consider below, represent two of multiple different, representative P&F architectures covered by the metrics M1.1



(a) The P&F architecture of the session extractor. Domain-specific stages are illustrated in white.



(b) The P&F architecture of the behavior model extractor. Domain-specific stages are illustrated in white.

Figure 12.5. The two P&F architectures used in the approach shown by Figure 12.4.

and M3.1. Hence, with this case study, we provide answers to our research questions Q1 (*How to Model Arbitrary P&F Architectures?*) and Q3 (*How to Execute Arbitrary P&F Architectures?*) mentioned in Section 7.2.

Methodology We show that this case study fulfills the domain-specific goal by providing a new TeeTime-based implementation for the approach illustrated by Figure 12.4. A visualization of an example application of the approach will show that TeeTime can not only model, but also execute P&F architectures in the domain of user behavior profiles. Our procedure is as follows. We let a student implement the approach with TeeTime. Afterwards, we let him instrument and subsequently start the GUI of the b+m bAV-Manager on a test server at the b+m Informatik AG. We then let five b+m employees (developers, architects, and project managers) perform 11 common business processes via the GUI. During the execution, the instrumented b+m bAV-Manager produces runtime data which is sent via TCP to the session extractor. From there on, the data is processed according to our P&F-based approach. We verify the proper execution by considering the output in the GraphML format. Furthermore, we provide the implementation as artifact at our Git repository for an in-depth inspection for the interested reader. Due to a confidentiality agreement, we are not allowed to provide the input and the output to the general public.

Results and Discussion We collected 53 session log files with 2381 user activities. Figure 12.6 shows the corresponding output. It represents an excerpt of a behavior model of the b+m bAV-Manager calculation process (dark gray) including its subprocesses (light gray) and screens (white). This GraphML-based visualization was generated by the TeeTime-based implementation of the approach shown in Figure 12.4. It shows that TeeTime has successfully been executed in the domain of user behavior profiles.

The P&F architecture of the session extractor uses the domain-specific stages `SessionExtractorStage`, `SessionAnonymizingStage`, and `SessionLogWriterStage`. These stages communicate with each other by producing and consuming the domain-specific entities `SessionEntryRecord`, `SessionEntry`, and `AnonymizedSessionEntry`. Similarly, the P&F architecture of the behavior

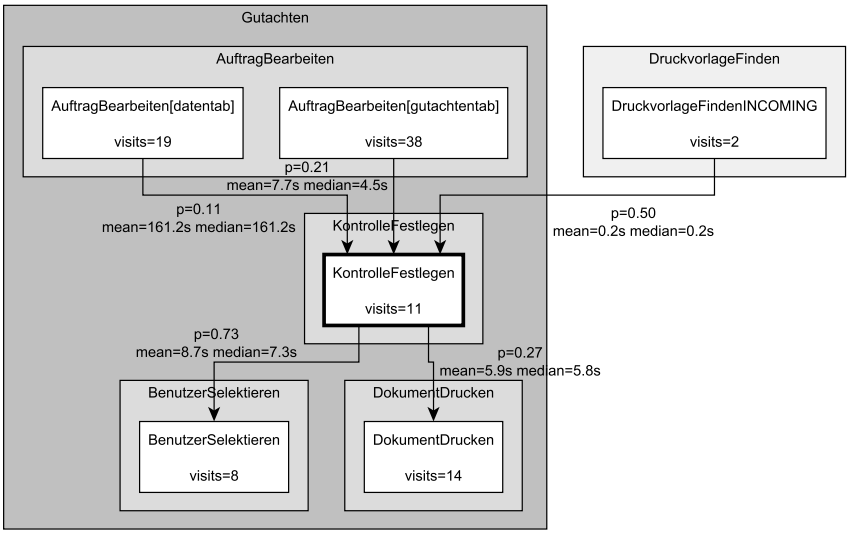


Figure 12.6. An excerpt of an example behavior model of the b+m bAV-Manager calculation process (dark gray) including its subprocesses (light gray) and screens (white). This GraphML-based visualization was generated by the TeeTime-based implementation of the approach shown in Figure 12.4.

model extractor uses domain-specific stages and entities. Hence, the implementations of both architectures show that TeeTime has successfully been used to model P&F architectures for the domain of user behavior profiles.

12.4 Case Study: Live Visualization of Performance Anomalies

In this case study, we use a P&F architecture built with TeeTime that performs live processing of monitoring data by interacting with a database. More specifically, it detects and visualizes performance anomalies of a monitored application at runtime.

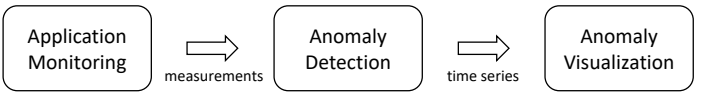


Figure 12.7. The general anomaly detection and visualization approach whose anomaly detection component should be implemented by a TeeTime-based P&F implementation.

Hence, this case study has two goals: one technical and one domain-specific goal. The technical goal is to show that TeeTime is able to perform live processing. The domain-specific goal is to show that TeeTime can be applied to the domain of performance anomaly detection.

We study the P&F architecture and its execution of the anomaly detection and visualization approach shown by Figure 12.7. This approach comprises three steps which are organized into the three components: application monitoring, anomaly detection, and anomaly visualization. The application monitoring component instruments a given application to collect the response times of service calls at runtime. Each measurement is directly passed to the anomaly detection component where it is compared with a forecasted measurement. The forecast measurement is computed based on the past measurements covered by a sliding window. This sliding window moves forward with new incoming measurements. In this way, a time series is recorded and stored in a database such that it can later be visualized by the anomaly visualization component. For more information about the approach itself, we refer to the work of Henning [Hen16].

The P&F architecture, which we consider below, represents one of multiple different, representative P&F architectures covered by the metrics M1.1 and M3.1. Hence, with this case study, we provide answers to our research questions Q1 (*How to Model Arbitrary P&F Architectures?*) and Q3 (*How to Execute Arbitrary P&F Architectures?*) mentioned in Section 7.2.

Methodology We show that this case study fulfills the technical goal by providing a new TeeTime-based implementation for the approach illustrated by Figure 12.7. A screenshot of the anomaly visualization in action will show that TeeTime can not only model, but also execute the approach. In

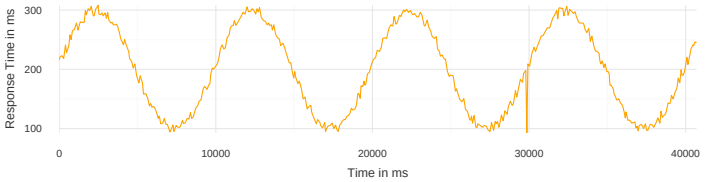


Figure 12.8. The scenario of this case study: a service call with a regularly oscillating response time which shows a performance anomaly at the 30th second.

this way, we simultaneously show that TeeTime fulfills the domain-specific goal as well. Our procedure is as follows. We let a student implement and execute the approach. In particular, we let him implement the anomaly detection component as TeeTime-based P&F architecture. Afterwards, we let him execute the approach such that it monitors a particular service call with a regularly oscillating response time. To show the anomaly detection, he induces a single performance anomaly at the 30th second where the response time rapidly drops from 200 to 100 ms. Figure 12.8 shows the corresponding workload curve. We verify the execution by comparing this workload curve with the approach's anomaly visualization. We expect that the visualization illustrates only one performance anomaly, namely the anomaly induced by the student. Furthermore, we provide the input, the implementation, and the output as artifacts at our Git repository for an in-depth inspection for the interested reader.

Results and Discussion Figure 12.9 shows a schematic view of the TeeTime-based P&F implementation. The anomaly detection component is represented as a composite stage with a single input and multiple, optional output ports. The stage continuously receives a measurement (1), updates its sliding window (2), forecasts an associated value (3), and determines the difference between the actual and the forecasted measurement indicated by an anomaly score (4). Afterwards, the current sliding window and the anomaly score are saved to the database (5). For each registered threshold filter, it outputs an anomaly if the anomaly score has exceeded or undercut the associated threshold (6). One of the filters is connected with the anomaly

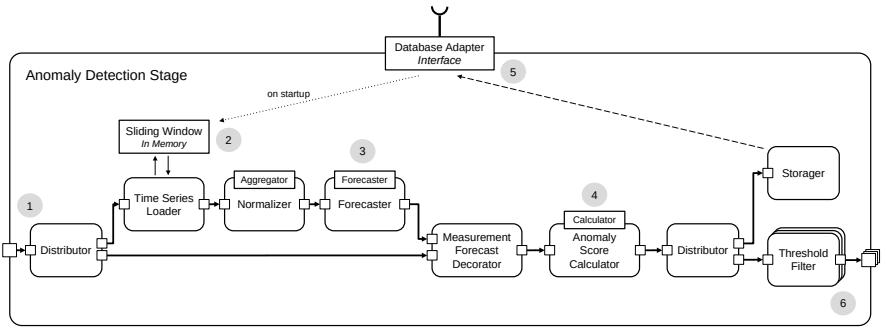


Figure 12.9. The composite stage which performs the anomaly detection.

visualization component such that the component can visualize an anomaly if one has been detected. Figure 12.10 shows the output of the anomaly visualization component for the given workload shown by Figure 12.8. The actual measurements are colored in orange, the forecasted values are colored in blue, and the anomaly scores are colored in red below. As the red triangle with the exclamation mark indicates, the visualization successfully illustrates the performance anomaly which was initially induced into the monitored application. Hence, TeeTime has successfully been used to model and to execute a live processing P&F architecture. Moreover, the architecture represents an example from the domain of performance anomaly detection.

12.5 Case Study: Distributed Execution

In this case study, we use a P&F architecture built with TeeTime that is distributed among multiple nodes. It generates random strings and passes them across two further nodes. Hence, the goal of this case study is to show that TeeTime is able to model and to execute distributed P&F architectures. This includes showing that the remote deployment, the remote execution, and the distributed communication between the worker nodes work as expected. Furthermore, this includes to show that fault tolerance in terms of a graceful termination works when a stage throws an exception or

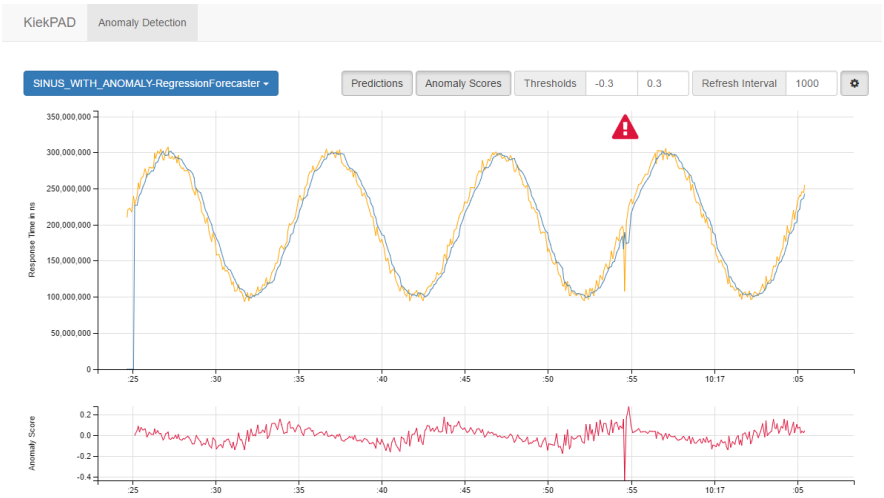


Figure 12.10. Screenshot of the anomaly visualization component from our scenario. The anomaly is correctly detected at point in time where the application has run 30 seconds.

a connection timeout occurs during the execution. We study the linear pipeline shown in Figure 12.11. It consists of five stages where three of them are instances of the `CPU Load generator` stage. The `Random String Generator` stage internally generates random strings with a configurable fixed size. For each string, it outputs an instance of a class which contains this string and a monotonically increasing counter index. The counter index is later used to verify that the string has arrived in order. On each incoming element, the `CPU Load Generator` stage generates CPU load for a configurable amount of cycles before forwarding the element without any modification to its output port. This stage simulates some work and thus serves as proxy for an arbitrary stage. The `End` stage has a counter attribute which is incremented for every element received. When the stage consumes an incoming element, it prints the current value of the counter attribute together with the counter index of the element to the console. In this way, it is possible to verify whether the stage receives the elements in the same order as produced

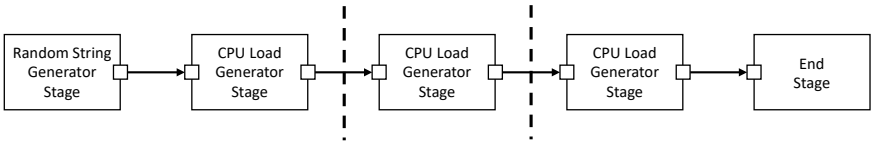


Figure 12.11. The linear pipeline architecture which should be distributed among three nodes as indicated by the two dashed lines.

by the `Random String Generator` stage. The dashed lines in Figure 12.11 indicate the locations where the pipeline should be distributed. Each part should then be deployed and executed on a dedicated node.

The corresponding distributed P&F architecture, which we present below, represents one of multiple different, representative P&F architectures covered by the metrics M1.1 and M3.1. Moreover, it uses multiple nodes such that it is additionally covered by the metric M3.2. Hence, with this case study, we provide answers to our research questions Q1 (*How to Model Arbitrary P&F Architectures?*) and Q3 (*How to Execute Arbitrary P&F Architectures?*) mentioned in Section 7.2.

Methodology We show that this case study fulfills the goal by modeling a corresponding distributed P&F architecture with TeeTime and by executing it on three different nodes. The setup of both the software and the hardware is described in detail by Echternkamp [Ech17]. Our procedure is as follows. We let a student implement, deploy, and execute the distributed P&F architecture. Thereby, the worker nodes record their relevant execution steps, such as their creation, their transmission of control messages and TeeTime signals, as well as their termination. In this way, we are able to reconstruct the temporal sequence and thus evaluate whether all steps were executed correctly and in the right order. To check whether each protocol works as expected, the distributed P&F architecture is executed once with TCP, once with UDP, and once with SSL over TCP.

For verifying the behavior in case of faults, the P&F architecture is executed another three times. The first time covers an exception in a TeeTime stage. An exception is thrown inside the `Random String generator` stage

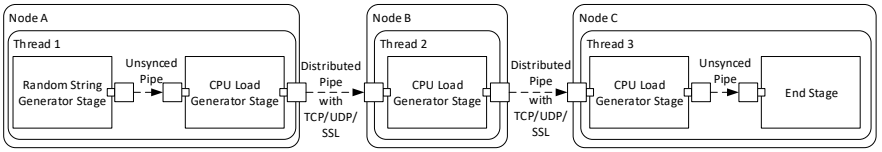


Figure 12.12. A schematic view of the TeeTime-based P&F implementation. It realizes the distributed version of the pipeline shown in Figure 12.11

after processing 500 strings. The second time covers an exception in the remote communication between the actors. A similar exception is thrown inside the actor of a distributed pipe after processing 500 strings. Finally, the third time covers a connection timeout to one of the nodes during the execution. For this purpose, a worker is killed manually.

As preparation, the Java reference implementation of TeeTime and the distributed P&F configuration are copied to four different nodes: one master node and three worker nodes. On each worker node, the associated worker actor is started by hand such that it can register and listen to the master as soon as the master is available. Then, the distributed P&F configuration is ready to be started from the master node. In addition to this feasibility evaluation, Echternkamp [Ech17] also evaluates the performance of the distributed P&F architecture. For more information on this performance evaluation, we refer to his associated work [Ech17].

Results and Discussion Figure 12.12 shows a schematic view of the TeeTime-based P&F implementation. Each of the three pipeline parts is deployed on a dedicated node and executed by a dedicated thread. Stages communicate with each other either via unsynchronized pipes, if they are deployed on the same node, or via distributed pipes, if they are deployed on different nodes. The corresponding implementation in Java can be found in the Git repository mentioned at the beginning of Chapter 12. Hence, TeeTime has successfully been used to model a distributed P&F architecture.

Table A.1 on Page 217 shows the time line protocol of the TCP-based

execution which generates and processes 5,000 strings each with a length of one. It verifies that the workers and the master are created correctly. Moreover, the workers register themselves as expected such that the master assigns each worker one of the three parts of the pipeline via an individual deployment message (see, e.g., Line 6-10). Afterwards, Worker 1 tries to connect its `CPU Load Generator` stage with the `CPU Load Generator` stage of Worker 3 (see Line 11, 13, 14, and 16). After receiving its deployment message as well, Worker 3 accepts the request of Worker 1 (see Line 21). Hence, when comparing this execution with the modeled P&F architecture illustrated in Figure 12.12, Worker 1 represents `Node A` and Worker 3 represents `Node B`. If we continue our evaluation in this level of detail, we would finally verify that the execution proceeded without problems and terminated gracefully. For an in-depth discussion of the temporal sequence, we refer to the work of Echternkamp [Ech17].

Table A.2 on Page 221 shows another time line protocol of the execution which gracefully terminates due to an exception in a stage from Worker 1 (see Line 1). The worker notifies the master about the error and initiates its shutdown by triggering a termination signal. Thereupon, the master notifies the remaining workers to terminate as well. We can observe a similar behavior for the other two cases where one of the actors and one of the workers crashes. We again refer to the work of Echternkamp [Ech17] for a detailed discussion of the temporal sequence. In summary, TeeTime has successfully been used to deploy and to execute a distributed P&F architecture even in case of runtime errors.

Chapter 13

Performance Evaluation

In this section, we evaluate the performance of our TeeTime framework by using the corresponding Java implementation. In Section 13.1, we evaluate the overhead which results from building and executing P&F architectures with our framework. In Section 13.2, we compare the performance of both schedulers described in Section 9.1. Finally, in Section 13.3, we evaluate the feasibility and the performance of our parallelization approach described in Section 9.2.

13.1 Framework Overhead Evaluation

For a framework, high performance means low runtime overhead imposed on software applications that are built with the framework [WH12]. For this purpose, we measure and compare the overhead of the three most performance-critical framework parts: the unsynchronized pipe, the synchronized pipe, and the composite stage. To increase the external validity, we perform this overhead evaluation on three different multi-core processor architectures.

13.1.1 Overhead of the Unsynchronized Pipe

In this section, we investigate how much runtime overhead the unsynchronized pipe introduces. Hence, with this evaluation, we provide answers to our research question Q4 (*How to Efficiently Parallelize Arbitrary P&F Architectures?*) mentioned in Section 7.2. The results will be in the range of the associated metric M4.3 (*Execution time and speedup*).

Figure 13.1. Pipeline configuration used as micro-benchmark for the overhead evaluation of TeeTime's unsynchronized pipe.

Methodology

We compare the execution time of two different implementations of the pipeline illustrated by Figure 13.1. The first implementation does not use any pipes for inter-stage communication. Instead, each stage knows its successor stage and directly communicates with it. The second implementation uses TeeTime and thus employs pipes between each two stages. Since the first implementation avoids the usage of pipes, we expect that it performs slightly faster than the second implementation. By computing the difference of both execution times, we obtain the runtime overhead of the unsynchronized pipes.

