Toward a Generic and Concurrency-Aware Pipes & Filters Framework

Christian Wulf, Nils Christian Ehmke, and Wilhelm Hasselbring

Software Engineering Group
Kiel University
24098 Kiel, Germany
{chw,nie,wha}@informatik.uni-kiel.de

Abstract: The Pipes-and-Filters design pattern is a well-known pattern to organize and execute components with sequential dependencies. The pattern is therefore often used to perform several tasks consecutively on large data streams, e.g., during image processing or dynamic analyses. In contrast to the pattern's familiarity and application, almost each common programming language lacks of flexible, feature-rich, fast, and concurrency-aware Pipes-and-Filters frameworks. So far, it is common practice that most developers write their own implementation tailored to their specific use cases and demands hampering any effective re-use.

In this paper, we discuss Pipes-and-Filters architectures of several Java-based applications and point out their drawbacks concerning their applicability and efficiency. Moreover, we propose a generic and concurrency-aware Pipes-and-Filters framework and provide a reference implementation for Java called TeeTime.

1 Introduction

Pipes-and-Filters is a common architectural pattern in several projects and applications, often used to process large data streams. Figure 1 shows an example Pipes-and-Filters-oriented processing to visualize program traces, which are reconstructed from serialized method events. Each filter component, also called *stage*, fetches incoming elements from its input ports, processes them, and finally sends the resulting elements via its output ports.

During the development of Kieker [RvHM+08, vHRH+09, vHWH12], an application performance monitoring and architecture discovery framework, we encounter various archi-

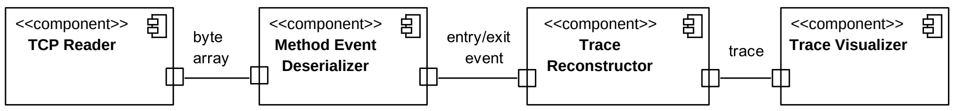

Figure 1: An example Pipes-and-Filters-oriented processing to visualize program traces reconstructed from serialized method events. Ports serve as interface to connect two stages with each other.

tecture and performance issues with Kieker's Pipes-and-Filters-oriented analysis component. Feedback from users confirm these issues. Although suitable for most post-mortem analyses, the framework is not capable of processing the amount of data required for more complex analyses, such as live architecture reconstruction and performance anomaly detection. The underlying analysis implementation does not take advantage of multi processor systems and relies extensively on slow reflection calls and string comparison. Moreover, Kieker's analysis component does not support the composition of multiple stages.

There are only very few Pipes-and-Filters frameworks that are not tailored to a particular use case, but designed for arbitrary pipeline architectures. To the best of our knowledge, none of them are easy to use, extensible, efficient, and address both the usage and the abstraction of multi-core architectures in one single framework.

Hence, we decided to develop a generic and concurrency-aware Pipes-and-Filters framework using our experience and our requirements from Kieker. This includes a fast reference implementation called TeeTime, ports for stages, and a convenient application programming interface which abstracts from the management of concurrency.

Our framework is not limited to Kieker and thus can be used in other projects as well. Furthermore, it can be applied to various programming languages, although TeeTime is written in the Java programming language. A manual and the source code of TeeTime are publicly available at Sourceforge[1].

Hence, our contributions are:

- An approach for a generic and concurrency-aware Pipes-and-Filters framework, and
- a reference implementation for the Java programming language.

The rest of this document is structured as followed. First, we present related work and existing frameworks in Section 2. In the subsequent two sections, we describe our approach focusing on its architecture in Section 3 and its concurrency handling in Section 4. Section 5 finally contains our conclusion and future work.

2 Related Work

Due to the fact that Pipes-and-Filters is a frequently used pattern, various generic solutions exist already. We present and discuss an excerpt of existing frameworks focusing on Java-based implementations. Furthermore, we list work regarding strategies for the concurrent execution of stages.