In both implementations, the pipeline consists of a producer, n instances of the same stage $s(i), i \in \{1, .., n\}$, and a sink. The producer outputs integer values which pass through the n stage instances and finally arrive at the sink. Each stage $s(i)$ performs a synthetic operation with a configurable workload factor of w to simulate an arbitrary realistic operation. For this purpose, we use the micro-benchmark framework JMH [Orab]. It provides a scalable workload generator that burns CPU cycles according to the given workload value w. We execute the pipeline with different sizes $n \in \{1, 2, 4, 8, 16, 32\}$ and with different workloads $w \in \{50, 100, 150, 200\}$. Since the workload value is an abstract value without any unit, we however measure first the execution time of a stage for each w. In this way, we can better grasp what the workload values mean in realistic settings. Subsequently, we let each pipeline configuration from above run by a single thread on each of the three different multi-core processor architectures shown in Table B.1. To amortize variations in the measurements, we execute each configuration multiple times in the following manner: we set the number of Java virtual machine (JVM) runs to 3, the number of warmup iterations to 20, and the

Table 13.1. The actual execution times of JMH's workload values measured on Intel's Xeon.

JMH's workload value:	50	100	150	200
Actual execution time:	140 ns	280 ns	420 ns	560 ns

measurement iterations to 40.[1] We chose this specific configuration because it has yielded stable results on all of the three processor architectures.

Results and Discussion

Table 13.1 shows our workload measurements collected from Intel's Xeon. The execution time grows linearly according to the workload value as we have expected from a workload generator which is titled to be scalable. Note that even for the highest workload, we use in this evaluation, the corresponding execution time is very small compared to realistic computations (in the range of nanoseconds). We discuss this point later again in more detail.

Figure 13.2 shows the throughput of the pipeline implementations without pipes and with unsynchronized pipes in comparison measured on Intel's Xeon. Each diagram shows the workload (on the x-axis) depending on the throughput (on the y-axis) for a particular size of the pipeline. We refer to Figure B.1 (on Page 224) and Figure B.2 (on Page 225) for the results of AMD's Opteron and Oracle's UltraSparcT2+, respectively.

First, we observe that the pipeline implementation without pipes is faster than the one with unsynchronized pipes for each workload in each diagram (i.e., for each pipeline size). However, the difference in throughput decreases with an increasing stage workload. Consider, for example, the upper-left diagram which shows the results for a pipeline size of one. For a workload of 50, the difference in throughput is more than one element per μs. In contrast, the difference is less than 0.5 elements per μs for a workload of 100. For the highest workload (i.e., 200), the difference is about 0.1 element

[1] The JVM may use different just-in-time compile strategies from run to run. Furthermore, JVM classes are compiled not until they are accessed the first time. Finally, they are optimized for performance when they are executed frequently.

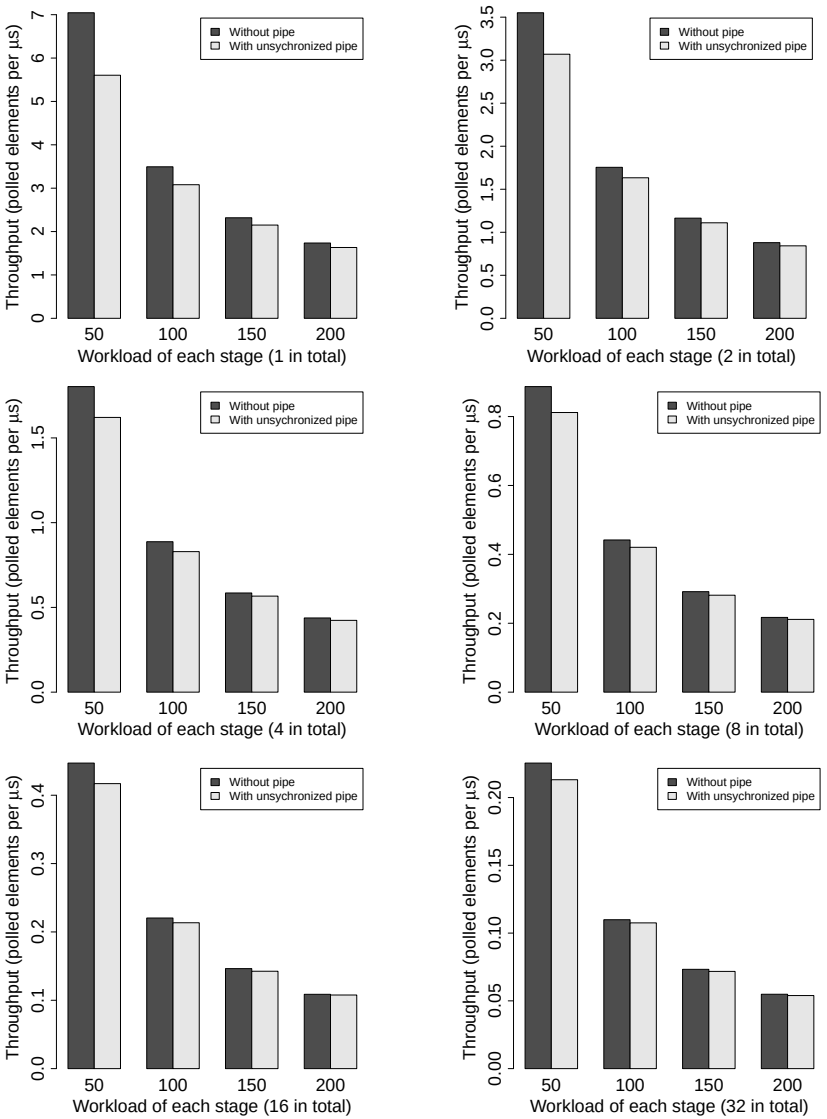


Figure 13.2. The pipeline implementations without pipes and with unsynchronized pipes on Intel's Xeon in comparison. Each diagram shows the results for a different pipeline size.

per μs. Now, remember that a workload of 200 corresponds to an execution time of only 560 ns on Intel's Xeon. If we instead assume a realistic stage workload of at least several hundreds of microseconds or even milliseconds, we can safely neglect the overhead of the unsynchronized pipe. In addition, the difference in throughput also decreases with an increasing size of the pipeline. Consider, for example, the lower-right diagram which shows the results for a pipeline size of 32. For a workload of 50, the difference in throughput is only 0.01 element per μs. Hence, it is less than two orders of magnitude compared to the pipeline with a size of one. Consequently, the actual overhead of the unsynchronized pipe contributes to the overall throughput even less when using a pipeline of a more realistic, larger size. In summary, we conclude that we can safely neglect the overhead of the unsynchronized pipe.

Threats to Validity

Internal validity: Although we vary the size of the pipeline and the workload per stage, we limited our overhead evaluation to a linear P&F architecture without any branch or any loop. This limitation allowed us to present the design and the results of the evaluation pipeline in a compact form. Moreover, we only used the Parallel Push/Pull scheduler introduced in Section 9.1.1. Other schedulers could perform differently. Furthermore, our evaluation bases on our own framework realization. Thus, different P&F architectures and different framework implementations could lead to different results.

External validity: We performed our experiment only on three different multi-core processors. Hence, the throughput results could differ on other multi-core processors. Furthermore, our results base on a particular JVM and OS version.

13.1.2 Overhead of the Synchronized Pipe

In this section, we investigate the runtime overhead of the synchronized pipe. Hence, with this evaluation, we provide answers to our research question Q4 (*How to Efficiently Parallelize Arbitrary P&F Architectures?*) mentioned in

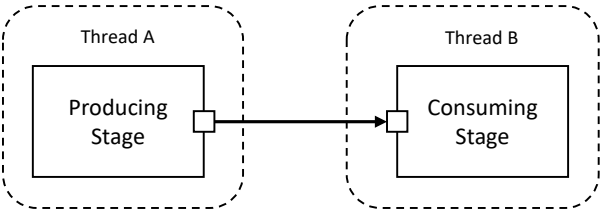


Figure 13.3. The experimental setup to measure the throughput of several different synchronized queue implementations.

Section 7.2. The results will be in the range of the associated metric M4.3 (*Execution time and speedup*).

Methodology

In contrast to the methodology of the unsynchronized pipe in Section 13.1.1, we do not compare a pipeline without pipes and a pipeline with synchronized pipes. We also do not compare the performance of the unsynchronized pipe and the synchronized pipe. Since, in any case, we expect to measure a significant overhead due to the additional synchronization effort. Instead, we compare TeeTime's synchronized pipe implementation with several different other synchronized pipe implementations. Each of the alternative implementations either bases on another lock-free or another lock-based queue. Hence, we determine the overhead of TeeTime's synchronized pipe relative to the other ones.

For this purpose, we implemented four additional lock-free versions and seven lock-based versions. They use either the JCTools library[2] or the Java API. For each version, we successively ran the following benchmark which is also illustrated by Figure 13.3. We built a P&F configuration comprising an active producing stage and an active consuming stage. The producing stage continuously adds a constant value to the pipe while the consuming stage simultaneously polls values from this pipe. We measured the throughput by counting the average number of elements polled from

[2] https://github.com/JCTools/JCTools

the pipe per microsecond. Since the lock-free versions are not resizable, we had to set a fixed capacity. We chose a relatively high capacity of 4096 such that the consuming stage could always poll a non-`null` element even in the presence of fluctuations caused by the OS scheduler. Like in Section 13.1.1, we built the benchmark with JMH and ran it on the three multi-core processors shown in Table B.1. To ensure a high reproducibility, we bound the benchmark to exactly two different cores on the same CPU socket.

Results and Discussion

Figure 13.4 shows the results of the lock-based pipes (on the left diagram) and the lock-free pipes (on the right diagram) on Intel's Xeon. For the results of the other processor architectures, we refer to Figure B.3 and Figure B.4.

Both diagrams show the name of the underlying queue (on the x-axis) depending on the measured average throughout (on the y-axis) together with its 99.9%-based confidence interval. The results show two major characteristics:

First, the lock-free single-producer/single-consumer pipe (SP/SC) used by TeeTime achieves the highest throughput on all three multi-core processors. In particular, it is 9.2x faster than the next best performing pipe MP/MC on Intel's Xeon.[3] In comparison with the worse-performing lock-free pipe which uses Java's `ConcurrentLinkedDeque`, TeeTime's SP/SC pipe outperforms it by a factor of 27.8.[4]

Second, blocking pipes often perform significantly slower than its lock-free versions. In particular, the blocking SP/SC pipe is slower than its lock-free version by a factor of 13 on Intel's Xeon.[5] This behavior can be observed again on all of the three multi-core architectures. The main reason is the additional synchronization effort necessary to synchronize multiple producers and/or multiple consumers.

Hence, these results show that we can heavily increase the throughput of P&F architectures on cache-coherent multi-core systems if we use a

[3] $139/15 \approx 9.2$
[4] $139/5 \approx 27.8$
[5] $139/10{,}7 \approx 13$

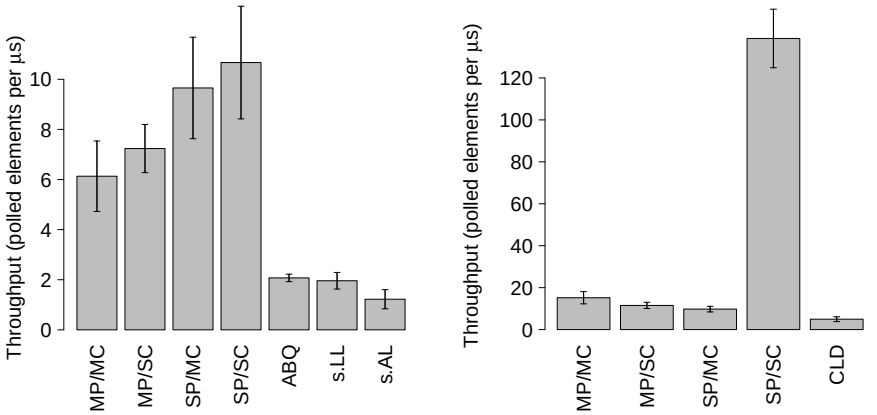


Figure 13.4. Throughput evaluation of various synchronized pipe implementations on Intel's Xeon. The left diagram shows the results for the blocking pipes, while the right diagram shows the results for the lock-free pipes. (ABQ=`ArrayBlockingQueue`, s.LL=`synchronized LinkedList`, s.AL=`synchronized ArrayList`, CLD=`ConcurrentLinkedDeque`)

synchronized pipe implementation which is lock-free and restricted to a single producer and a single consumer thread. In particular, the lock-free SP/SC pipe used by TeeTime is by far the fastest pipe and provides therefore the lowest runtime overhead of all.

Threats to Validity

Internal validity: The experimental results depend on our proposed framework design and the TeeTime-based pipe implementations. Different designs and implementations could lead to different results. Moreover, we conduct our experiment only with five lock-free pipes (including TeeTime's pipe) and seven blocking pipes. Alternative pipe implementations which were unknown to us during the experimentation phase could perform better than TeeTime's synchronized pipe implementation.

External validity: Like in Section 13.1.1, we performed our experiment only on three different multi-core processors. Hence, other multi-core proces-

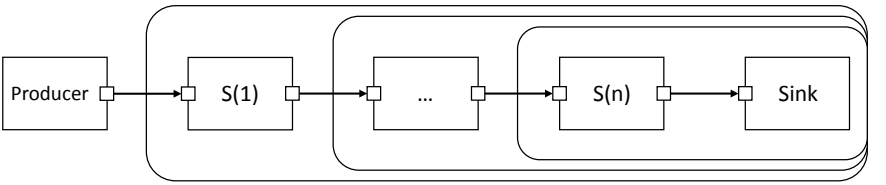


Figure 13.5. Pipeline configuration used as micro-benchmark for the overhead evaluation of TeeTime's stage composition. Each rounded rectangle represents a composite stage.

sors could perform differently. Furthermore, our results base on a particular JVM and OS.

13.1.3 Overhead of the Composite Stage

In this section, we investigate how much runtime overhead the composition of stages introduce. Hence, with this evaluation, we provide answers to our research question Q4 (*How to Efficiently Parallelize Arbitrary P&F Architectures?*) mentioned in Section 7.2. The results will be in the range of the associated metric M4.3 (*Execution time and speedup*).

Methodology

We compare the execution time of the pipelines illustrated in Figure 13.1 and Figure 13.5. Both implementations use TeeTime to its full extent, that is, in particular with unsynchronized pipes. However, the first implementation uses primitive stages only while the second one uses composite stages only. As explained in Section 8.2.1, we do not expect any additional runtime overhead due to the second implementation since composite stages are flattened to its underlying P&F architectures. The rest of the experimental setup equals the one of the unsynchronized pipe evaluation in Section 13.1.1.

Results and Discussion

Figure 13.6 shows the throughput of the pipeline implementations with primitive stages and with composite stages in comparison measured on Intel's Xeon. Each diagram shows the workload (on the x-axis) depending on the throughput (on the y-axis) for a particular size of the pipeline. We refer to Figure B.5 and B.6 for the results of AMD's Opteron and Oracle's UltraSparcT2+, respectively.

Let us first consider the upper-left diagram in isolation which shows the results for a pipeline size of one. Independent of a particular workload, the difference in throughput is minimal. If we take a closer look at the confidence intervals, we can see that these intervals overlap for each of the four workload values. Thus, for a pipeline size of one, we cannot verify a measurable difference in throughput. If we consider each of the other five diagrams in the same way, we are again not able to identify a significant increase or decrease in throughput. Hence, the performance of composite stages do not depend on the size of the pipeline. In summary, we conclude that composite stages do not introduce any runtime overhead.

Threats to Validity

Internal validity: Although we vary the size of the pipeline and the workload per stage, we limited our overhead evaluation to a linear P&F architecture without any branch or any loop. This limitation allowed us to present the design and the results of the evaluation pipeline in a compact form. Moreover, we only used the Parallel Push/Pull scheduler introduced in Section 9.1.1. Other schedulers could perform differently. Furthermore, our evaluation bases on our own framework realization. Thus, different P&F architectures and different framework implementations could lead to different results.

External validity: We performed our experiment only on three different multi-core processors. Hence, the throughput results could differ on other multi-core processors. Furthermore, our results base on a particular JVM and OS version.

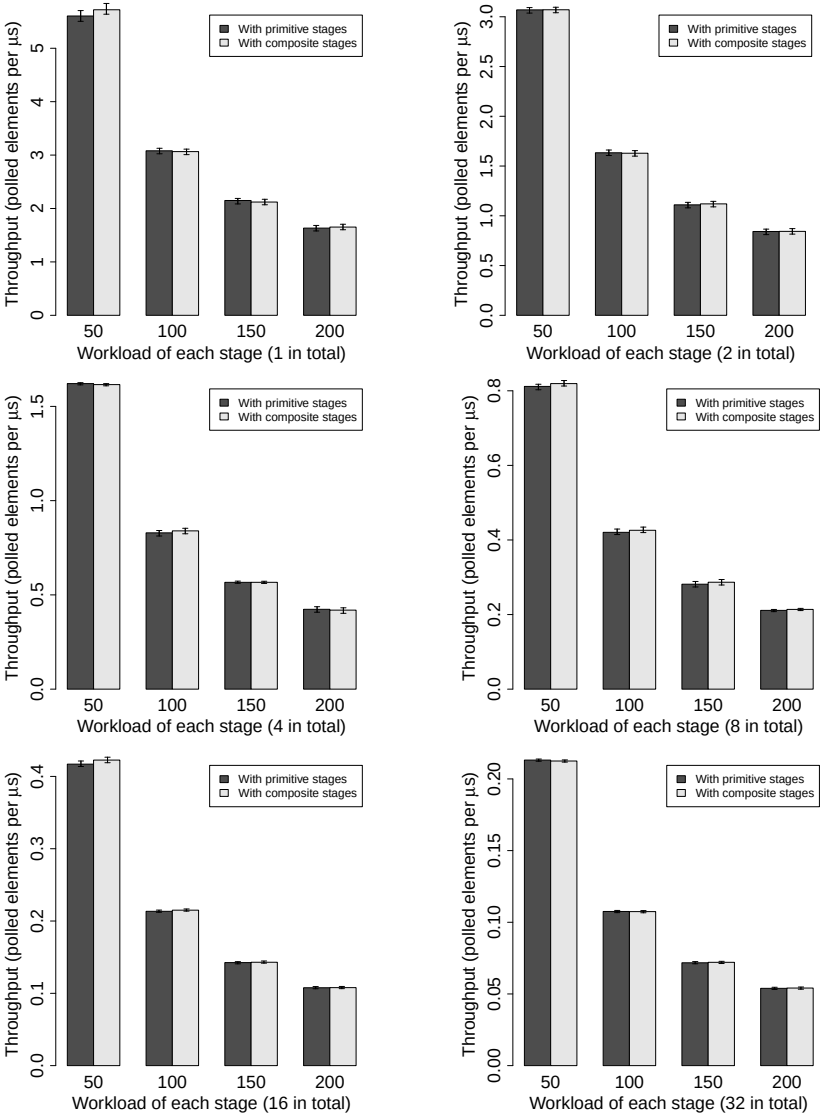


Figure 13.6. The pipeline implementations with primitive stages and with composite stages on Intel's Xeon in comparison. Each diagram shows the results for a different pipeline size.

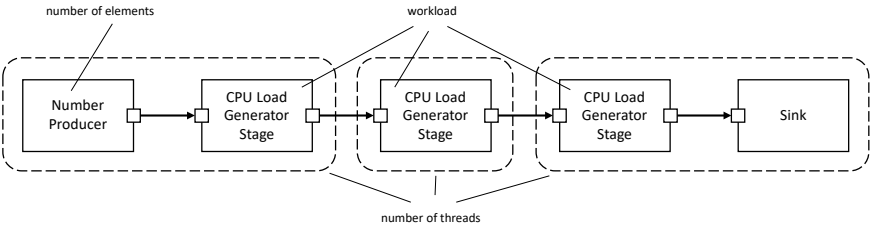


Figure 13.7. The P&F configuration used as benchmark for the scheduling approach evaluation. It is configurable by the three parameters: the number of elements, the workload per stage, and the number of threads.