[1] `https://sourceforge.net/projects/teetime/`

2.1 Current Java-based Pipes-and-Filters Frameworks

A generic Pipes-and-Filters framework is Apache Commons Pipeline[2]. This framework can take advantage of additional threads, but assumes that the stages are implemented in a thread-safe manner. Furthermore, the project has no released version and is not developed any further since 2009.

Apache Camel[3] is a Java framework to configure routing and mediation rules using the enterprise integration patterns. It can also be used to assemble Pipes-and-Filters 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[4].

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 Pipes-and-Filters oriented style, it requires additional knowledge to configure and execute a pipeline configuration that goes beyond the Pipes-and-Filters pattern. Furthermore, its use of a scheduler and coarse-grained synchronization mechanisms results in an additional run-time overhead.

The Kieker Monitoring and Dynamic Analysis Framework[5] provides a Pipes-and-Filters-API to create analysis networks. Due to its drawbacks mentioned in Section 1, we will replace it by TeeTime's API.

ExplorViz [FWBH13] is a tool that enables live trace visualization for system and program comprehension in large software landscapes. It comprises a Pipes-and-Filters-based component that is tailored to the processing of program traces. Hence, it is not suited as a generic Pipes-and-Filters framework.

XML Calabash[6] is an implementation of the XML pipeline language XProc. [7] This language can be used to describe pipelines consisting of atomic or compounded operations on XML documents.

Pipes[8] is a Java-based framework using process graphs. So called pipes represent the atomic computing steps and form, once connected, the processing graph. As the pipes implements interfaces with generics, they are type-safe, but have to be arranged in sequential order.

Java 8 introduced a new streams API 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.

Akka[9] 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

[2]http://commons.apache.org/sandbox/commons-pipeline/
[3]https://camel.apache.org/
[4]see http://camel.apache.org/parallel-processing-and-ordering.html
[5]http://kieker-monitoring.net/
[6]http://xmlcalabash.com/
[7]http://www.w3.org/TR/xproc/
[8]https://github.com/tinkerpop/pipes/wiki
[9]http://doc.akka.io

it is possible to map Akka's actor-based API to a Pipes-and-Filters-based one, Akka is not optimized for the execution of pipeline architectures.

2.2 Concurrent Execution of Stages

There are at least two common strategies to execute a pipeline configuration concurrently [SLY+11, SQKP10].

The first strategy (S1) distributes the given threads over a distinct subset of the declared stages, e.g., each thread executes a single stage. 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 are usually more efficient for stable and predictable pipeline configurations, dynamic assignment approaches 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 require pipes to be synchronized, but rather stateful stages since the copies of a stage usually share one single state.

Some approaches [SLY+11, NATC09] use work-stealing pipes to balance the workload across all available threads. Such pipes then require single-producer-multiple-consumers data structures that cause additional run-time overhead due to further synchronization and management effort.

3 Toward the TeeTime Framework

Our framework targets software engineers that need to process data in a stream-oriented, throughput-optimized, and type-safe fashion. Although our reference implementation is written in Java, our framework's software architecture can be easily adapted to other object-oriented programming languages that are aware of threads and type parameters, e.g., C#.

We base our framework on the Tee-and-Join-Pipeline design pattern [BMR+96], a generalized version of the Pipes-and-Filters design pattern. It allows a stage to be connected by more than one input and output pipe. For this reason, we call our reference implementation TeeTime.

After giving an overview of our framework's architecture, we describe the most important features in more detail.

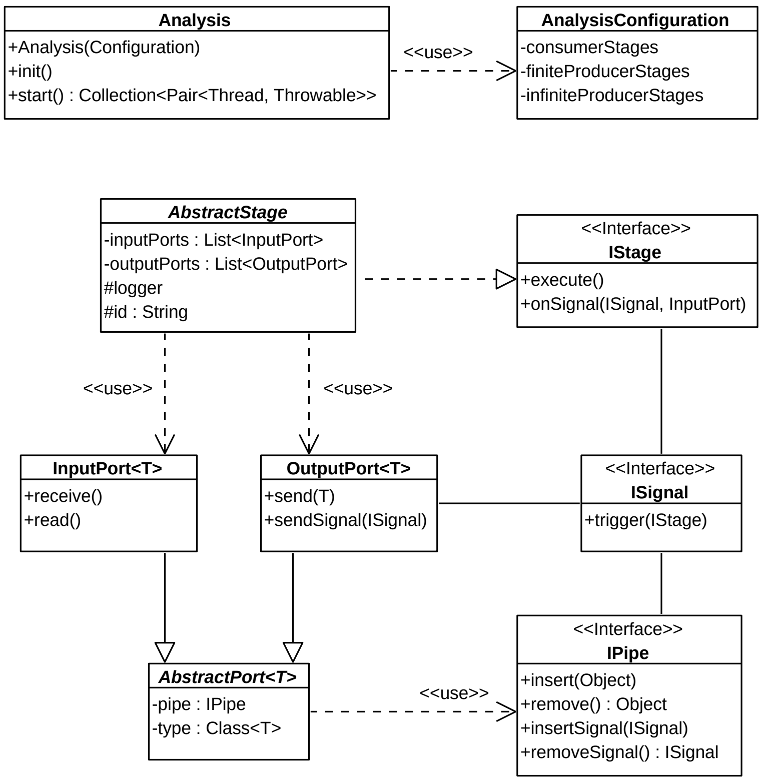

Figure 2: Overview of the basic entities in our Pipes-and-Filters framework architecture

3.1 Overview of the Software Architecture

The core of our framework consists of four basic entities: `Stages`, `Ports`, `Pipes`, and `AnalysisConfigurations`.

A stage represents a processing unit within a pipeline that takes elements from its *input ports*, uses them to compute a result, and puts this result to its *output ports*. Therefore, the concept of ports allows a stage to communicate with multiple predecessors and successors. Additionally, a stage is assigned a unique identifier and a logger to ease in debugging.

A pipe connects an output port of one stage with an input port of another stage by means of a queue. However, a pipe can also be used to realize self-loops by connecting an output port with an input port of the same stage.

An analysis configuration represents a concrete pipeline setup. It defines the stages that should be used and connects their ports with each other.

In Figure 2 we show an overview of the basic entities in our Pipes-and-Filters framework architecture.

3.2 Stage Scheduling

Our framework does not make use of an explicit stage scheduler to decide what stage should be executed next. Instead the pipes between the stages undertake this task.

Each time a stage sends an element to one of its output port, the corresponding connected pipe stores the element in its internal queue. Depending on the pipe's implementation, it then either triggers the execution of the receiver stage or it immediately returns to the sender stage.

The former implementation is used to realize intra-thread communication, i.e., to connect two stages that are executed by the same thread. It ensures progress and reduces the memory usage since passing an element through the whole pipeline has the highest priority. This strategy is also known as the *backpressure* technique.

The latter implementation is used to realize inter-thread communication, i.e., to connect two stages that are executed by two different threads. It causes only a minimal delay in the execution of the sending stage.

In this way, we gain the highest possible throughput with respect to the framework's flexible architecture. For example, we require a throughput of ten thousands to millions of elements per second when analyzing monitoring logs with Kieker. In such situations, an explicit scheduler causes too much run-time overhead because it needs to switch too often between all stages.

The avoidance of a dedicated scheduler additionally reduces the overall complexity of the framework itself and allows the (JIT-)compiler to perform further optimizations, e.g., utilizing cache locality due to back-pressuring [SLY+11].

3.3 Type-safe Connection of Stages

We extend our framework with typed ports to provide a fully type-safe environment. Therefore, it is impossible to connect two incompatible ports which dramatically reduces the need for debugging.

The framework checks for type-safetiness at compile-time and at run-time. The former is realized by specifying a type parameter for each port. In this way, the compiler is able to verify that a pipe connects two ports of the same type.