13.2 Scheduling Approach Evaluation

In this section, we investigate our Parallel Push/Pull scheduler introduced in Section 9.1.1. Hence, with this evaluation, we provide answers to our research questions Q3 (*How to Execute Arbitrary P&F Architectures?*) and Q4 (*How to Efficiently Parallelize Arbitrary P&F Architectures?*) mentioned in Section 7.2. The results will be in the range of the associated metrics M4.2 (*Number and kind of different parallel scheduling algorithms*) and M4.3 (*Execution time and speedup*).

13.2.1 Methodology

We compare our new parallel push/pull scheduler (PPP) from Section 9.1.1 with the commonly used global task pool (GTP) scheduler from Section 9.1.2. For this purpose, we implemented both schedulers with TeeTime's Java implementation. Furthermore, we implemented a P&F configuration which serves as a benchmark. Figure 13.7 shows the corresponding P&F setup. According to the given number of elements, the `Number Producer` produces one number after the other. Each number then passes through the three `CPU Load Generator Stage` such that it finally arrives at the `Sink`. Thereby, the `CPU Load Generator Stages` delay the forwarding of the numbers by taking up as much CPU time as the given workload value indicates. In this way, we simulate the duration of real computations. We execute this benchmark

Table 13.2. The benchmark configurations used for the scheduler comparison.

	#elements	workload	#threads
SH_1	1,000	1,000,000	1
SL_1	1,000,000	1,000	1
SH_3	1,000	1,000,000	3
SL_3	1,000,000	1,000	3

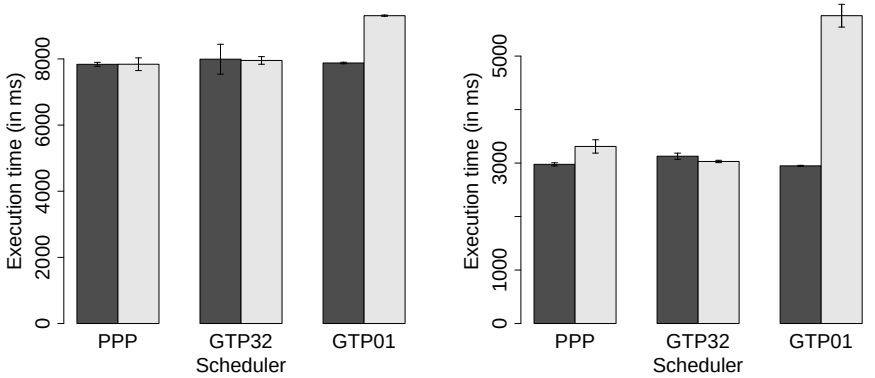
multiple times to evaluate the impact of different numbers of elements, different workloads per stage, and different numbers of threads. For each such different benchmark configuration, we first perform three warmup iterations to prepare the Just-in-Time compiler for the actual measurements. Subsequently, we measure the average execution time of ten measurement iterations to amortize variations in the measurements. Our scenarios, that is, our benchmark configurations are as follows.

1. Few elements with a high workload per stage for one thread and three threads.

2. Many elements with a low workload per stage for one thread and three threads.

Table 13.2 shows our benchmark configurations with their actual parameter values. Since the performance of the GTP scheduler depends on the number of executions per task, our comparison considers one GTP version with one execution and another GTP version with 32 executions.[6] For an increased external validity, we execute these scenarios on two different Intel systems. Details about the hardware and software environment are shown in Table B.2. Our expectation is that the new PPP scheduler performs at least equally well as the GTP scheduler does. For a low number of executions per task, we expect that the GTP scheduler takes more time than the PPP scheduler due to the additional synchronization in the MP/MC pipes and in the global task pool.

[6]More than 32 executions did not improve the performance on both systems any further.



(a) Results of the scenarios SH_1 (dark gray) and SL_1 (light gray). Both scenarios used one thread.

(b) Results of the scenarios SH_3 (dark gray) and SL_3 (light gray). Both scenarios used three threads.

Figure 13.8. Intel i5 results of the scenarios shown in Table 13.2 (PPP=Parallel Push/Pull scheduler, GTPxx=Global Task Pool scheduler with xx executions per task).

13.2.2 Results and Discussion

Figure 13.8 shows the results of the four scenarios performed on the Intel i5 system. Figure 13.8a shows the results of the scenarios SH_1 and SL_1, that is, the ones which use one thread. Both schedulers (independent of the executions per task) require about 7,900 ms for SH_1 (dark gray bars). The confidence intervals overlap. However, for SL_1 (light gray bars), the GTP version with one execution per task takes about 1,300 ms longer than PPP and GTP32. The exact measurement values are given by Table B.3.

Figure 13.8b shows the results of the scenarios SH_3 and SL_3, that is, the ones which use three threads. The schedulers PPP and GTP01 take about 2,950 ms for SH_3 (dark gray bars). Their confidence intervals overlap. GTP32, however, takes about 100 ms longer. For SL_3 (light gray bars), PPP and GTP01 take about 150 ms and, respectively, 2,600 ms longer than GTP32.

Both diagrams show that, in our high workload scenarios SH_1 and SH_3

(dark gray bars), both schedulers perform equally well. In our low workload scenarios SL_1 and SL_3 (light gray bars), the global task pool scheduler with only one execution per task performs worst. GTP01 creates much more tasks than GTP32 and thus it has a much higher access rate on the task pool. Since processing of tasks take less time in these scenarios, the management effort including the synchronization carries much more weight. For this reason, the execution times of GTP01 are larger than the ones of GTP32 in low workload scenarios. PPP, however, performs again similarly well compared to GTP32.

The Intel i7 results of the scenarios show a similar picture (see Table B.4). In most cases, the schedulers perform equally well. Again, only GTP01 takes more time in the low workload scenarios.

Hence, in summary, our new scheduler PPP can greatly keep up with the commonly used global task pool scheduler. If GTP is not properly configured with regard to its executions per task, PPP can outperform GTP in a significant way. Furthermore, the implementation of PPP requires considerably less effort. It does not use a task pool and its threads do not execute other stages than the ones initially assigned. For this reason, the Java files implementing PPP have a size of only 36,15 KB in total. In comparison, the Java files of GTP have a size of 46,4 KB in total, that is, 28% more than PPP. However, in return, PPP requires to declare stages active by the application developer in order to utilize multi-core systems. Thus, for our scenarios, we recommend to use PPP or GTP32. Both schedulers show the same efficiency. This result is noteworthy on its own because both schedulers are based on entirely different concepts.

13.2.3 Threats to Validity

Internal validity: Although we built benchmarks that represent important scenarios, we see potential for improvements in the design of the benchmarks. For example, the benchmarks could be changed to implement more realistic use cases. Furthermore, the experimental results depend on our proposed design and the TeeTime-based implementation of both schedulers. Different designs and implementations could lead to different results.

External validity: We evaluated our scheduler only on two multi-core

systems on particular JVM and OS versions. Other systems and system configurations could perform differently and might require an adjustment of GTP's optimal number of executions per tasks. Moreover, they could require to change the VM/warmup/real iterations to get stable results.

13.3 Parallelization Approach Evaluation

In this section, we investigate our P&F parallelization approach based on the task farm stage (TFS) and the associated self-adaptive manager (SAM) introduced in Section 9.2. Note that this evaluation is mainly taken from our previous work [WWH16]. It provides answers to our research questions Q3 (*How to Execute Arbitrary P&F Architectures?*) and Q4 (*How to Efficiently Parallelize Arbitrary P&F Architectures?*) mentioned in Section 7.2. The results will be in the range of the associated metrics M3.2 (*Number of threads and nodes*), M4.1 (*Number and kind of different parallelization approaches*), and M4.3 (*Execution time and speedup*).

We evaluate our approach with respect to its feasibility and, especially, to its performance. On the one hand, we show that it increases the overall throughput. On the other hand, we show how we can maximize this increase. For this purpose, we raise the following, more specific research questions:

Feasibility

1a) Does our TFS increase the overall throughput?

1b) Does our SAM automatically adapt the number of stages according to the current runtime workload?

Performance

2a) To what extent does the throughput prediction algorithm influence the overall throughput?

2b) To what extent does the throughput boundary influence the overall throughput?

13.3.1 Scenarios

We consider four different scenarios. The first scenario represents a CPU-intensive computation with a balanced workload. Similarly, the second scenario represents a CPU-intensive computation, though with an unbalanced workload. The third scenario represents an I/O-intensive computation where, e.g., the file system is accessed most of the time. The fourth scenario represents a more common and more realistic kind of computation. It covers both a CPU-intensive part and an I/O-intensive part.

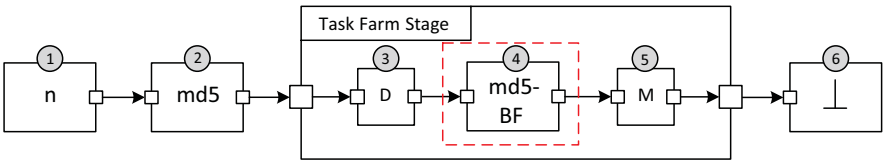
We implemented each scenario as a benchmark. Figure 13.9 shows their corresponding P&F architectures. Benchmark 1 (B1) uses the task farm to compute the original number for a given hash value and a fixed hash function by applying bruteforcing (see Stage 4 in Figure 13.9a). It generates the hash value for every number (from 0 to $max(integer)$) until it matches the input hash value. Then, it outputs the corresponding number. Stage 1 serves as a workload generator for a sequence of either a fixed number (balanced workload) or a linearly increasing number (unbalanced workload). Stage 2 computes the hash value of each incoming number and serves as input for the task farm. Stage 6 represents a sink which discards the incoming original numbers.

Benchmark 2 (B2) uses the task farm to write n characters to a temporary text file for a given number n (see Stage 3 in Figure 13.9b). Similar to B1, Stage 1 and 6 serve as a balanced workload generator and as a sink, respectively.

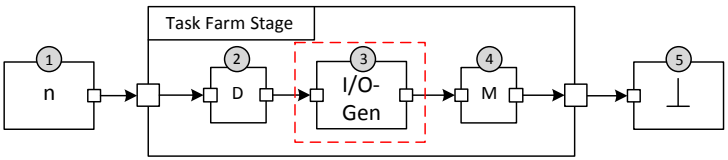
Benchmark 3 (B3) uses the task farm to transform XML files by applying a fixed XSLT transformation (see Figure 13.9c). Stage 3 loads the incoming XML file, Stage 4 transforms it in memory, and Stage 5 writes it back to the file system. Again, Stage 1 and 7 serve as a balanced workload generator and as a sink, respectively. For benchmarks details, we refer to our replication package [CH16].

13.3.2 Experimental Setup

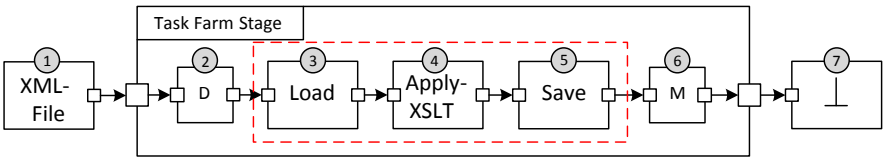
We execute each benchmark on four different multi-core systems. Table B.5 shows their hardware and software details. To answer the questions 2a and



(a) Benchmark 1 (B1) represents a CPU-intensive computation with either a balanced workload (using a constant workload generator) or an unbalanced workload (using a linearly increasing workload generator).



(b) Benchmark 2 (B2) represents an I/O-intensive computation with a balanced workload.



(c) Benchmark 3 (B3) represents a combined CPU-I/O-intensive computation with a balanced workload. Unlike B1 and B2, it uses a composite duplicable stage.

Figure 13.9. Benchmarks for the performance evaluation. Each red rectangle represents the duplicable stage of the task farm stage.

2b concerning the performance, we additionally execute each benchmark with three different throughput prediction algorithms and various different throughput boundary values. In this way, we get to know how these aspects influence the overall throughput. For comparison, we also execute a benchmark implementation which does not use the TFS at all. Moreover, we apply the following configurations. We set the iteration interval of our SAM to 50 ms in order to measure a throughput greater than one for all scenarios on all of the four systems. We configured the workload generator stages to produce 1,000 to 10,000 elements depending on the benchmark and the system. Since we do not intend to compare benchmark configurations or systems with each other, we chose different values for each pair. In this way, we effectively limited the execution times of the benchmarks.

We use the Oracle Java Runtime Environment (JRE)[7] in the version *1.8.0_60-b27*. We set the number of Java Virtual Machine (JVM) runs to 3, the number of warmup iterations to 3, and the measurement iterations to 5 to amortize variations in the measurements.[8] We chose this specific configuration because it has yielded stable results on all of the four systems. The final execution time of a benchmark configuration for a given system is then defined by the median of all 15 measurements.

13.3.3 Results and Discussion

Since we evaluated our TFS for four scenarios on four multi-core systems with three different throughput prediction algorithms and various different throughput boundary values, we are not able to present all of the results in detail. Thus, we first discuss an aggregated view on the results for each benchmark configuration on each multi-core system. Afterwards, we exemplarily discuss the results for the *Intel* system concerning feasibility and performance. We chose the *Intel* system since it is the most recent system of all used systems. The benchmarks on the other systems yield similar results. For detailed results, we refer to our replication package [CH16].

[7] `http://www.oracle.com/technetwork/java/javase/overview/index.html`

[8] The JVM may use different just-in-time compile strategies from run to run. Furthermore, JVM classes are compiled not until they are accessed the first time. Finally, they are optimized for performance on frequent access.

Table 13.3. Lowest mean execution times of the benchmark configurations achieved without and, respectively, with our TFS on the four multi-core systems. For each benchmark configuration, the regression prediction algorithm was used.

Benchmark config.	Duration on *SUN* (w/o vs. w/ TFS)	Duration on *AMD-I* (w/o vs. w/ TFS)	Duration on *Intel* (w/o vs. w/ TFS)	Duration on *AMD-II* (w/o vs. w/ TFS)
B1 (bal. workload)	21 sec./5 sec. = 4.2 bound. val. = 0.025	10 sec./3 sec. = 3.3 bound. val. = 0.025	17 sec./3 sec. = 5.7 bound. val. = 0.025	25 sec./12 sec. = 2.1 bound. val. = 0.2
B1 (unbal. workload)	20 sec./5 sec. = 4.0 bound. val. = 0.0	35 sec./7 sec. = 5.0 bound. val. = 0.025	29 sec./4 sec. = 7.3 bound. val. = 0.0	20 sec./10 sec. = 2.0 bound. val. = 0.2
B2 (bal. workload)	13 sec./4 sec. = 3.3 bound. val. = 0.025	49 sec./14 sec. = 3.5 bound. val. = 0.225	15 sec./4 sec. = 3.8 bound. val. = 0.025	26 sec./17 sec. = 1.5 bound. val. = 0.2
B3 (bal. workload)	34 sec./7 sec. = 4.9 bound. val. = 0.2	13 sec./4 sec. = 3.3 bound. val. = 0.025	13 sec./2 sec. = 6.5 bound. val. = 0.025	9 sec./5 sec. = 1.8 bound. val. = 0.2

Overview

Table 13.3 shows the lowest mean execution times of each benchmark configuration for each system which we have measured with and, respectively, without our TFS. The results consistently indicate a speedup for all benchmark configurations on all systems. We measured a speedup ranging from 1.5 (B2 on *AMD-II*) to 7.3 (unbalanced B1 on *Intel*). Hence, our TFS successfully increases the overall throughput. Moreover, the results show that the lowest execution time depends on the used throughput boundary. Different benchmark configurations and different systems require different boundary values. Finally, the lowest execution time also depends on the used prediction algorithm as discussed later on Page 175. However, the regression algorithm yields the best results for all benchmark configurations on all systems.

Feasibility

Figure 13.10 illustrates the total throughput at any point in time while running B1 (balanced workload) on the *Intel* system with various throughput boundaries.[9] If we consider the time axis, we see that the fastest run of the benchmark takes about six seconds for a throughput boundary below 0.025. All runs with a throughput boundary greater than 0.025 take more

[9] We increased the execution time from 3 sec. to 6 sec. for clarity.

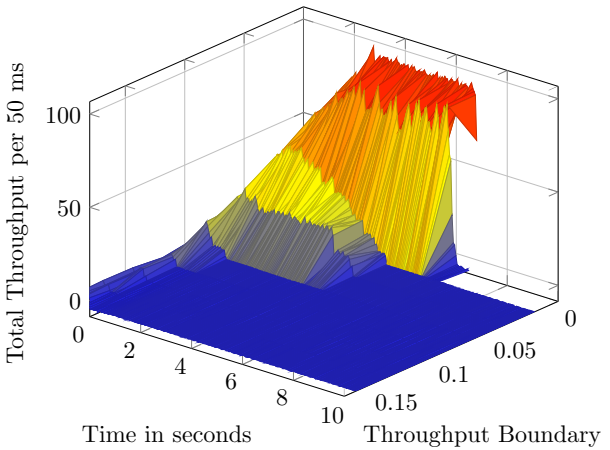


Figure 13.10. Total throughput at any point in time while running B1 (balanced workload) on the *Intel* system with various throughput boundaries.

than ten seconds.[10] The maximum peak of the throughput is reached at the third second and is kept until the end of the run. Compared to an execution with a higher throughput boundary, e.g., 0.15, the throughput is 30 times higher. These measurements perfectly correlate to the number of stages at the corresponding points in time (see Figure 13.11). The higher the total throughput, the higher is the number of stages within our TFS.

Hence, our TFS increases the overall throughput (Question 1a). Furthermore, our SAM automatically adapts the number of stages according to the current runtime workload (Question 1b).

Performance

Figure 13.12a to Figure 13.12d illustrate the execution times of all four benchmarks depending on the used prediction algorithm and the used throughput boundary. In addition to that, each figure shows the execution times of the benchmark implementation which does not use the TFS (see the

[10] We omitted boundaries greater than 0.15 for clarity.

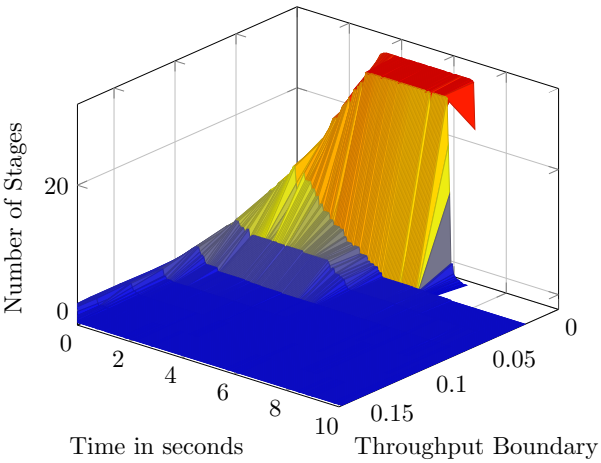
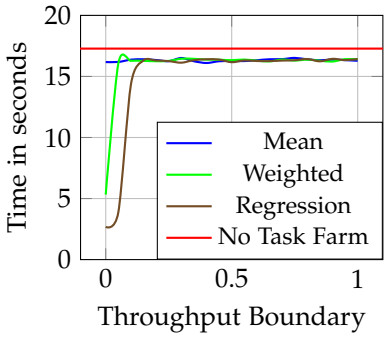


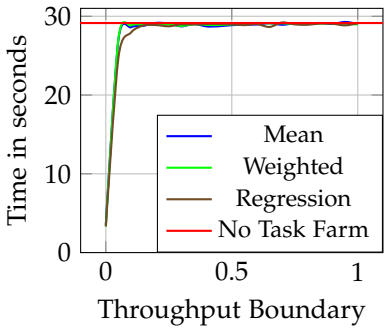
Figure 13.11. Total number of stages at any point in time while running B1 (balanced workload) on the *Intel* system with various throughput boundaries.

red lines). For this reason, this non-TFS implementation always takes the same amount of time independent of the throughput boundary. In contrast, the execution times of the TFS-based implementations vary. However, they are always either equal to or lower than the ones of the non-TFS implementation. This is because the TFS introduces at least two new threads: one for the initial worker stage and one for the merger. Therefore, these stages can work in parallel and cause a speedup all the time. Nevertheless, this speedup is often not very high. It heavily depends on the throughput boundary and on the used prediction algorithm. In short, the regression algorithm outperforms its alternatives in all four scenarios provided we choose a proper throughput boundary. Consistent with the results shown in Table 13.3, Figure 13.12a to Figure 13.12d show that the lowest execution time on the *Intel* system is reached by a throughput boundary below 0.025.

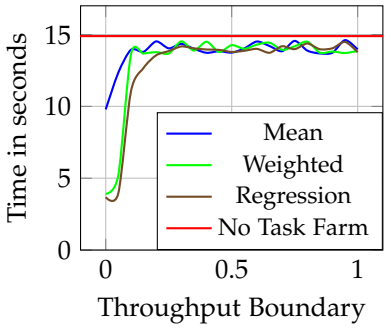
Hence, the right choice of the prediction algorithm is crucial for a high overall throughput (Question 2a). In our benchmarks, the regression algorithm consistently performs best. Moreover, the throughput boundary is also important for a high overall throughput (Question 2b). In our benchmarks,



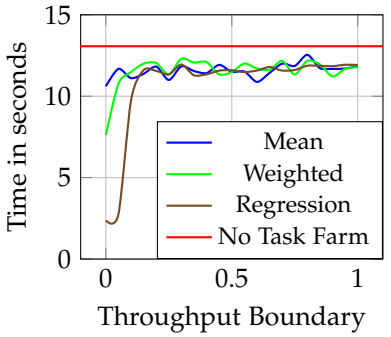
(a) Execution times of B1 (balanced workload) for the *Intel* system.



(b) Execution times of B1 (unbalanced workload) for the *Intel* system.



(c) Execution times of B2 (balanced workload) for the *Intel* system.



(d) Execution times of B3 (balanced workload) for the *Intel* system.

Figure 13.12. Execution times of the benchmarks on the *Intel* system either with the mean algorithm (blue), with the weighted algorithm (green), with the regression algorithm (brown), or without our task farm stage (red).

a throughput boundary between 0 and 0.225 performs best.

13.3.4 Threats to Validity

Internal validity: Although we built benchmarks that represent the four scenarios, we see potential for improvements in the design of the benchmarks. For example, the benchmarks could be changed to implement more realistic use cases. Furthermore, the experimental results depend on our proposed design and the TeeTime-based implementation of both the task farm stage and the self-adaptation manager. Different designs and implementations could lead to different results.

External validity: We evaluated our TFS for four coarse-grained scenarios only. Further scenarios would increase the external validity. Moreover, we evaluated our TFS only on four multi-core systems with different hard drives. Other systems could perform differently and might require an adjustment of the VM/warmup/real iterations to get stable results. Furthermore, our results base on a particular JVM and OS version.

Chapter 14

Reusability and Extensibility Evaluation

In this chapter, we evaluate the reusability and the extensibility of our P&F framework TeeTime. For this purpose, we consider the P&F architectures built and used in Chapter 12 and 13. They show that some stages are often reused even in different areas of application. Moreover, they show that project-external developers could make extensions to TeeTime with reasonable effort.

14.1 Methodology

The main goal of this evaluation is to show that TeeTime can serve as a base for further research and for industrial use. We will show that TeeTime fulfills this goal by verifying that TeeTime's architecture and TeeTime's stages provide a high reusability and a high extensibility. First, we collect and consider the number of reused and extended TeeTime entities and stages from the P&F architectures described in Chapter 12 and 13. Second, we discuss and justify which TeeTime entities and stages belong to the public API and which are hidden from the application developer. Finally, we briefly describe how students extend TeeTime by means of two examples. Hence, with this evaluation, we provide an answer to our research question Q2 (*How to Provide Reusability and Extensibility by the Framework?*) mentioned in Section 7.2. In particular, we identify fixed, extensible, and exchangeable framework entities covered by the metric M2.1. Moreover, we identify public and private entities of the framework covered by metric M2.2.

Table 14.1. Entities and stages which were reused in the evaluations from Chapter 12 and 13.

Entity/Stage	Trace Rec.	Quick-Sort	User Beh.	Live Vis.	Distr. Exec.	PPP Sched.	GTP Sched.	TFS
Distributor	✓		✓	✓				✓
InitialElementProducer	✓	✓						
CollectorSink		✓						✓
InstanceOfFilter	✓		✓					
Directory2FilesStage	✓		✓					
Traverser						✓	✓	
SynchedPipe						✓	✓	✓

14.2 Reusable and Extensible Entities

Table 14.1 shows the TeeTime entities and the TeeTime stages which were reused in the evaluations from Chapter 12 and 13. For example, the `Distributor` stage is used in four of our evaluations for different P&F architectures. In each case, it distributes incoming elements to one or all of its output ports without knowing the type of the elements. Similarly, the Parallel Push/Pull scheduler shares some entities with the Global Task Pool scheduler. Both of them use (1) the traverser to perform internal analyses and (2) the synchronized pipe to connect stages which are executed by different threads. Future P&F architectures and scheduler implementations can reuse these entities and stages in a similar way. For this purpose, our two TeeTime implementations already provide many reusable stages as indicated by the number of stage declarations shown in Table 11.1 on Page 131. Hence, TeeTime enables application developers to reuse its framework entities as well as its stages.

Table 14.2 shows the TeeTime entities and the TeeTime stages which were extended in the evaluations from Chapter 12 and 13. For example, the `Configuration` entity is extended in all of our evaluations to define custom P&F architectures. Similarly, the `AbstractStage` is extended for each custom stage in the corresponding evaluations. Furthermore, both of our schedulers extend the `TeeTimeScheduler` interface to specify new scheduling algorithms.

Table 14.2. Entities and stages which were extended in the evaluations from Chapter 12 and 13.

Entity/Stage	Trace Rec.	Quick-Sort	User Beh.	Live Vis.	Distr. Exec.	PPP Sched.	GTP Sched.	TFS
Configuration	✓	✓	✓	✓	✓	✓	✓	✓
AbstractStage	✓	✓	✓	✓	✓			✓
CompositeStage	✓	✓	✓	✓				✓
TeeTimeScheduler						✓	✓	
AbstractPipe						✓	✓	

In addition, each of the schedulers define custom pipes (e.g., `SpScPipe` and `MpScPipe`) which inherit from the `AbstractPipe` entity. Future P&F architectures, stages, and schedulers can extend these entities and stages in a similar way and thus exchange available implementations. For this purpose, both of our TeeTime implementations offer all of the framework entities illustrated by Figure 8.1 on Page 64 as publicly accessible and extensible program elements. Hence, TeeTime enables application developers to extend its framework entities as well as its stages.

14.3 Public and Private Entities

We developed TeeTime's framework architecture as well as its two reference implementations carefully in terms of API design in general and access control[1] in particular. Therefore, most internal framework entities and methods are hidden from the application developer by using visibility modifiers such as `protected` and `private`. In this way, we additionally support and improve TeeTime's reusability and extensibility.

For example, we declared the method `execute()` from `AbstractStage` as `protected` (see Figure 8.1 on Page 64) such that the application developer can implement the method for a new stage. However, she cannot invoke the method outside the stage by herself. Due to the visibility modifier, only the

[1] `https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html`

```
1 protected <T> InputPort<T> createInputPort(final Class<T> type,
      final String name) {
2   InputPort<T> inputPort = new InputPort<T>(type, this, name);
3   this.inputPorts.add(inputPort);
4   this.numOpenedInputPorts++;
5   return inputPort;
6 }
```

Listing 14.1. The Java implementation of the method `createInputPort`.

framework itself may call `execute()` on stages. Hence, by using `protected` instead of `public`, we reduce the risk of misusing TeeTime's stages.

Another example refers to the port creation. We declared the constructors of the entities `InputPort` and `OutputPort` as `package-private` such that they cannot be instantiated by the developer. Instead, the developer creates ports via the two methods `createInputPort(..)` and `createOutputPort(..)`. Listing 14.1 shows the corresponding Java implementation of the method `createInputPort` from the `AbstractStage`. Since the entities `InputPort` and `AbstractStage` share the same package, the method `createInputPort` from `AbstractStage` can invoke the constructor of `InputPort` (Line 2). Simultaneously, it also adds the new `InputPort` instance to an internal list of input ports (Line 3). Furthermore, it increments a counter which represents the current number of open input ports (Line 4). If this counter is zero, the stage terminates and passes the termination signal to its successor stages.

Hence, by leaving the instantiation of ports to factory methods, we enable the framework to autonomously manage the port instantiation and the port handling as well as the stage termination logic. In this way, the developer is forced to create ports in only one single way and thus does not run into danger of declaring stages in a wrong way. For the same reason, we hide the creation of intermediate pipes when connecting ports in the configuration. Similarly, the logics for passing signals and for handling exceptions is encapsulated by the framework such that the developer cannot accidentally manipulate it. In sum, by specifying a well-defined API and by hiding framework internal procedures from the developer, we provide sophisticated support for reusability and extensibility in TeeTime.

14.4 Case Studies

In the following, we present two small case studies which show how project-external developers add the Global Task Pool Scheduler from Section 9.1.2 and the Task Farm Stage from Section 9.2 to TeeTime's Java implementation. They provide insight into how TeeTime's reusability and extensibility can support the development process of P&F architectures.

14.4.1 Adding the Global Task Pool Scheduler

We let a bachelor student implement the Global Task Pool Scheduler in the context of a seminar paper. For this purpose, we only gave him the description of the approach, access to the source code, and the documentation of TeeTime's Java implementation. Then, he proceeded as follows. After having understood the approach and read the documentation, he created a class which represents the new scheduling approach. Subsequently, he let the class implement the `TeeTimeScheduler` interface by filling the bodies of the required methods with the corresponding logic. For example, he implemented the initialization logic which includes creating the list of stage pools and setting the level index of each stage. For this purpose, he reused the `Traverser` such that the scheduler is able to visit all of the stages of a configuration. Furthermore, he reused the traverser visitor from the Parallel Push/Pull Scheduler which replaces intermediate pipes by specific pipe implementations. Since the Global Task Pool Scheduler exclusively uses MP/SC pipes, he wrote a corresponding implementation and configured the visitor accordingly. Finally, after finishing the implementation of the new scheduler, he wrote associated unit tests to verify that the scheduler works as expected. This case study shows that the project-external developer only required to extend and to reuse available entities to add a new scheduler to TeeTime. In particular, he did not need to change any existing entity. Hence, we are confident that adding further schedulers should not cause any unexpectedly high implementation effort. The reusability and the extensibility of TeeTime seem to support the application developer in a sufficient manner.

14.4.2 Adding the Task Farm Stage

We let a master student implement the Task Farm Stage (TFS) in the context of a master's thesis. For this purpose, we only gave him the description of the approach, access to the source code, and the documentation of TeeTime's Java implementation. Then, he proceeded as follows. After having understood the approach and read the documentation, he created a class which represents the new task farm stage. In order to make the class a composite stage, he let it inherit from TeeTime's `CompositeStage` class. Subsequently, he implemented the stage accordingly. For this purpose, he reused the `Distributor` and the `Merger` which are already included in the TeeTime distribution. Moreover, he added the `ITaskFarmDuplicable` interface as specified by Figure 9.7. Afterwards, he let all of the existing TeeTime stages, which have exactly one input port and one output port, implement this interface. Then, he implemented the self-adaptation manager (SAM) in such a way that it can be easily plugged in or out according to the developer's need. Finally, after finishing the implementation of the TFS and its associated SAM, he wrote associated unit tests to verify that each in isolation and both together work as expected. This case study shows that the project-external developer almost exclusively required to extend and to reuse available entities to add a new complex stage to TeeTime. The intrusive adaptations to existing TeeTime stages were only necessary to provide a generic TFS that is independent of any specific stage. According to the student, this task was easy and did not require much additional implementation effort. Hence, we are confident that adding further stages should not cause any unexpectedly high implementation effort. The reusability and the extensibility of TeeTime seem to support the application developer in a sufficient manner.

Chapter 15

Related Work

In this chapter, we discuss related work concerning TeeTime which goes beyond those already described in Chapter 2. We start in Section 15.1 and Section 15.2 by describing related parallel and, respectively, distributed P&F frameworks. Afterwards, in Section 15.3, we present related parallel P&F execution strategies. Finally, we briefly discuss two related architectural styles in Section 15.4.

15.1 Related Parallel P&F Frameworks

A generic P&F framework is Apache Commons Pipeline.[1] This framework can take advantage of additional threads, but assumes that the stages are implemented in a thread-safe manner. Thus, taking advantage of additional computing power requires a deeper knowledge of concurrency issues from the application developer. Furthermore, the project has no released version and is not developed any further since 2009.

Ptolemy II [BL10] is a Java framework that supports experimenting with the actor-oriented design. Although it can be used to assemble networks in a P&F oriented style, it requires additional knowledge to configure and execute a pipeline configuration that goes beyond the P&F pattern. Furthermore, its use of a scheduler and coarse-grained synchronization mechanisms results in an additional run-time overhead.

The monitoring and dynamic analysis framework Kieker [HWH12] provides an internal P&F framework to create arbitrary P&F-based analyses. However, it lacks support for composing stages, static type checking of

[1] http://commons.apache.org/sandbox/commons-pipeline

ports, and distributed P&F architectures. Moreover, its scheduler as well as unnecessarily coarse-granular synchronization points effectively sequentialize the execution. Thus, we have replaced Kieker's internal P&F analysis framework by TeeTime as of version 1.14.

XML Calabash[2] is an implementation of the XML pipeline language XProc.[3] This language can be used to describe pipelines consisting of atomic or compounded operations on XML documents. Therefore, similar to ExplorViz, it does not support arbitrary P&F architectures.

Pipes[4] is a Java-based framework to define and execute linear sequences of computations. So-called pipes represent the atomic computing steps and form, once connected, a processing graph. Hence, in this framework, the term *pipe* corresponds to the term *stage* in the P&F style. As the pipes implement interfaces with generics, they are type-safe. Furthermore, they can be combined into a composed pipe. However, the Pipes framework does not support a parallel or a distributed execution.

Java 8 introduced a new streams API[5] that allows to successively apply multiple functions on a stream of elements. Besides the lack of reading from and writing to more than one stream at once, its support for executing stages in parallel is limited to particular use cases. Furthermore, the API does not allow to add custom operations on streams.

Microsoft's Task Parallel Library (TPL) includes a dataflow library[6] which enables the parallelization of CPU-intensive and I/O-intensive .NET applications. It provides an actor-oriented programming model which focuses on stateless computations via message passing. Although it is highly customizable, it lacks support for multiple input and output ports as well as a distributed execution.

Black et al. [BHW+02] propose an abstraction for multimedia streaming applications called Infopipes. Similar to the P&F style, it provides components such as source, filter, and sinks which are connected directly via ports. Pipes do not exist in this concept. One of its main goals is to estimate

[2] http://xmlcalabash.com/

[3] http://www.w3.org/TR/xproc/

[4] https://github.com/tinkerpop/pipes/wiki

[5] http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

[6] https://msdn.microsoft.com/de-de/library/hh228603(v=vs.110).aspx

the overall delay of multiple components connected to a pipeline. For this purpose, Infopipes requires that the cost of communication between such components must be insignificant. In this way, the performance can be estimated in advance by summing the cost of each component in isolation. For this reason, expensive communication, like remote communication and synchronization, has to be modeled explicitly as additional components. As opposed to that, the P&F style in general and TeeTime in particular aim at encapsulating communication and synchronization in pipes. In this way, performance estimation is still possible while simultaneously avoiding to mix computation and communication.

Sugerman et al. [SFB+09] present a programming model for graphics pipelines called GRAMPS. It supports the definition and execution of pipelines with loops for graphical computational problems. GRAMPS focuses on an efficient execution on both CPU and GPU devices. For this purpose, it not only distinguishes stateless and stateful CPU-stages, but also hardware-specific stages. The latter ones allow computations in GRAMPS to leverage special-purpose hardware, such as particular GPUs. However, GRAMPS does not support composite stages. Furthermore, it does not allow to execute an application on multiple nodes. For TeeTime, we adapted the idea of back-pressuring and stage priorities, especially for the global task pool scheduler described in Section 9.1.2.

StreamIt [TKA02; GTA06] is a high-level programming language for developing large-scale and high-performance streaming applications. Like TeeTime, it allows to model and to execute linear P&F architectures. Furthermore, it supports stage compositions, split-and-join combinations, and feedback loops. However, StreamIt does not support modeling branches. Moreover, it is not designed to execute distributed P&F architectures. In addition, according to the official website,[7] it is not developed any further.

StreamFlex [SPG+07] is a programming model inspired by StreamIt for high-throughput, real-time stream programming in Java. Its main contributions are its concepts to avoid triggering the garbage collector. For this purpose, it uses a custom type system, a custom compiler, and a custom virtual machine. Like TeeTime, it allows to model and to execute linear

[7] http://groups.csail.mit.edu/cag/streamit/index.shtml

and branched P&F architectures. However, it does not support modeling and executing distributed P&F architectures. Moreover, the software architecture of StreamFlex is tailored to the real-time domain. That means, it has an efficient, but fixed scheduling strategy. Its execution environment is similar to the one of the actor model. Each filter can be executed by each thread. Thus, filters always communicate with each other in a synchronized way. For this purpose, StreamFlex uses software transactional memory. To guarantee stringent quality-of-service requirements, StreamFlex uses the Ovm virtual machine [ABC+07].

DANBI [ME13] is another programming model for streaming applications. It is optimized for applications on many-core systems such that it provides a high scalability. Furthermore, it supports irregular workloads, including dynamic input/output rates and feedback loops. Scalability, especially under changing workloads, is achieved by using a global, composite, load-balancing scheduler with a scalable ticket synchronization mechanism. Moreover, DANBI is one of few approaches which includes a termination strategy. In this way, it allows to automatically shutdown the application similar to TeeTime. As opposed to TeeTime, DANBI has a fixed design which hampers extensibility and thus experimentation. For example, DANBI is designed to use blocking synchronization and read-only access to shared states. Lock-free queues or parallel writes to a shared data model are not allowed. Additionally, it does not support a distributed execution. Nevertheless, it would be interesting to implement and to evaluate DANBI's scheduler in TeeTime.

XJava [OPT09] is an object-oriented programming language which extends Java with new constructs to define streaming applications. It enables a declarative pipeline definition and allows for composing multiple stages. Furthermore, it simplifies the implementation of parallel programming patterns by abstracting from explicit manual synchronization. Its associated XJava compiler generates native Java code and the corresponding XJava runtime environment manages the execution. However, XJava does not support multiple ports and thus is not able to define branches or loops. Moreover, although the scheduler can be exchanged by another implementation, it is bound to the concept of the global task pool scheduler. Hence, each stage execution is a task in a task queue which is accessible by all of the threads

in a thread pool.

Thies et al. [TCA07] present an annotation-based approach to define and execute linear P&F architectures in C programs. First, the application developer needs to place some specific macros to divide a given code region into a set of partitions. Then, a corresponding runtime environment performs dynamic analysis in a training phase to identify the data which is shared between each pair of partitions. Finally, the runtime environment executes each partition in a dedicated process by forking the parent process. The individual partitions receive and send data by using pipes as inter-process communication mechanism. Hence, this approach does not need to declare a new type for each stage. However, it only supports modeling linear P&F architectures. Furthermore, it requires a number of training phases to identify which data should be transfered between the partitions.