The latter is realized by checking whether each two ports have the same type attribute (cp. Figure 2) just before a pipeline is being executed. It is not possible to check the type parameter at run-time because some object-oriented programming languages, such as Java, perform type-erasure on the type parameters.

3.4 Reacting to Signals

We further extend our framework by a generic signal concept. A signal represents an event that is triggered within one stage and passed to all other stages with respect to the given pipeline architecture.

Similar to transmitting an element, a signal is sent from an output port to an input port using a pipe. For this purpose, both the intra-thread and the inter-thread pipe implementation already mentioned above have another internal queue that is used for signals only.

For example, a `StartingSignal` is triggered and processed by the first stage when the execution of a pipeline begins. Afterwards the signal is passed to all successor stages of the first stage where the same procedure is repeated until the last stage has processed the signal.

Besides the `StartingSignal`, our framework implementation currently provides a `ValidatingSignal` that occurs when the connections of a stage are being validated and a `TerminatingSignal` that occurs when the execution of a pipeline is being terminated.

Moreover, our framework allows to add further signal types. A new signal type must conform to the `ISignal` interface and needs to be triggered once. The correct signal passing is then handled by our framework automatically.

3.5 Composition of Stages

Stage composition is a key concept that allows to build on top of already available stages. It effectively hides complexity and also improves the usability. For this reason, our framework offers direct support for composing stages.

3.6 Basic Stages

While developing and using Kieker [RvHM$^+$08, vHRH$^+$09, vHWH12], we have identified various atomic stages that are often used either directly or indirectly to build more complex, high-level stages. Below, we describe some of the most important ones in more detail.

Distributor The distributor is characterized by one input port and a customizable number of output ports. It takes an element from its input port and distributes it to its output ports. The distributor uses the `DistributionStrategy` interface to decide how the input element should be distributed among the output ports. We currently provide implementations that forwards the input element either to one output port according to the round-robin style, or to each output port. Instead of simply forwarding the element, a solution would be conceivable where the distributor delivers a deep clone of the input element.

Merger The merger is characterized by a customizable number of input ports and a single output port. It takes elements from a subset of its input ports and merges them to its output port. Similar to the distributor, the merge step is done according to a particular implementation of the `MergeStrategy` interface. We currently provide a round-robin implementation that forwards the element of the next non-empty input port to the output port.

File system stages The directory reader outputs all available files within a given directory. The text line file reader reads in a text file from its input port and successively sends each text line via its output port. The file writer takes a byte array from its input port and writes it to a pre-configured file.

Element type independent stages Besides the stages mentioned above, there are also stages that process elements independent of the their concrete types. The repeater stage takes one element from its input port and outputs a multiple of it to its output port. The delay stage delays the passing of an element to its successor stage by a customizable amount of time. The throughput stage counts the number of passing elements and outputs the corresponding throughput if it is triggered by another stage to do so. The InstanceOfFilter stage checks whether an incoming element matches a configurable particular type and passes it to one of its two output ports representing a valid or invalid match.

Further stages We continuously extend the set of provided stages. Besides the basic ones mentioned above, we currently offer stages that, e.g., count the words contained in a string, encrypts and decrypts a byte array, and outputs the current time in a configurable regular interval.

We also provide stages that interact with I/O devices, such as the file system and the network (via TCP, JMX, JMS, and JDBC). In this way, it is also possible to build a pipeline distributed over several computing nodes.

3.7 Example stage

We now consider the TeeTime implementation of the directory reader described above (see Listing 1) to show (1) that the API only requires few additional knowledge beyond the Pipes-and-Filters pattern itself and (2) that it is able to abstract from any concurrent data structure or directive.

The stage extends the `ConsumerStage` that, among others, provides methods for the declaration of ports (see line 3) and encapsulates the handling of the first input port by means of the parametrized `execute` method (see line 11). If a stage requires more than one input port, it simply declares it and checks the ports on available input.