Fastflow [ADK+14] is a C++ parallel programming framework for streaming applications. It provides a set of composable patterns for parallel programming to structure a given application in a high-level way. Two examples are the pipeline pattern and the task farm pattern. In this way, Fastflow allows to write efficient and portable applications for multi-core systems and accelerators, such as general purpose GPUs. Compared to TeeTime, Fastflow uses the actor model to execute its stages. Hence, each stage is always executed by a dedicated thread. Moreover, Fastflow does not support branches.

15.2 Related Distributed P&F Frameworks

Apache Camel[8] is a Java framework to configure routing and mediation rules using the enterprise integration patterns. It can also be used to assemble P&F oriented systems. However, in contrast to TeeTime, it does neither support typed ports nor the concurrent execution of multiple stages. The official recommendation to handle concurrency is, among others, the usage of a database as synchronization point.[9]

ExplorViz [FWB+13; FRH15; FKH17] is a tool that enables live trace

[8] https://camel.apache.org/

[9] see http://camel.apache.org/parallel-processing-and-ordering.html

visualization for system and program comprehension in large software landscapes. It comprises a P&F-based component that is tailored to the distributed processing of program traces. Hence, it is not suited as a generic P&F framework.

Apache Spark Streaming[10] is a framework to build and to execute distributed, scalable, fault-tolerant, streaming applications in Scala, Java, and Python. It is optimized on live processing continuous big data streams by using so-called *resilient distributed datasets*. According to Zaharia et al. [ZCD+12], Spark Streaming can outperform Hadoop[11] by a factor up to 20.

Akka[12] is a framework for both Scala and Java following the actor model [HBS73]. It focuses on scalability (regarding concurrency and remoting) and fault tolerance. Although it is possible to map Akka's actor-based API to a P&F-based one, Akka is not optimized for the execution of pipeline-based architectures. The mailbox of each actor allows multiple senders and thus needs to be synchronized in a stronger way than necessary for stage-to-stage communication. Moreover, this mailbox is untyped such that each incoming message needs to be checked and casted according to its type. Thus, a faulty connection between actors can only be detected at run time. Finally, Akka has no built-in concept for the end of an execution. The application developer has to design and implement a valid automatic and graceful termination approach by his own.

15.3 Parallel Execution Strategies for Stages

Besides the global task pool strategy (see Section 9.1.2 on Page 81), there are at least two further common strategies (S1 and S2) to execute a pipeline configuration in parallel [SLY+11; SQK+10].

The first strategy (S1) distributes the given threads over a distinct subset of the declared stages. One well-known manifestation of this strategy is to assign a dedicated thread to each stage. For example, Fastflow [ADK+14] follows this approach. However, this naïve assignment has some disadvan-

[10] https://spark.apache.org/streaming

[11] http://hadoop.apache.org

[12] http://doc.akka.io

tages which we have already discussed in Section 2.7 on Page 25. For this reason, TeeTime supports all manifestations of this strategy in the following way. Depending on whether two connected stages are executed by the same thread or by different threads, the corresponding pipe is synchronized or unsynchronized, respectively. Stages are typically not synchronized as each of them is executed by only a single thread.

The major challenge of this strategy lies in finding the optimal assignment of threads to stages. While static assignment approaches [e.g., SPG+07; SLY+11] are usually more efficient for stable and predictable pipeline configurations, dynamic assignment approaches [e.g., SQK+10; ME13] can additionally handle imbalanced stages.

The second strategy (S2) assigns a copy of the whole pipeline structure to each thread. Each thread maintains a sorted list of all available pipes. Moreover, each thread uses a scheduler that iteratively takes one of the last stages of the pipeline whose input pipes are non-empty. This strategy does not only require pipes, but also stateful stages to be synchronized since the copies of a stage usually share one single state.

Some approaches [SLY+11; NAT+09] use work-stealing pipes to balance the workload across all available threads. Such pipes then require multiple-producer/multiple-consumers data structures that cause additional runtime overhead due to further synchronization and management effort. Moreover, Kumar et al. [KFB+12] show in a case study that the steals-to-task ratio is often below 0.1%, i.e., steals are extremely rare.

15.4 Related Architectural Styles

Besides the original P&F style [SG96; TMD09], related architectural styles are map-reduce architectures and actor-based architectures.

Map-reduce [DG08] is an architectural style and a programming model for processing and generating big data sets on a large cluster of commodity machines. With TeeTime, we focus on modeling and executing P&F-based architectures for multi-core systems. Conversely to map-reduce, we only provide a rudimentary distributed execution environment which is by far not optimized to such an extent.

The actor model [Agh86] is a mathematical model that was originally designed to reason about concurrent computations for artificial intelligence. It employs actors as independent, concurrently running computation entities that communicate with each other by message passing. Hence, the actor-model can be categorized as event-based architectural style that is often used for building fault-tolerant distributed systems. Instead of decomposing the computational workload, we follow the *separation of concerns* design principle to improve the reusability and the maintainability. Moreover, we achieve an increased throughput and a higher scalability by restricting the communication to single-producer/single-consumer scenarios.

Part IV

Conclusions and Future Work

Chapter 16

Conclusions

In this thesis, we discussed open challenges with respect to the modeling and the execution of P&F architectures. For example, we miss a P&F framework which is extensible and supports arbitrary P&F architectures. Furthermore, we see a high potential for future work when it comes to an efficient P&F execution on multi-core and many-core systems. Hence, we developed and presented our generic and parallel P&F framework TeeTime. On the one hand, it can be used by software developers to build and to run arbitrary P&F-based applications in an efficient way. On the other hand, it provides a platform for researchers to experiment with the P&F style. Additionally, we showed TeeTime's capabilities in several, different experiments and case studies from both research and industry. In the following Sections 16.1- 16.3, we summarize the thesis according to TeeTime's modeling concepts, execution concepts, and development concepts. Moreover, we provide an overview of our corresponding evaluation results in Section 16.4.

16.1 Modeling Concepts

We developed TeeTime according to the architectural description by Shaw [Sha89] as well as Allen and Garlan [AG92]. Each TeeTime-based application is built upon stages, pipes, ports, and configurations. These entities in combination enable a modular software architecture and thus encourage type-safety, extensibility, and reusability. Moreover, such an architecture can also ease the navigation in code for software developers.

Unlike previous work, we also presented and discussed different possible realizations of each P&F entity. Simultaneously, we considered their resulting impact on the execution behavior, especially on the performance.

For example, we allow a stage to adapt its number of ports at runtime because a fixed set of ports would prevent to react on varying workload (see our task farm stage in Section 9.2). Additionally, we discussed whether a stage may contain synchronized sections, because such sections are prone to errors and can have a huge impact on the performance. We also differentiate pipes by their capability to communicate (unsynchronized, synchronized, and distributed) and presented different implementations (e.g., based on an SP/SC or an MP/MC queue).

Furthermore, we considered the P&F style itself in more depth than previous work. For example, we successfully realized type-safety with two different compile-time approaches: by using Java generics and by using C++ template parameters. Since compile-time checking is not possible in some cases, as explained in Section 8.2.2, we also presented a type-check mechanism which works at run-time. Moreover, TeeTime's extensibility goes beyond that of stages. It also allows to integrate further pipes or alternative schedulers with minimal implementation effort. Similarly, TeeTime does not only support reusability for primitive stages, but at several different layers. For example, it allows to encapsulate multiple stages to a composite one in order to hide complexity. Furthermore, both reference implementations of the framework itself are freely available as open-source for reuse by others.

In this way, we addressed one of the major lacks of current frameworks and its implementations: the support of arbitrary P&F architectures (see our scientific contribution *SC1: The Generic and Parallel P&F Framework TeeTime* in Section 1.2 on Page 3).

16.2 Execution Concepts

Besides the modeling, we also addressed the execution of P&F architectures. We presented two stage scheduling approaches: our new parallel push/pull (PPP) scheduler and the commonly used global task pool (GTP) scheduler. The PPP scheduler differentiates between active and passive stages in order to determine the thread assignment for a given P&F architecture. Based on this information, it also chooses the required pipe implementation between each two interconnected ports. While executing, a thread does not execute

stages of another thread. Hence, threads do not directly communicate with each other. In doing so, the PPP scheduler causes only a minimal communication effort. Moreover, its execution paths are easy to follow when debugging a running P&F architecture. By contrast, the GTP scheduler allows every thread to execute all of the stages available in the given P&F architecture. Communication between threads is realized by a global, semi-ordered, synchronized task pool. Furthermore, data exchange between ports must always be implemented by MP/MC-based pipes. To compensate this higher synchronization effort, each thread executes a stage multiple times before reinserting it into the pool again. We improved the GTP approach by adding support for stages which produce all of its elements in one single execution (see Section 9.1.2 on Page 82). Hence, we provide one new and one enhanced stage scheduling approach for the execution of P&F architectures (see our scientific contribution *SC2: A New Filter Scheduling Approach* in Section 1.2 on Page 4).

Besides the scheduling, we introduced a modular, incremental, and generic approach to introduce parallelism in arbitrary P&F architectures. Our main idea is to provide a composite stage that is wrapped around an existing stage in order to parallelize it. We call this composite stage the Task Farm Stage (TFS) since it utilizes the Task Farm Parallelization Pattern. In combination with our self-adaptive manager, the TFS is able to scale with the number of cores and with the incoming workload, even if it changes at runtime (see our scientific contribution *SC3: A New P&F Parallelization Approach* in Section 1.2 on Page 5).

We also discussed several concepts and realizations to execute P&F architectures in a distributed fashion. Although we focused on the execution on multi-core and many-core systems, we nevertheless developed an approach based on the actor-model. Furthermore, it automatically deploys the distinct parts of the distributed P&F architecture on the associated nodes.

Finally, we discussed the modeling and the execution of feedback loops in P&F architectures. Although they provide an easy way to implement recursive functions in P&F architectures, we do not recommend them. Their modeling requires careful consideration in terms of scheduling, termination, and stage design. Nevertheless, we allow to declare loops with TeeTime, especially for research reasons.

16.3 Development Concepts

To support the modeling process itself, we also presented development concepts to ease the definition of stages and P&F configurations. Moreover, we presented some tools and techniques to improve the debugging of P&F configurations. More precisely, we showed that TeeTime provides a minimal, but sufficient API to develop arbitrary stages. A corresponding example stage in Java gave an impression of how to declare multiple ports, the execution logic, and optionally handlers for the initialization and the termination event. We also developed and discussed a stage DSL that should further ease the writing of stages. However, we came to the conclusion that its development and its maintenance are too laborious. In contrast, our internal DSL for testing stages showed positive results. It (1) hides the configuration and the execution from the test, and (2) focuses on the input and the output of the stage. In this way, the DSL relieves the application developer when writing stages and thus reduces the potential for errors. Concerning the definition of P&F configurations, we provided three approaches with TeeTime. The most verbose approach is the object-oriented, programmatic one which uses the syntax and semantics of the underlying programming language. It is flexible enough to define branches and loops. An alternative approach employs an internal DSL similar to the one for writing stage tests. It is good at defining linear P&F structures. The last approach uses an external DSL and allows to generate the configuration into arbitrary target programming language syntax. In particular, it allows to define and to generate distributed configurations. We recommend to use the external DSL approach if a generator for the target programming language and a corresponding IDE plugin is available. As proof-of-concept implementation, we provide a generator to Java and an associated Eclipse plugin as open source.[1]

When developing software, it is crucial to have supporting tools and techniques which ease in debugging the software. Hence, we presented a programmatic approach and a DSL-based approach to monitor the performance of TeeTime-based applications. The programmatic approach relies on probes which are hard coded into the base stage of TeeTime. In this way,

[1] https://build.se.informatik.uni-kiel.de/teetime/teetime-distributed-dsl

all of the stages, which are implemented with TeeTime, can automatically be monitored without any additional effort by the application developer. This approach allows to measure not only the execution time, but also the waiting time of each stage. In this way, it enables a fine-grained analysis of concurrently running stages. As opposed to the programmatic approach, the DSL-based approach relies on probes which are instrumented via aspect-oriented programming (AOP). This allows to also instrument applications which were not written with TeeTime. Moreover, it is not necessary to have access to the source code in order to monitor the application's behavior. Hence, the DSL-based approach is less intrusive, has a more compact notation, and allows to monitor component-based applications. However, it is less accurate than the programmatic approach because it cannot distinguish waiting times from execution times.

Besides this debugging support, we also presented two new behavior diagrams to support the profiling of TeeTime-based applications (see our scientific contribution *SC4: Live Visualization of TeeTime-based Applications* in Section 1.2 on Page 5). Each diagram visualizes the live execution in a different way. The execution behavior diagram shows the components' behavior either from the current point in time, or from a past point in time. It allows to analyze components in terms of an arbitrary metric at every point in time. As opposed to the execution behavior diagram, the timing behavior diagram represents the individual executions of the components in a way similar to a UML sequence diagram. Thus, it highlights simultaneous executions of different components and, in this sense, supports the identification of concurrency issues.

16.4 Evaluation Results

With our evaluation results, we showed that we successfully fulfilled our research goal of this thesis. That means, with TeeTime, we provide a generic and parallel P&F framework that enhances the development and the execution of P&F-based applications. We evaluated our framework by applying the goal-question-metrics approach with the following four research questions initially stated in Section 7.2:

- ▻ Q1: How to Model Arbitrary P&F Architectures?
- ▻ Q2: How to Provide Reusability and Extensibility by the Framework?
- ▻ Q3: How to Execute Arbitrary P&F Architectures?
- ▻ Q4: How to Efficiently Parallelize Arbitrary P&F Architectures?

To answer these questions, we applied literature reviews, proof-of-concept implementations, lab experiments, and case studies as research methods.

We structured our evaluation according to the five characteristics: language independence, feasibility, performance, reusability, and extensibility. The goal of the language independence evaluation was to show that TeeTime's framework architecture is not bound to the features of a particular programming language. For this purpose, we considered the similarities and the differences of our two reference implementations in Java and in C++. The evaluation shows that both of them successfully realize the general language-agnostic framework architecture presented in Chapter 8, albeit they partly use different techniques for this. Hence, with this evaluation, we provide answers to our research question Q1 (*How to Model Arbitrary P&F Architectures?*).

For the feasibility evaluation, we conducted five case studies which altogether cover both various technical and various domain-specific aspects. They show that TeeTime can model and execute several different P&F architectures, including linear pipelines, branches, and loops. Moreover, they show that TeeTime can be employed in several different research and industrial fields of application. Hence, with this evaluation, we provide answers to our research question Q1 (*How to Model Arbitrary P&F Architectures?*) and Q3 (*How to Execute Arbitrary P&F Architectures?*).

Our performance evaluation was one of the major parts within our evaluation. We considered the overhead of TeeTime's Java implementation which results from building and executing P&F architectures with it. We evaluated the three most performance-critical framework parts: the unsynchronized pipe, the synchronized pipe, and the composite stage. To increase the external validity, we performed this evaluation on three different multicore processor architectures. The results show that we can safely neglect the overhead of the unsynchronized pipe. The overhead of the synchronized

pipe can be heavily reduced on cache-coherent multi-core systems if we use an implementation which is lock-free and restricted to a single producer and a single consumer thread. We showed that the lock-free SP/SC pipe used by TeeTime has by far the lowest runtime overhead among all of the other 11 pipes which we have evaluated. Finally, we considered the overhead of composite stages. We could show that the composition does not add any overhead to the execution. Thus, TeeTime provides a way of modularization and abstraction without negatively impacting performance.

Besides evaluating TeeTime's performance-critical parts, we also compared the performance of our two execution models *Parallel Push/Pull Scheduling (PPP)* and *Global Task Pool Scheduling (GTP)*. Our results showed that our new PPP scheduler can greatly keep up with the commonly used GTP scheduler. If GTP is not properly configured with regard to its executions per task, PPP can outperform GTP in a significant way. Furthermore, the implementation of PPP requires considerably less effort. In return, PPP requires to declare stages as active by the application developer in order to utilize multi-core systems. In our scenarios, we recommended to use PPP or GTP32 (with 32 executions per task) because both schedulers showed the same efficiency. This result is noteworthy on its own because PPP and GTP32 are based on entirely different concepts.

Finally, we investigated the performance of our parallelization approach based on the task farm stage (TFS) and the associated self-adaptive manager (SAM). In all of our four different scenarios, we showed that the TFS increases the overall throughput and that the SAM automatically adapts the number of stages according to the current runtime workload. Hereby, we observed that the right choice of the prediction algorithm and the throughput boundary are crucial for a high overall throughput. In our benchmarks, the regression algorithm consistently performs best if it is used with a throughput boundary between 0 and 0.225.

Hence, with this comprehensive performance evaluation, we provide answers to our research question Q3 (*How to Execute Arbitrary P&F Architectures?*) and Q4 (*How to Efficiently Parallelize Arbitrary P&F Architectures?*).

In our last evaluation, we investigated TeeTime in terms of its reusability and its extensibility. On the one hand, we collected and considered the number of reused and extended TeeTime entities and stages from the

P&F architectures described in the feasibility evaluation and the overhead evaluation. On the other hand, we discussed and justified which TeeTime entities and stages belong to the public API and which are hidden from the application developer. Additionally, we described by means of two case studies how project-external developers have successfully extended TeeTime. Our results showed that TeeTime provides sophisticated support for reusability and extensibility. Its architecture hides internal details and simultaneously is open for modifications. Hence, with this evaluation, we provide answers to our research question Q2 (*How to Provide Reusability and Extensibility by the Framework?*).

Chapter 17

Future Work

As described in this thesis, TeeTime is able to model and to execute arbitrary P&F architectures. Nevertheless, there are many open issues left for future work. In the following, we describe some of the most interesting issues.

17.1 Enhanced Modeling

Our task farm stage (TFS) is able to duplicate a stage and thus to parallelize its execution. However, the TFS can currently only duplicate stages which have at most one input port and at most one output port. We limited our focus on these cases because they are very common and less complex to handle. Multiple input or output ports would require a more sophisticated distribution and merge concept. For example, consider a stage with two input ports which only produces output elements if it consumes one element from each input port. A distribution approach has to take this particular requirement into account if it should still be efficient and not lead to a deadlock or livelock situation.

17.2 Execution on Many-Core Systems

In this thesis, we presented two execution models which utilize the capabilities of modern multi-core systems. Moreover, we also presented an approach to execute P&F architectures in a distributed way. However, we have not considered the execution on many-core systems, such as GPUs and FPGAs. Data parallel computations could benefit from such hardware where CPUs are not optimal for. Some researchers [SFB+09; BDL+13b] have

already worked on a combination of multi-core and many-core systems in the context of streaming applications. One major challenge in this area is to identify which stages are data-parallel and which are computation-intensive. So far, the application developer is forced to declare a stage accordingly. Often, he is also forced to use a different programming language or a special API to access a particular many-core system. It would be interesting to evaluate whether the identification process could be automatized or whether the access to a heterogeneous system could at least be unified.

Concerning the task farm stage, it would be interesting to automate the duplication process without explicitly declaring any task farm stage. This approach would dramatically ease the use of TeeTime since the application developer would not need to place task farm stages anymore. If the slowest stage is already executed by a dedicated thread, TeeTime could wrap a TFS around that stage at runtime. In this way, our framework could duplicate any stage autonomously—provided that multiple instances of the stage can be efficiently executed in parallel. For this purpose, TeeTime has to be extended in order to analyze and adapt each stage accordingly. It is important to note that the performance should not suffer significantly from such an approach.

Furthermore, there are many smaller tasks which are worth to be considered. Although we provide some distribution strategies for the distributor stage, they are by far not optimal in terms of performance. Thus, load balancing could be improved by optimizing our strategies and by implementing additional ones. For example, it would be interesting to investigate and compare strategies based on work-stealing. Good starting points are the works of Chase and Lev [CL05] and of Agrawal et al. [ALS10]. Analogously, the scheduling could be improved. In particular, different execution models could yield better throughput measurements in particular P&F configurations or parts of P&F configurations. Hence, it would also be interesting to combine multiple execution models for the execution of a single P&F configuration.

17.3 Extended Implementations

In this thesis, we developed a general framework architecture for the modeling and the execution of arbitrary P&F-based applications. To show its generality, we implemented this framework architecture in two different object-oriented programming languages, namely Java and C++. However, the C++ implementation does not yet cover all of the features which the Java implementation provides. For an overview of missing features, we refer to Table 11.3 and to the official website.[1] Thus, future work could address these features. Furthermore, additional implementations in different programming languages could be added.

17.4 Empirical Evaluation

In this thesis, we showed by a feasibility evaluation that the TeeTime framework is able to model and to execute arbitrary P&F architectures. Moreover, we performed different evaluations to show that TeeTime is language-agnostic, has a low overhead, can efficiently execute stages in parallel, and provides a high reusability and extensibility. Nevertheless, we see some potential for future work. For example, it would be interesting to perform controlled experiments with humans to evaluate to what extent the DSL for P&F configurations improves the definition of P&F architectures. Furthermore, controlled experiments with humans would determine to what extent our visualizations provide benefits for debugging and profiling P&F architectures.

17.5 Migrate Existing Custom P&F Implementations to TeeTime

As described in the introduction (see Section 1.1), there are still many custom P&F framework implementations which at first serve their purpose, but in the end lack in some features or necessary quality requirements. If we

[1] https://github.com/teetime-framework/TeeTime-Cpp#known-limitations

want to migrate those implementations, we could manually re-implement the logic with TeeTime. However, this requires a huge implementation effort and leads to a high potential for errors. As an example, we have replaced Kieker's P&F framework with TeeTime. We refer to Kieker's analysis framework in version 1.14 for the corresponding reimplementation.

Instead, we propose a semi-automatic, pattern-based approach to detect and to transform arbitrary P&F architectures to TeeTime-based P&F architectures. The approach, called PARROT, allows to migrate existing custom P&F implementations in order to introduce more flexibility and more parallelism. A first prototype implementation and preliminary evaluation results show that PARROT is promising (see Section 17.5.2). It even turned out that the approach is also suitable for migrating single-core legacy applications to parallel ones. Moreover, the approach could also migrate faulty code regions to correct ones. However, the approach still requires much implementation and research effort which is why we leave it for future work.

17.5.1 PARROT: Approach Overview

The approach provides seven consecutive steps which lead through the migration process. As input, it requires the desired custom P&F implementation as well as a set of predefined patterns to detect and to transform this implementation. As shown by Figure 17.1, the migration process proceeds as follows.

In the first step (S1), we transform the source code of the given application to a system dependence graph (SDG) [HRB90] and store it in a graph database. In the second step (S2), we enrich the SDG by runtime information to allow a detailed analysis of the P&F implementation (cf. other dynamic approaches such as [TF10] and [HAJ+16]). We instrument the given application by an application monitoring framework, such as Kieker [HWH12], and run it multiple times with representative input values. Thereby, we record the actual runtime behavior of the application in terms of metrics, such as a method's execution time or the frequency of a method call. We aggregate the recorded information by annotating the response times (e.g., median and average) as properties at the corresponding nodes and edges in our SDG. In this way, we have direct access to both static and dynamic

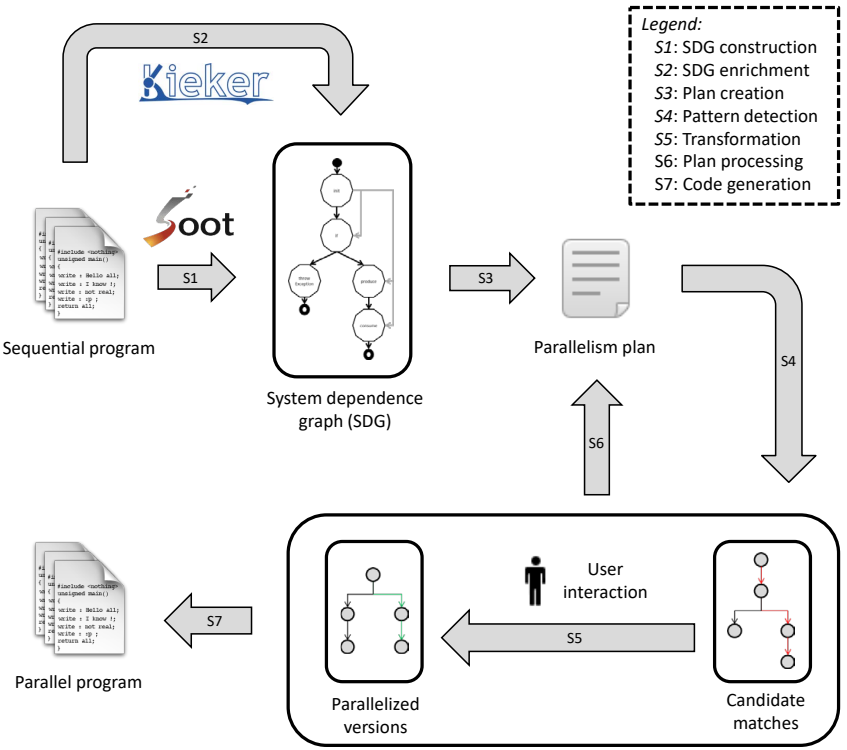


Figure 17.1. Overview of the P&F migration approach PARROT

information in one single unified data structure which effectively eases the pattern definition.

In the third step (S3), we apply a pre-filtering to reduce the search space and to prioritize those code sections which are relevant for the migration process. For this purpose, we create a migration plan that represents an ordered list of such code sections. The list is sorted by user-defined criteria based on the information stored in the SDG. For example, possible criteria are heavily used code regions, the aggregated method execution time of multiple runs, the length of the critical path, or the self-parallelism [GJL+11].

These criteria are in particular useful if the P&F implementation does not support a parallel execution. Then, a transformation to TeeTime would significantly increase the performance of such code regions.

After building the SDG and the migration plan, the workflow of our approach is as follows. The migration engineer triggers the pattern-matching of all predefined detection patterns on the SDG for the first code section in the migration plan (S4). For each match which requires a human decision (S5), the migration engineer is asked to either reject or approve it for migration. Upon rejection, the match is skipped. Otherwise, the match is transformed into a TeeTime-based version indicated by an associated predefined transformation pattern. For all matches which do not require a user interaction, the transformation is executed automatically. Afterwards, the migration engineer can repeat this workflow with the next code section on the migration plan (S6). Alternatively, the current version of the SDG can be generated to the original programming language syntax (S7). To preserve the structure and non-code elements, such as comments and annotations, of untouched code regions, our SDG also incorporates this information. For a unified indentation, we do not encode it into the SDG, but rather use an automatic code formatter.

In summary, we think that PARROT could provide a great extension to TeeTime. While TeeTime targets the lack of a generic and parallel P&F framework, PARROT targets the lack of a systematic and efficient P&F migration approach. However, as described above, PARROT still requires much work. For this reason, we especially recommend to focus on this particular research area.

17.5.2 Preliminary Works Related to PARROT

Related Publications

▻ [Wul14] Christian Wulf. “Pattern-based Detection and Utilization of Potential Parallelism in Software Systems.” In: *Proceedings of the Software Enginnering*. 2014

In this work, we present an overview of our semi-automatic parallelization approach PARROT. The approach itself covers more than the de-

tection and parallelization of P&F architectures. It can be applied for refactoring or for parallelizing arbitrary code sections in object-oriented applications.

▻ [WH15] Christian Wulf and Wilhelm Hasselbring. "Software Performance Anti-Patterns Observed and Resolved in Kieker." In: *Symposium on Software Performance 2015: Joint Developer and Community Meeting of Descartes/Kieker/Palladio*. Vol. 35. 5. Softwaretechnik-Trends, Nov. 2015, pp. 38–40

In this work, we identify and resolve three performance anti-patterns in Kieker. These patterns address an optimized data flow and an improved way of asynchronous communication to increase the throughput. They build the basis for our detection and transformation patterns in PARROT.

Co-Supervised Student Works

▻ [Fin15] Dean Jonas Finkes. "Entwicklung einer Call-Graph-Transformation von Soot zu Neo4j." Studienarbeit. Department of Computer Science, Jan. 2015

Finkes presents the first prototype implementation of PARROT's phase one. In particular, he describes the software architecture and his evaluation of the transformation from the source code to the system dependency graph. For this purpose, he uses the dataflow analysis framework Soot [Soo] and the graph database system Neo4J [Neo16].

▻ [Blü15] Lars Erik Blümke. "Konzeption und Implementierung eines Eclipse-Plugins zur Erstellung von graphbasierten Quellcodemustern." Bachelor's thesis. Department of Computer Science, Oct. 2015

Blümke presents the first concept and prototype implementation for a graphical editor as Eclipse-plugin that allows for defining graph-based source code patterns for PARROT.

▻ [Kra15] Johanna Elisabeth Krause. "A pattern-based transformation approach to parallelise software systems using a system dependency graph." Master's thesis. Department of Computer Science, Dec. 2015

Krause presents the first concept and prototype implementation for PARROT's pattern detection and pattern transformation phases.

▻ [Kop16] Erik Koppenhagen. "GUI-based automated generation of Neo4j Cypher queries for candidate patterns and parallelization patterns." Master's thesis. Department of Computer Science, Sept. 2016

Koppenhagen presents the first concept and prototype implementation for an automatic generation of Neo4J queries for patterns built with PARROT's graphical editor [Blü15]. As a result, patterns can be defined in their natural forms as system dependency graphs without requiring knowledge about Neo4J's query language Cypher.

▻ [Fin16] Dean Jonas Finkes. "A hierarchical Eclipse-based editor for system dependency graphs." Master's thesis. Department of Computer Science, Oct. 2016

Finkes presents an extension to PARROT's graphical editor [Blü15]. He adds hierarchy to the visualization of system dependency graphs and related patterns. As a result, composite nodes can be expanded and collapsed such that a user can understand and work even with large graphs.

▻ [Jäh16] Daniel Thorben Jähde. "Quellcode-unterstützte Musterdefinition für Systemabhängigkeitsgraphen." Master's thesis. Department of Computer Science, Nov. 2016

Jähde presents an extension to PARROT's graphical editor [Blü15]. He adds support for automatically importing the system dependency graph of a particular source code segment. The extension allows a user to select a code region from within the Eclipse and to visualize the corresponding graph in PARROT's editor. Consequently, a parallel pattern expert can use the programming language of his choice instead of the SDG to define detection and transformation patterns.

▻ [Bar16] Alexander Barbie. "Runtime information integration into system dependency graphs." Studienarbeit. Department of Computer Science, 2016

Barbie presents a prototype implementation to integrate Kieker logs into PARROT's system dependency graph, so that patterns can also exploit dynamic information.

Part V

Appendix

Appendix A

Case Study: Distributed Execution

Table A.1. The timeline protocol of the TCP-based execution which generates and processes 5,000 strings each with a length of one.

LN	Timestamp	Master	Worker 1	Worker 2	Worker 3
1	15:48:52,317		Worker sys. created		
2	15:48:52,872			Worker sys. created	
3	15:48:53,327				Worker sys. created
4	15:48:53,844	Master sys. created			
5	15:48:53,848	Master actor created			
6	15:49:02,951		Worker Actor created		
7	15:49:03,417	Registered Node1			
8	15:49:03,421	Deployed Node1			
9	15:49:03,435		Deploy msg received		
10	15:49:03,435		Fetched Node1 config		
11	15:49:03,441		Receiver requested		
12	15:49:03,458		Execution created		
13	15:49:04,476		Receiver requested		

14	15:49:05,496		Receiver requested		
15	15:49:06,364				Created Worker Actor
16	15:49:06,516		Receiver requested		
17	15:49:06,585	RegisteredNode3			
18	15:49:06,585	Deployed Node3			
19	15:49:06,599				Deploy msg received
20	15:49:06,600				Fetched Node3 config
21	15:49:06,604				Registered receiver
22	15:49:06,616				Execution created
23	15:49:06,624	Waiting for more worker...			
24	15:49:06,878			Created Worker Actor	
25	15:49:07,102	Registered Node2			
26	15:49:07,103	Deployed Node2			
27	15:49:07,120			Deploy msg received	
28	15:49:07,121			Fetched Node2 config	
29	15:49:07,129			Receiver requested	
30	15:49:07,129			Registered receiver	
31	15:49:07,142				Receiver requested by Node2
32	15:49:07,143			Execution created	
33	15:49:07,163			Receiver ActorRef resolved	
34	15:49:07,164			All sender ready	

35	15:49:07,536		Receiver requested		
36	15:49:07,547			Receiver requested by Node1	
37	15:49:07,568		Receiver ActorRef resolved		
38	15:49:07,568		All sender ready		
39	15:49:07,571	Start signal to Node1			
40	15:49:07,572	Start signal to Node2			
41	15:49:07,572	Start signal to Node3			
42	15:49:07,574		Execution started	Execution started	Execution started
43	15:49:07,574		Signal sent		
44	15:49:07,590		Element sent		
45	15:49:07,595			Received start signal	
46	15:49:07,596			Signal sent	
47	15:49:07,602		Element sent		
48	15:49:07,603			Received element	
49	15:49:07,605			Received element	
50	15:49:07,628			Element sent	Received start signal
51	15:49:07,632				Received element
52	15:49:07,640			Element sent	
53	15:49:07,642				Received element
54	...	...	...	...	...
55	15:49:51,739		Element sent		
56	15:49:51,739			Received element	
57	15:49:51,740		Signal sent	Element sent	
58	15:49:51,743			Received TerminatingSignal	

59	15:49:51,744			Poison pill (Receiver)	
60	15:49:51,747		Poison pill (Sender)		
61	15:49:51,748		Terminate worker		
62	15:49:51,749		Execution terminated	Element sent	
63	15:49:51,749		Poison pill (Worker)		Received element
64	15:49:51,749			Signal sent	
65	15:49:51,750	Worker Node1 ack shutdown			
66	15:49:51,752				Received TerminatingSignal
67	15:49:51,753				Poison pill (Receiver)
68	15:49:51,754				Terminate worker
69	15:49:51,756			Poison pill (Sender)	
70	15:49:51,757			Terminate worker	
71	15:49:51,757			Execution terminated	
72	15:49:51,757			Poison pill (Worker)	
73	15:49:51,758	Worker Node2 ack shutdown			
74	15:49:53,488				Execution terminated
75	15:49:53,488				Poison pill (Worker)
76	15:49:53,489	Worker Node3 ack shutdown			
77	15:49:53,490	Poison pill			
78	15:49:56,874				Shutdown system
79	15:49:56,876		Shutdown system		
80	15:49:56,879			Shutdown system	

81	15:49:56,881	Shutdown system			

Table A.2. The timeline protocol of the execution which gracefully terminates due to an exception in a stage from Worker 1 (see LN 1).

LN	Timestamp	Master	Worker 1	Worker 2	Worker 3
Distributed configuration running normally...					
1	17:20:43,323		**TeeTime crashed**		
2	17:20:43,343	**Error on Worker**			
3	17:20:43,348	Send shutdown signal to worker 3			
4	17:20:43,348	Send shutdown signal to worker 2			
5	17:20:43,354	Worker 3 acknowledged shutdown			
6	17:20:43,355	Worker 2 acknowledged shutdown			
7	17:20:43,951			Execution aborted	
8	17:20:44,362				Execution aborted
9	17:20:45,225		Shutdown Actor system		
10	17:20:45,230	Shutdown Actor system			
11	17:20:45,822			Shutdown Actor system	
12	17:20:46,234				Shutdown Actor system

Appendix B

Performance Evaluation

B.1 Framework Overhead Evaluation

B.1.1 Overhead of the Unsynchronized Pipe

See

▻ Table B.1

▻ Figure B.1

▻ Figure B.2

B.1.2 Overhead of the Synchronized Pipe

See

▻ Figure B.3

▻ Figure B.4

Table B.1. Hardware and software environment of the framework overhead evaluation.

CPU name	Clock Freq.	Cores	RAM	OS	JRE
Intel Xeon E5-2650	2.8 GHz	16 (2x8)	128 GB	Debian 3.16.7	1.8.0_51
AMD Opteron 2384	2.7 GHz	8 (2x4)	16 GB	Debian 3.16.7	1.8.0_51
Oracle UltraSparcT2+	1.4 GHz	16 (2x8)	64 GB	Solaris 11.2	1.8.0_51

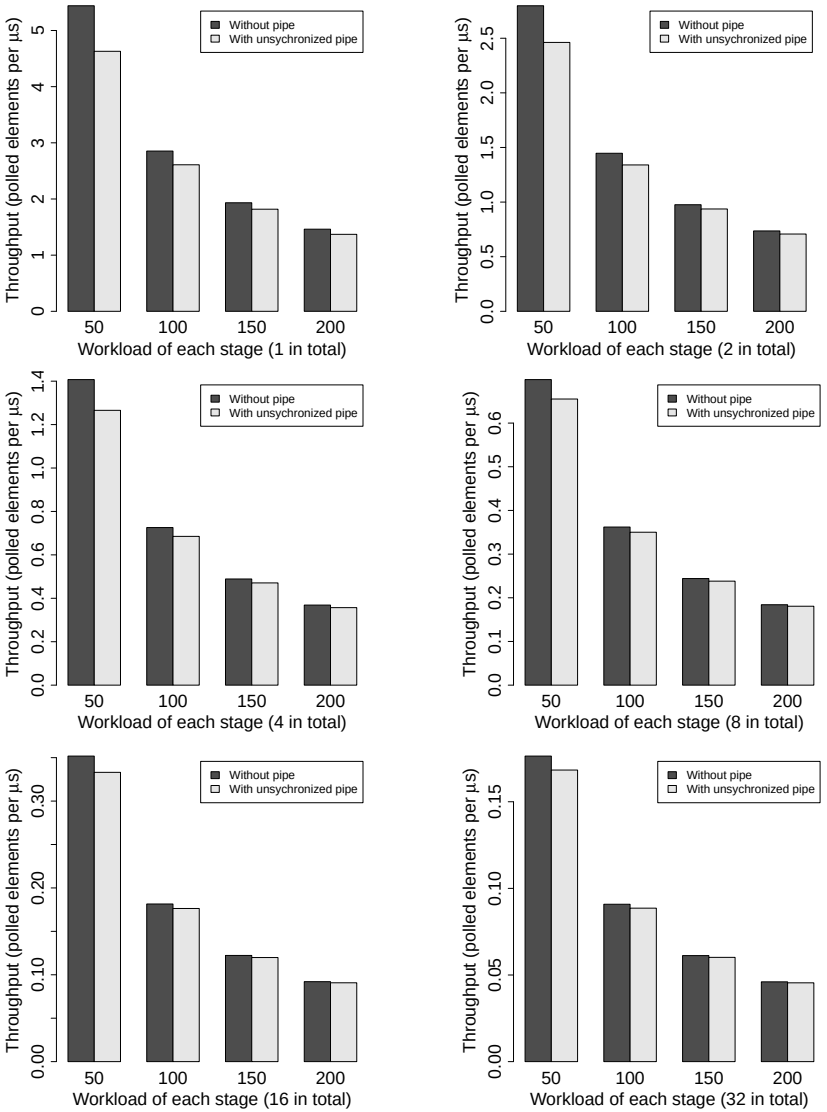


Figure B.1. The pipeline implementations without pipes and with unsynchronized pipes on AMD's Opteron in comparison. Each diagram shows the results for a different pipeline size.