Line 5 and 6 contains the declarations of a file filter and a file comparator, respectively. Both are used in the `execute` method to list only relevant files (see line 12) in the desired order (see line 20). Finally, the read files are successively send via the declared output port (see lines 23-25).

Hence, the source code only contains those statements that are necessary to declare a directory reader. In particular, it does not make use of any concurrent data structure, e.g., a concurrent blocking queue, and any concurrent directive, such as `volatile` or `synchronized` to interact with its predecessor or successor stage.

Listing 1: The TeeTime implementation of the directory reader

```
 1  public class Directory2FilesFilter extends ConsumerStage<File> {
 2
 3    private final OutputPort<File> outputPort = this.createOutputPort();
 4
 5    private FileFilter filter;
 6    private Comparator<File> fileComparator;
 7
 8    // omitted costructors
 9
10    @Override
11    protected void execute(final File inputDir) {
12      final File[] inputFiles = inputDir.listFiles(this.filter);
13
14      if (inputFiles == null) {
15        this.logger.error("Directory␣'" + inputDir + "'␣does␣not␣exist␣or␣
            an␣I/O␣error␣occured.");
16        return;
17      }
18
19      if (this.fileComparator != null) {
20        Arrays.sort(inputFiles, this.fileComparator);
21      }
22
23      for (final File file : inputFiles) {
24        this.send(this.outputPort, file);
25      }
26    }
27
28    // omitted getter and setter
29
30    public OutputPort<File> getOutputPort() {
31      return outputPort;
32    }
33
34  }
```

4 Concurrency Handling

To the best of our knowledge there is no similar Pipes-and-Filters framework that manages and handles a concurrent execution in such a transparent and efficient way. In the following, we describe how our approach concurrently executes the stages, how it handles synchronization issues, and how it does everything in a highly efficient and automatic way.

4.1 Concurrent Execution of Stages

Since there are several different use cases for using a Pipes-and-Filters-oriented architecture, we do not commit ourselves to one of the two strategies mentioned in Section 2. Instead, we recommend a hybrid approach that makes use of both approaches.

Consider the sample pipeline consisting of a producer and a consumer whereby elements are produced faster than consumed. In this scenario, duplicating and executing the producer in more than one thread does not increase the throughput since the consumer is the bottleneck. Furthermore, if the producer reads from an I/O device, instantiating more than one producer is often not advisable since most I/O devices can be accessed only in a sequential way excluding any concurrent access. However, duplicating and executing the consumer stage increases the performance provided the internal state can be shared efficiently. Hence, a combination of S1 and S2 can perform better than each of the strategies in isolation.

4.2 Automatic Thread Instantiation and Management

To enable concurrent execution of stages, a number of threads and a suitable management handling their life-cycle is required. Instead of manually instantiating and managing these threads for each individual pipeline, we propose a more abstract concept that encapsulates such technical issues by our framework.

For this purpose, each stage is declared either as consumer or producer where a producer is further divided into finite or infinite.

A **consumer** changes its internal state and/or produces one or more output elements if and only if it receives one or more input elements. Thus, a consumer stage provides at least one input port. It terminates when it receives a termination signal.

A **producer** changes its internal state and/or produces one or more output elements according to its semantics. It provides no input ports and is responsible for its termination.

A producer is **finite** if it terminates itself after a finite number of executions. A producer is **infinite** if it does not terminate autonomously. In this case, the framework terminates the producer stage after all finite producers are terminated.

Based on this categorization, the framework is able to determine the amount of threads necessary for the execution and the assignment of the stages to the threads. For instance, since a producer is independent of other stages, it can be executed concurrently. However, for the sake of fine-grained optimizations, it is also possible to manually assign stages to threads.

4.3 Synchronization of Stages

4.3.1 Synchronization by Pipe

In our framework, a pipe is responsible for the transmission of elements between two stages.