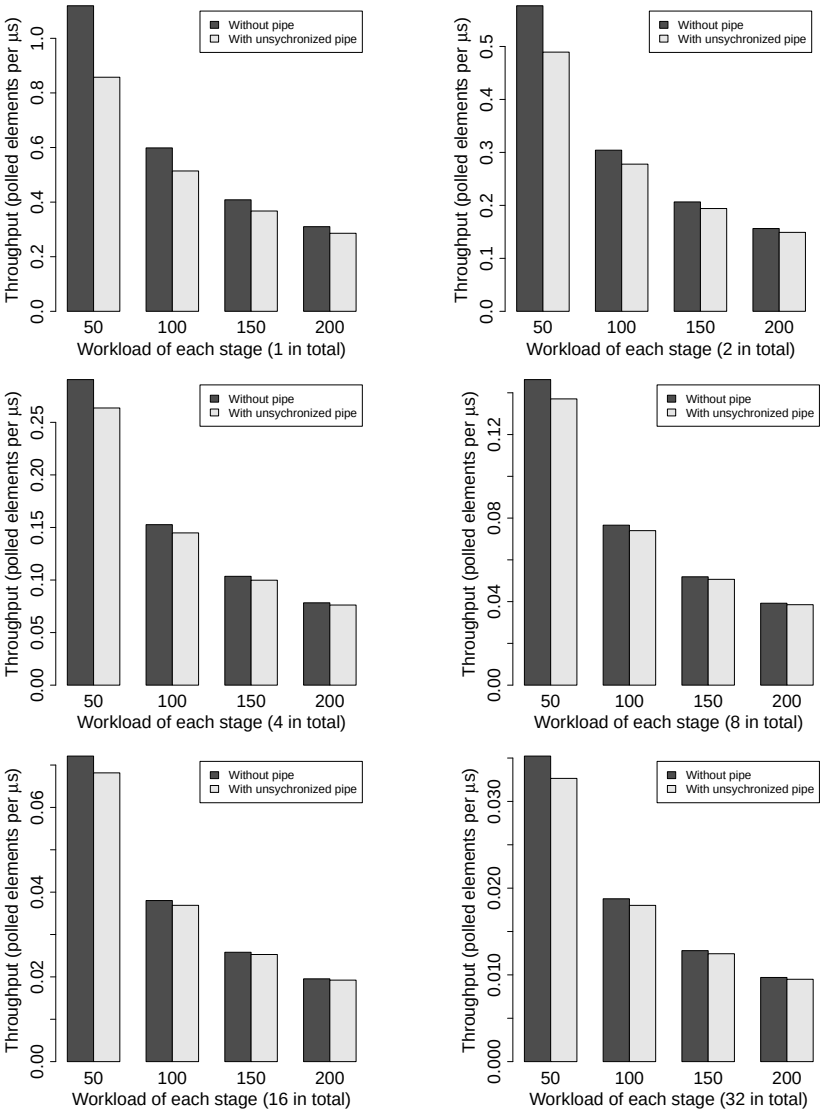


Figure B.2. The pipeline implementations without pipes and with unsynchronized pipes on Oracle's UltraSparcT2+ in comparison. Each diagram shows the results for a different pipeline size.

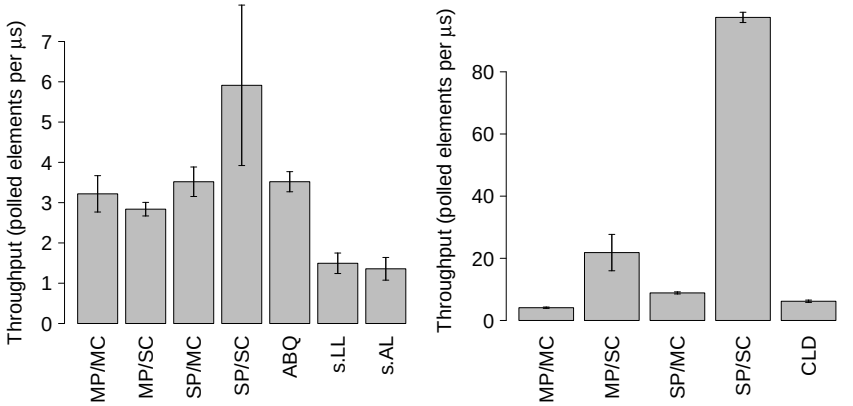


Figure B.3. Throughput evaluation of various synchronized pipe implementations on AMD's Opteron. The left diagram shows the results for the blocking pipes, while the right diagram shows the results for the lock-free pipes. (ABQ=`ArrayBlockingQueue`, s.LL=`synchronized LinkedList`, s.AL=`synchronized ArrayList`, CLD=`ConcurrentLinkedDeque`)

B.1.3 Overhead of Composite Stages

See

▻ Figure B.5

▻ Figure B.6

B.2 Scheduling Approach Evaluation

See

▻ Table B.2

▻ Table B.3

▻ Table B.4

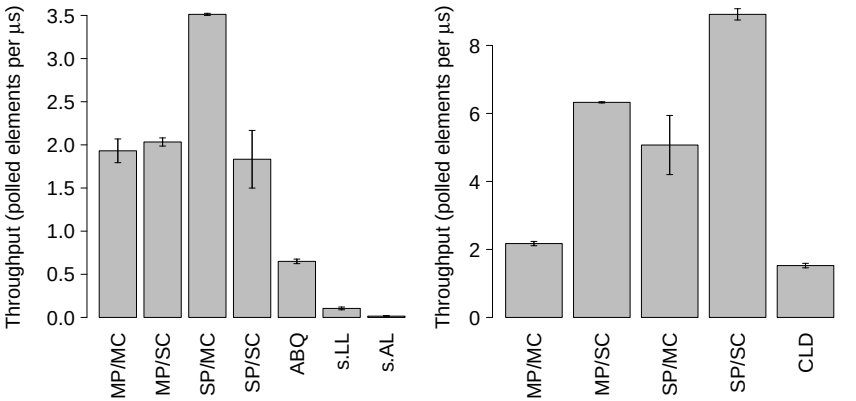


Figure B.4. Throughput evaluation of various synchronized pipe implementations on Oracle's UltraSPARC T2+. The left diagram shows the results for the blocking pipes, while the right diagram shows the results for the lock-free pipes. (ABQ=`ArrayBlockingQueue`, s.LL=`synchronized LinkedList`, s.AL=`synchronized ArrayList`, CLD=`ConcurrentLinkedDeque`)

Table B.2. Hardware and software environment of the scheduling approach evaluation.

CPU name	Clock Freq.	Cores	RAM	OS	JRE
Intel i5-2520M	2.5 GHz	2	8 GB	Windows 10 Pro 64-bit	1.8.0_162
Intel i7-4770K	3.5 GHz	4	16 GB	Windows 10 Pro 64-bit	1.8.0_162

Table B.3. Exact measurement results of the scheduling approach evaluation on the Intel i5 system.

Scheduler	SH_1	SL_1	SH_3	SL_3
PPP	7837.036	7839.597	2976.172	3310.900
↳ ci (99.9%)	± 59.462	± 191.382	± 29.185	± 124.507
GTP32	7992.198	7954.178	3127.880	3029.995
↳ ci (99.9%)	± 450.111	± 115.590	± 57.595	± 21.797
GTP01	7876.783	9307.232	2947.371	5755.240
↳ ci (99.9%)	± 23.000	± 19.143	± 10.602	± 212.843

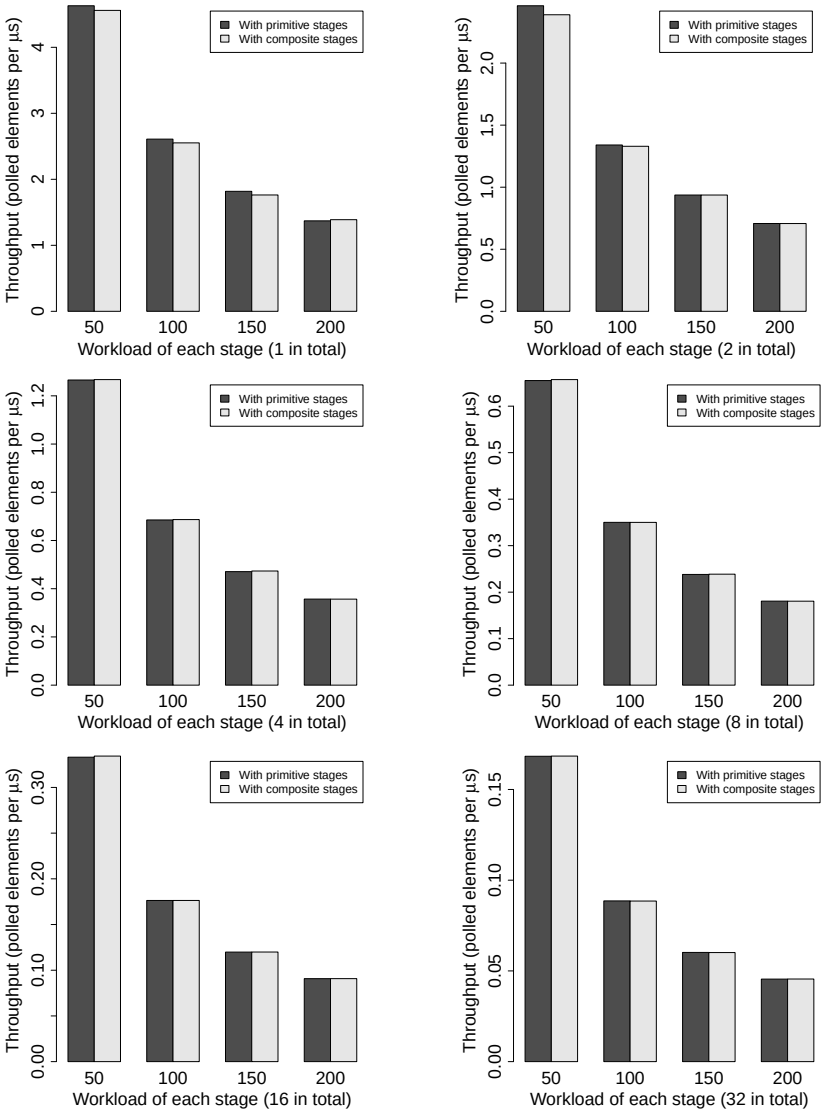


Figure B.5. The pipeline implementations with primitive stages and with composite stages on AMD's Opteron in comparison. Each diagram shows the results for a different pipeline size.

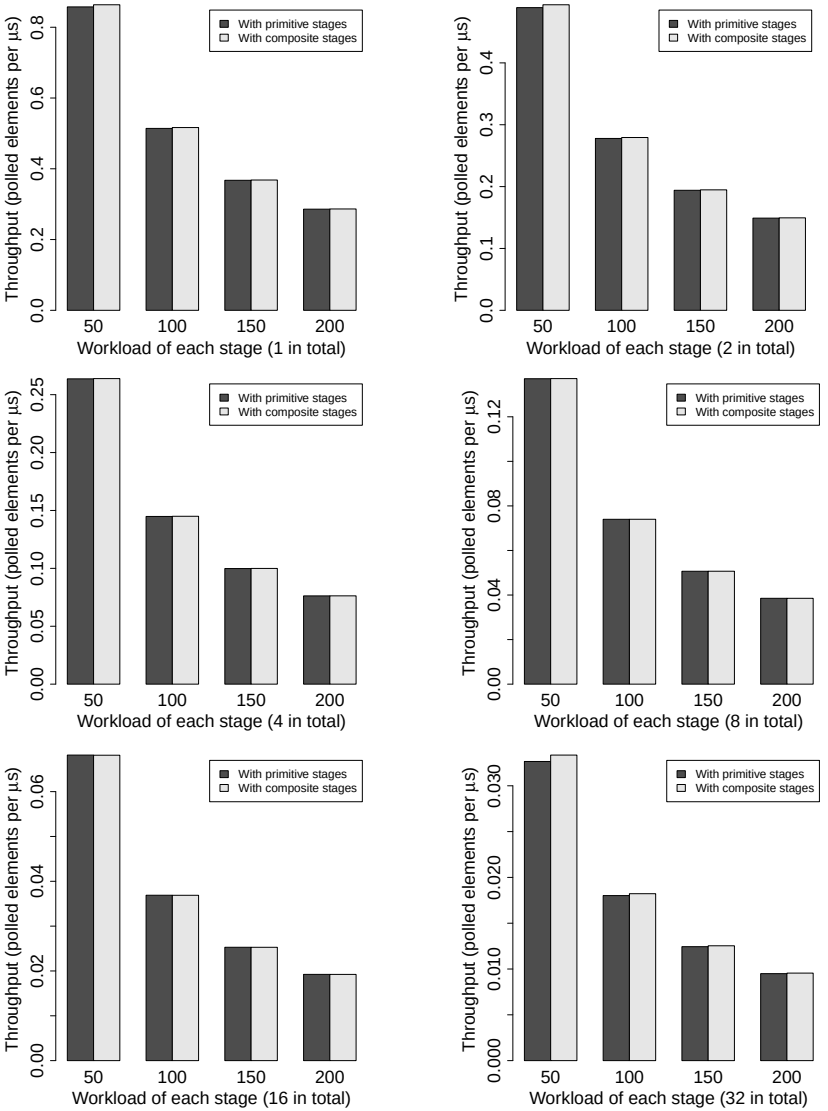


Figure B.6. The pipeline implementations with primitive stages and with composite stages on Oracle's UltraSparcT2+ in comparison. Each diagram shows the results for a different pipeline size.

Table B.4. Exact measurement results of the scheduling approach evaluation on the Intel i7 system.

Scheduler	SH_1	SL_1	SH_3	SL_3
PPP	10004.597	9996.150	3385.553	3381.166
↳ ci (99.9%)	± 2.859	± 10.885	± 14.178	± 15.328
GTP32	9977.305	10100.051	3550.272	3444.895
↳ ci (99.9%)	± 3.673	± 3.078	± 76.045	± 8.225
GTP01	9984.193	11105.845	3365.073	4933.254
↳ ci (99.9%)	± 7.446	± 17.034	± 23.110	± 347.554

Table B.5. Multi-core systems used for the parallelization approach evaluation from Section 13.3.

System	*SUN*	*AMD-I*	*Intel*	*AMD-II*
Processor	UltraSPARC T2+	AMD Opteron 2384	Intel Xeon E5-2650	AMD Opteron 2356
Architecture	SPARC V9 (64 Bit)	x86-64	x86-64	x86-64
# Processors	2	2	2	1
Cores per processor (hardware threads)	8 (64)	4 (4)	8 (16)	4 (4)
Clock/-Core	1.4 GHz	2.7 GHz	2.8 GHz	2.3 GHz
RAM	64 GB	16 GB	128 GB	4 GB
Disk Controller	RAID1/SAS	RAID1/SATA	SATA	RAID1/SATA
OS	Solaris 10	Debian 8	Debian 8	Debian 7

B.3 Parallelization Approach Evaluation

See

▻ Table B.5

Bibliography

[FastFlow] *FastFlow Documentation*. URL: http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:about (visited on 10/22/2015).

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. "Using style to understand descriptions of software architecture." In: *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*. ACM, 1993.

[AAG95] Gregory D. Abowd, Robert Allen, and David Garlan. "Formalizing style to understand descriptions of software architecture." In: *ACM TOSEM* 4.4 (Oct. 1995).

[ABC+07] Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. "A real-time java virtual machine with applications in avionics." In: *ACM Trans. Embed. Comput. Syst.* 7.1 (Dec. 2007).

[AD99] Marco Aldinucci and Marco Danelutto. "Stream Parallel Skeleton Optimization." In: *Proceedings of the International Conference on PDCS*. 1999.

[ADK+12] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. "An efficient unbounded lock-free queue for multi-core systems." In: *Proceedings of the Workshops of the 18th International Conference on Parallel Computing*. 2012.

[ADK+14] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. "Fastflow: high-level and efficient streaming on multi-core". In: *Programming Multi-core and Many-core Computing Systems*. Ed. by Sabri Pllana and Fatos Xhafa. Wiley, Oct. 2014. Chap. 13.

[Ado16] Marc Adolf. “Self-adapting execution of Pipe-and-Filter systems.” Master’s thesis. Department of Computer Science, Mar. 2016.

[AG92] Robert Allen and David Garlan. “Towards Formalized Software Architectures”. In: *Computer Science Today: Recent Trends and Developments. Recent Trends and Developments*. Vol. 1000. Springer-Verlag, 1992.

[AG94] Robert Allen and David Garlan. “Formalizing architectural connection.” In: *Proceedings of the 16th International Conference on Software Engineering*. IEEE Computer Society Press, 1994.

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.

[All13] Jamie Allen. *Effective akka*. O’Reilly Media, Inc., 2013.

[All97] Robert Allen. “A formal approach to software architecture.” Issued as CMU Technical Report CMU-CS-97-144. PhD thesis. Carnegie Mellon, School of Computer Science, Jan. 1997.

[ALS10] K. Agrawal, C.E. Leiserson, and J. Sukha. “Executing task graphs using work-stealing.” In: *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. Apr. 2010.

[Arm13] Joe Armstrong. *Programming erlang: software for a concurrent world*. Pragmatic Bookshelf, 2013.

[Asp] Team of AspectJ. *Aspectj - a framework for aspect-oriented programming in java*. URL: `https://www.eclipse.org/aspectj` (visited on 02/08/2018).

[AVW+93] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent programming in erlang*. 1st ed. Prentice Hall PTR, 1993.

[AZ05] P. Avgeriou and Uwe Zdun. “Architectural patterns revisited - a pattern language.” In: *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*. July 2005.

[Bar16] Alexander Barbie. “Runtime information integration into system dependency graphs.” Studienarbeit. Department of Computer Science, 2016.

[Bar18] Alexander Barbie. “Reporting of performance tests in a continuous integration environment.” Masterarbeit. Institute of Computer Science, Mar. 2018.

[BDL+13a] D. Buono, M. Danelutto, S. Lametti, and M. Torquati. “Parallel patterns for general purpose many-core.” In: *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. Feb. 2013.

[BDL+13b] Daniele Buono, Marco Danelutto, Silvia Lametti, and Massimo Torquati. “Parallel patterns for general purpose many-core.” In: *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. Feb. 2013.

[Bet16] Lorenzo Bettini. *Implementing domain specific languages with xtext and xtend*. 2nd. Packt Publishing, 2016.

[BHW+02] Andrew P. Black, Rainer Huang Jie and Koster, Jonathan Walpole, and Calton Pu. “Infopipes: an abstraction for multimedia streaming.” In: *Multimedia Systems* 8.5 (Dec. 2002).

[BL10] Christopher Brooks and Edward A. Lee. *Ptolemy II - Heterogeneous Concurrent Modeling and Design in Java*. Poster presented at the 2010 Berkeley EECS Annual Research Symposium (BEARS). Feb. 2010.

[Blü15] Lars Erik Blümke. “Konzeption und Implementierung eines Eclipse-Plugins zur Erstellung von graphbasierten Quellcodemustern.” Bachelor’s thesis. Department of Computer Science, Oct. 2015.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., 1996.

[Bor07] Shekhar Borkar. “Thousand core chips: a technology perspective.” In: *Proceedings of the 44th Annual Design Automation Conference*. ACM, 2007.

[CB14] Paul Chiusano and Rnar Bjarnason. *Functional programming in scala*. 1st. Manning Publications Co., 2014.

[CH16] Claus Christian Wiechmann Christian Wulf and Wilhelm Hasselbring. *Experimental data for the paper: Increasing the Throughput of Pipe-and-Filter Architectures by Integrating the Task Farm Parallelization Pattern*. Mar. 2016. URL: https://doi.org/10.5281/zenodo.46776.

[CH89] P. P. Chang and W.-W. Hwu. "Inline function expansion for compiling c programs." In: *SIGPLAN Not.* 24.7 (June 1989).

[CL05] David Chase and Yossi Lev. "Dynamic circular work-stealing deque." In: *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2005.

[Col91] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1991.

[Dan00] M. Danelutto. "Task Farm Computations in Java". In: *High Performance Computing and Networking*. Ed. by Marian Bubak, Hamideh Afsarmanesh, Bob Hertzberger, and Roy Williams. Vol. 1823. Springer Berlin Heidelberg, 2000.

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In: *Communications of the ACM* 51.1 (2008).

[Dit16] Gunnar Dittrich. "Extraction of user behavior profiles for software modernization." Master's thesis. Department of Computer Science, May 2016.

[DL16] Peter J. Denning and Ted G. Lewis. "Exponential laws of computing growth." In: *Commun. ACM* 60.1 (Dec. 2016).

[DW16] Gunnar Dittrich and Christian Wulf. "Extraction of operational workflow-based user behavior profiles for software modernization." In: *Proceedings of the Symposium on Software Performance*. Nov. 2016.

[DWG+12] Xiaoning Ding, Kaibo Wang, Phillip B. Gibbons, and Xiaodong Zhang. "BWS: Balanced Work Stealing for Time-sharing Multicores." In: *Proceedings of the 7th ACM European Conference on Computer Systems*. ACM, 2012.

[Ech17] Florian Echternkamp. "Distributed Pipe-and-Filter architectures with TeeTime." Master's thesis. Department of Computer Science, May 2017.

[EEK+12] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. "Xbase: implementing domain-specific languages for java." In: *SIGPLAN Not.* 48.3 (Sept. 2012).

[EW16] Florian Echternkamp and Christian Wulf. "Kieker in Eclipse - a plug-in for application performance monitoring and dynamic analysis in Eclipse." In: *Proceedings of the Symposium on Software Performance*. Nov. 2016.

[Fin15] Dean Jonas Finkes. "Entwicklung einer Call-Graph-Transformation von Soot zu Neo4j." Studienarbeit. Department of Computer Science, Jan. 2015.

[Fin16] Dean Jonas Finkes. "A hierarchical Eclipse-based editor for system dependency graphs." Master's thesis. Department of Computer Science, Oct. 2016.

[FKH17] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. "Software landscape and application visualization for system comprehension with explorviz." In: *Information and Software Technology* 87 (July 2017).

[Foua] The Apache Software Foundation. *Kafka Library*. URL: `https://kafka.apache.org` (visited on 01/13/2019).

[Foub] The Apache Software Foundation. *Spark Framework*. URL: `http://spark.apache.org/streaming` (visited on 01/13/2019).

[Fouc] The Apache Software Foundation. *Storm Framework*. URL: `http://storm.apache.org` (visited on 01/13/2019).

[Foud] The Linux Foundation. *Kubernetes System*. URL: https://kubernetes.io (visited on 01/13/2019).

[Fow10] Martin Fowler. *Domain specific languages*. 1st. Addison-Wesley Professional, 2010.

[FRH15] Florian Fittkau, Sascha Roth, and Wilhelm Hasselbring. "Explorviz: visual runtime behavior analysis of enterprise application landscapes." In: *23rd European Conference on Information Systems (ECIS 2015)*. May 2015.

[FWB+13] Florian Fittkau, Jan Waller, Peer Christoph Brauer, and Wilhelm Hasselbring. "Scalable and live trace processing with Kieker utilizing cloud computing." In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days 2013*. Nov. 2013.

[FWW+13] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. "Live trace visualization for comprehending large software landscapes: the explorviz approach." In: *1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. Sept. 2013.

[GC08] H. Gonzalez-Velez and M. Cole. "An adaptive parallel pipeline pattern for grids." In: *Proceedings of the IPDPS*. 2008.

[GHJ+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[GJL+11] Saturnino Garcia, Donghwan Jeon, Christopher M. Louie, and Michael Bedford Taylor. "Kremlin: rethinking and rebooting gprof for the multicore age." In: *SIGPLAN Not.* 46.6 (2011).

[GMV08] John Giacomoni, Tipp Moseley, and Manish Vachharajani. "Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue." In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2008.

[GS93] David Garlan and Mary Shaw. "An introduction to software architecture." In: *Advances in Software Engineering and Knowledge Engineering*. Publishing Company, 1993.

[GTA06] Michael I. Gordon, William Thies, and Saman Amarasinghe. "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs." In: *SIGARCH Comput. Archit. News* 34.5 (Oct. 2006).

[Gul15] Mohammed Guller. "Programming in scala." In: *Big Data Analytics with Spark: A Practitioner's Guide to Using Spark for Large-Scale Data Processing, Machine Learning, and Graph Analytics, and High-Velocity Data Stream Processing*. Apress, 2015.

[GY06] Swapna S. Gokhale and Sherif M. Yacoub. "Reliability analysis of pipe and filter architecture style." In: *Proceedings of the 18th SEKE*. 2006.

[HAJ+16] Z. U. Huda, R. Atre, A. Jannesari, and F. Wolf. "Automatic parallel pattern detection in the algorithm structure design space." In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2016.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. "A universal modular actor formalism for artificial intelligence." In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers Inc., 1973.

[Hen16] Sören Henning. "Visualization of performance anomalies with Kieker." Bachelor's thesis. Department of Computer Science, Sept. 2016.

[Hew10] Carl Hewitt. "Actor model for discretionary, adaptive concurrency." In: *CoRR* abs/1008.1459 (2010). arXiv: 1008.1459.

[HHJ+13] Wilhelm Hasselbring, Robert Heinrich, Reiner Jung, Andreas Metzger, Klaus Pohl, Ralf Reussner, and Eric Schmieders. *Iobserve: integrated observation and modeling techniques to support adaptation and evolution of software systems*. Forschungsbericht. Christian-Albrechts-Universität Kiel, Oct. 2013.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. "Interprocedural slicing using dependence graphs." In: *ACM Trans. Program. Lang. Syst.* 12.1 (1990).

[HWH12] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. "Kieker: a framework for application performance monitoring and dynamic software analysis." In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ACM, 2012.

[HWH17] Sören Henning, Christian Wulf, and Wilhelm Hasselbring. "RadarGun: Toward a Performance Testing Framework." In: *Symposium on Software Performance*. Nov. 2017.

[Jäh16] Daniel Thorben Jähde. "Quellcode-unterstützte Musterdefinition für Systemabhängigkeitsgraphen." Master's thesis. Department of Computer Science, Nov. 2016.

[JM10] Natalia Juristo and Ana M. Moreno. *Basics of software engineering experimentation*. 1st. Springer Publishing Company, Incorporated, 2010.

[JW16] Reiner Jung and Christian Wulf. "Advanced typing for the kieker instrumentation languages." In: *Proceedings of the Symposium on Software Performance*. Nov. 2016.

[KFB+12] Vivek Kumar, Daniel Frampton, Stephen M. Blackburn, David Grove, and Olivier Tardieu. "Work-stealing without the baggage." In: *SIGPLAN Not.* 47.10 (Oct. 2012).

[Kie] Team of Kieker. *The kieker project at github*. URL: `https://github.com/kieker-monitoring/kieker` (visited on 02/21/2018).

[KM08] Manjunath Kudlur and Scott Mahlke. "Orchestrating the execution of stream programs on multicore platforms." In: *SIGPLAN Not.* 43.6 (June 2008).

[Kop16] Erik Koppenhagen. "GUI-based automated generation of Neo4j Cypher queries for candidate patterns and parallelization patterns." Master's thesis. Department of Computer Science, Sept. 2016.

[KPP+02] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. "Preliminary guidelines for empirical research in software engineering." In: *IEEE Transactions on Software Engineering* 28.8 (Aug. 2002).

[Kra15] Johanna Elisabeth Krause. "A pattern-based transformation approach to parallelise software systems using a system dependency graph." Master's thesis. Department of Computer Science, Dec. 2015.

[KSA09] Rajesh K. Karmani, Amin Shali, and Gul Agha. "Actor frameworks for the jvm platform: a comparative analysis." In: *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. ACM, 2009.

[Lig] Inc. Lightbend. *Akka Framework*. URL: `http://akka.io` (visited on 01/13/2019).

[ME13] C. Min and Y. I. Eom. "Danbi: dynamic scheduling of irregular stream programs for many-core systems." In: *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. Sept. 2013.

[ME15] C. Min and Y. I. Eom. "Dynamic scheduling of irregular stream programs toward many-core scalability." In: *IEEE Transactions on Parallel and Distributed Systems* 26.6 (June 2015).

[MHH11] Robert Massow, André van Hoorn, and Wilhelm Hasselbring. "Performance Simulation of Runtime Reconfigurable Component-Based Software Architectures." In: *Proceedings of the 5th European Conference on Software Architecture (ECSA '11)*. Ed. by Ivica Crnkovic, Volker Gruhn, and Matthias Book. Springer-Verlag, Sept. 2011.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and how to develop domain-specific languages." In: *ACM Comput. Surv.* 37.4 (Dec. 2005).

[MRH+09] N. Marwede, M. Rohr, A. v. Hoorn, and W. Hasselbring. "Automatic failure diagnosis support in distributed large-scale software systems based on timing behavior anomaly correlation." In: *13th European Conference on Software Maintenance and Reengineering*. Mar. 2009.

[MSM04] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. First. Addison-Wesley Professional, 2004.

[NAT+09] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval. "Analytical modeling of pipeline parallelism." In: *Proceedings of the 18th PACT*. Sept. 2009.

[Neo16] Neo4j, Inc. *Neo4j - a native graph database*. 2016. URL: https://neo4j.com/blog/why-database-query-language-matters/ (visited on 10/14/2016).

[Neu16] Mathis Neumann. "Extension of a domain-specific language for Pipe-and-Filter stage developers in teetime." Studienarbeit. Department of Computer Science, 2016.

[Ohl16] Johannes Ohlemacher. "Conception and development of a Pipe & Filter framework for C++." Master's thesis. Department of Computer Science, Nov. 2016.

[OPT09] Frank Otto, Victor Pankratius, and WalterF. Tichy. "XJava: Exploiting Parallelism with Object-Oriented Stream Programming". In: *Euro-Par 2009 Parallel Processing*. Ed. by Henk Sips, Dick Epema, and Hai-Xiang Lin. Vol. 5704. Springer Berlin Heidelberg, 2009.

[Oraa] Oracle, Inc. *Java's Streaming API*. URL: http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html (visited on 01/13/2019).

[Orab] Oracle, Inc. *JMH: A Java harness for building, running, and analysing microbenchmarks*. URL: http://openjdk.java.net/projects/code-tools/jmh (visited on 01/13/2019).

[Orac] Oracle, Inc. *The Java HotSpot Performance Engine Architecture*. URL: http://www.oracle.com/technetwork/java/whitepaper-135217.html#method (visited on 10/14/2016).

[Orad] Oracle Inc. *Thread Pools in Java*. URL: https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html.

[ORS+05] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. "Automatic thread extraction with decoupled software pipelining." In: *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2005.

[Ort10] Jorge Luis Ortega-Arjona. *Patterns for parallel software design*. 1st. Wiley Publishing, 2010.

[Par72] D. L. Parnas. "On the criteria to be used in decomposing systems into modules." In: *Commun. ACM* 15.12 (1972).

[PD10] Jongsoo Park and William J. Dally. "Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures." In: *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2010.

[Peg16] Adrian Pegler. "Evaluating approaches to detect bottlenecks in the Pipe & Filter framework TeeTime." Bachelor's thesis. Department of Computer Science, Mar. 2016.

[Piv] Inc. Pivotal Software. *RabbitMQ Library*. URL: https://www.rabbitmq.com (visited on 01/13/2019).

[RBW15] Raymond Roestenburg, Rob Bakker, and Rob Williams. *Akka in action*. 1st. Manning Publications Co., 2015.

[RMJ06] Derek Rayside, Lucy Mendel, and Daniel Jackson. "A dynamic analysis for revealing object ownership and sharing." In: *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*. ACM, 2006.

[SB99] R. van Solingen and E. Berghout. *The goal/question/metric method: a practical guide for quality improvement of software development*. McGraw-Hill, 1999.

[SBC+02] Rini van Solingen, Vic Basili, Gianluigi Caldiera, and H. Dieter Rombach. "Goal question metric (gqm) approach." In: *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., 2002.

[SBG+09] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. "Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures." In: *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*. ACM, 2009.

[SFB+09] Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. "Gramps: a programming model for graphics pipelines." In: *ACM Trans. Graph.* 28.1 (2009).

[SG96] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., 1996.

[Sha89] M. Shaw. "Larger scale systems require higher-level abstractions." In: *SIGSOFT Softw. Eng. Notes* 14.3 (1989).

[Sha96] Mary Shaw. "Procedure calls are the assembly language of software interconnection: connectors deserve first-class status." In: *Selected Papers from the Workshop on Studies of Software Design*. Springer-Verlag, 1996.

[SLY+11] D. Sanchez, D. Lo, R.M. Yoo, J. Sugerman, and C. Kozyrakis. "Dynamic fine-grain scheduling of pipeline parallelism." In: *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*. Oct. 2011.

[Sof] Esoteric Software. *Kryo Framework*. URL: `https://github.com/EsotericSoftware/kryo` (visited on 01/13/2019).

[Soo] Team of Soot. *Soot - a framework for analyzing and transforming java and android applications*. URL: `https://sable.github.io/soot` (visited on 10/16/2016).

[SPG+07] Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. "Streamflex: High-throughput Stream Programming in Java." In: *SIGPLAN Not.* 42.10 (Oct. 2007).

[Spi89] J. M. Spivey. *The z notation: a reference manual*. Prentice-Hall, Inc., 1989.

[SQK+10] M. Aater Suleman, Moinuddin K. Qureshi, Khubaib, and Yale N. Patt. "Feedback-directed pipeline parallelism." In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. 2010.

[SSH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. "Just model! – Putting automatic synthesis of node-link-diagrams into practice." In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*. Sept. 2013.

[Str16] Hannes Strubel. "Terminierung von Pipe & Filter Architekturen mit Feedback-Schleifen." Studienarbeit. Department of Computer Science, 2016.

[Str17] Hannes Strubel. "Monitoring distributed traces with Kieker." Master's thesis. Department of Computer Science, May 2017.

[SW02] Connie U. Smith and Lloyd G. Williams. "New software performance antipatterns: more ways to shoot yourself in the foot." In: *Proceedings of the CMG*. 2002.

[SW16] Hannes Strubel and Christian Wulf. "Refactoring Kieker's Monitoring Component to further Reduce the Runtime Overhead." In: *Proceedings of the Symposium on Software Performance*. Nov. 2016.

[Tav16] Nelson Tavares de Sousa. "Live execution time visualization of component-based software systems." Master's thesis. Department of Computer Science, Nov. 2016.

[TCA07] W. Thies, V. Chandrasekhar, and S. Amarasinghe. "A practical approach to exploiting coarse-grained pipeline parallelism in c programs." In: *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*. Dec. 2007.

[TF10] Georgios Tournavitis and Björn Franke. “Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information.” In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2010.

[TKA02] William Thies, Michal Karczmarek, and Saman Amarasinghe. “Streamit: a language for streaming applications”. In: *Compiler Construction*. Ed. by R.Nigel Horspool. Vol. 2304. Springer Berlin Heidelberg, 2002.

[TLH94] J. Torrellas, H. S. Lam, and J. L. Hennessy. “False sharing and spatial locality in multiprocessor caches.” In: *IEEE Transactions on Computers* 43.6 (June 1994).

[TMD09] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. “SEDA: An Architecture for Well-conditioned, Scalable Internet Services.” In: *SIGOPS Oper. Syst. Rev.* 35.5 (Oct. 2001).

[WEH14] Christian Wulf, Nils Christian Ehmke, and Wilhelm Hasselbring. “Toward a Generic and Concurrency-Aware Pipes & Filters Framework.” In: *Proceedings of the Symposium on Software Performance 2014: Joint Descartes/Kieker/Palladio Days*. Nov. 2014.

[WH12] Jan Waller and Wilhelm Hasselbring. “A comparison of the influence of different multi-core processors on the runtime overhead for application-level monitoring.” In: *Proceedings of the MSEPT)*. Ed. by Victor Pankratius and Michael Philippsen. Vol. 7303. Springer Berlin / Heidelberg, June 2012.

[WH15] Christian Wulf and Wilhelm Hasselbring. “Software Performance Anti-Patterns Observed and Resolved in Kieker.” In: *Symposium on Software Performance 2015: Joint Developer and Community Meeting of Descartes/Kieker/Palladio*. Vol. 35. 5. Softwaretechnik-Trends, Nov. 2015.

[WH17] Christian Wulf and Wilhelm Hasselbring. “Increasing the throughput of pipe-and-filter architectures by integrating the task farm parallelization pattern.” In: *Proceedings of the Software Engineering (SE'17)*. Vol. P-267. Feb. 2017.

[WHO17] Christian Wulf, Wilhelm Hasselbring, and Johannes Ohlemacher. “Parallel and generic pipe-and-filter architectures with TeeTime.” In: *Proceedings of the International Conference on Software Architecture (ICSA)*. Apr. 2017.

[Wie15] Christian Claus Wiechmann. “On improving the performance of Pipe-and-Filter architectures by adding support for self-adaptive task farms.” Master's thesis. Department of Computer Science, Oct. 2015.

[WRH+12] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in software engineering*. Springer Publishing Company, Incorporated, 2012.

[Wul14] Christian Wulf. “Pattern-based Detection and Utilization of Potential Parallelism in Software Systems.” In: *Proceedings of the Software Enginnering*. 2014.

[Wul15] Christian Wulf. *Java-based reference implementation of the TeeTime framework*. 2015. URL: http://teetime.sourceforge.net.

[WWF+13] Jan Waller, Christian Wulf, Florian Fittkau, Philipp Döhring, and Wilhelm Hasselbring. “SynchroVis: 3D Visualization of Monitoring Traces in the City Metaphor for Analyzing Concurrency.” In: *1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. Sept. 2013.

[WWH16] Christian Wulf, Christian Claus Wiechmann, and Wilhelm Hasselbring. “Increasing the Throughput of Pipe-and-Filter Architectures by Integrating the Task Farm Parallelization Pattern.” In: *Proceedings of the International Symposium on Component Based Software Engineering (CBSE)*. 2016.

[Wya13] Derek Wyatt. *Akka Concurrency*. Artima Incorporation, 2013.

[WZT+13] Junchang Wang, Kai Zhang, Xinan Tang, and Bei Hua. "B-queue: efficient and practical queuing for fast core-to-core communication." In: *International Journal of Parallel Programming* 41.1 (2013).

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing." In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2012.

[Zlo15] Martin Zloch. "Development of a domain-specific language for Pipe-and-Filter stage developers." Studienarbeit. Department of Computer Science, 2015.

[Zlo16] Martin Zloch. "Development of a domain-specific language for Pipe-and-Filter configuration builders." Studienarbeit. Department of Computer Science, 2016.

[ZXW+16] Matei Zaharia et al. "Apache spark: a unified engine for big data processing." In: *Commun. ACM* 59.11 (Oct. 2016).