When connecting two stages within the same thread, the framework chooses an unsynchronized pipe implementation holding a single element. Once a stage sends an element, the corresponding pipe stores it and executes the receiver stage. The receiver stage finally pulls the element from the pipe.

When connecting two stages executed within two different threads, the framework chooses a pipe implementation consisting of a synchronized queue. Once a stage sends an element, the corresponding pipe adds it in a non-blocking fashion to the queue. The receiver stage can either use a busy-waiting or a blocking strategy in order to pull the element from the pipe.

The synchronized queue is a highly optimized implementation [10] for the single-producer/single-consumer scenario. It is lock-free, cache-aware, and uses only a few well-placed load/load and store/store barriers for the synchronization. In particular, it does not make use of any types of the Java Concurrency API and of any volatile declaration. We have also tested a pipe implementation that bases on the work-stealing paradigm, however it performs significantly slower due to the heavy use of volatile fields.

4.3.2 Synchronization by Shared State

We call a stage stateful if it contains attributes and uses them to compute its outputs. Otherwise we call a stage stateless. It is recommended to implement stages in a stateless way, as this avoids synchronization overhead and issues between stages. However, in the Kieker project we identified various scenarios relying on stateful stages.

For instance, consider an aggregation stage that outputs a result not until a particular number of executions. Before reaching such a threshold, this stage needs to memorize the result of the current and all previous executions.

Our framework does not yet support in synchronizing a shared state although we plan to abstract from particular scenarios (see Section 5). Currently, a developer must be aware of concurrency issues that could arise due to a shared state and has to ensure a proper synchronization.

[10]`https://github.com/JCTools/JCTools/`

5 Conclusions and Outlook

In this paper, we indicated the drawbacks of current Pipes-and-Filters frameworks and propose a more flexible and concurrency-aware Pipes-and-Filters framework architecture. It focuses on a type-safe application programming interface that makes complex decisions for the programmer and abstracts from technical issues to avoid building an incorrect and inefficient pipeline. We also provide a fast Java-based reference implementation called TeeTime.

We constantly improve our framework by reducing the management overhead, by adding more functionality, and by simplifying the API. Our next step is to give the responsibility of choosing and instantiating a pipe completely to the framework. Afterwards, we plan to automatize the thread assignment as much as possible so that ideally the programmer only needs to describe the pipeline structure and to specify the number of processor cores to use. For this purpose, we follow the idea to utilize further stage information available at compile-time, such as the ratio of the input and output ports or the properties of a stage's state. Moreover, we work on an automatic exception handling by the framework and a way to save and load arbitrary pipeline configurations that were created at run-time.

References

[BL10] Christopher Brooks and Edward A. Lee. Ptolemy II - Heterogeneous Concurrent Modeling and Design in Java, February 2010. Poster presented at the 2010 Berkeley EECS Annual Research Symposium (BEARS).

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1. edition, 1996.

[FWBH13] 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*, volume 1083, pages 89–98. CEUR Workshop Proceedings, November 2013.

[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*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[NATC09] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval. Analytical Modeling of Pipeline Parallelism. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 281–290, September 2009.

[RvHM+08] Matthias Rohr, André van Hoorn, Jasminka Matevska, Nils Sommer, Lena Stoever, Simon Giesecke, and Wilhelm Hasselbring. Kieker: Continuous Monitoring and on demand Visualization of Java Software Behavior. In Claus Pahl, editor, *Proceedings of the IASTED International Conference on Software Engineering 2008 (SE'08)*, pages 80–85, Anaheim, CA, USA, Februar 2008. ACTA Press.

[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 Con-*

ference on Parallel Architectures and Compilation Techniques (PACT), pages 22–32, October 2011.

[SQKP10] 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 (PACT)*, PACT '10, pages 147–156, New York, NY, USA, 2010. ACM.

[vHRH^{+}09] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Forschungsbericht, Kiel University, November 2009.

[vHWH12] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM, April 2012.