Parallel and Generic Pipe-and-Filter Architectures with TeeTime

Christian Wulf
Software Engineering Group
Kiel University
24098 Kiel, Germany
Email: chw@informatik.uni-kiel.de

Wilhelm Hasselbring
Software Engineering Group
Kiel University
24098 Kiel, Germany
Email: wha@informatik.uni-kiel.de

Johannes Ohlemacher
Software Engineering Group
Kiel University
24098 Kiel, Germany
Email: johl@informatik.uni-kiel.de

***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 paper, we present our 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. Besides a description of its major features, we also present an application example in Java.

I. Introduction

Pipe-and-Filter (P&F) is a well-known and often used architectural style in industry and research since it potentially combines a high modularity with a high throughput and a small memory footprint. Recent research [2, 9, 15] shows that the P&F style still provides a high potential for optimization, especially in the context of parallelization and the execution on heterogeneous platforms. The P&F style may be applied in various application domains. For instance, with Kieker [10] and ExplorViz [6], we use the P&F style for dynamic analysis and for live processing of high-volume monitored traces.

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 requirements inhibiting effective re-use. Furthermore, concurrency is often handled, if at all, only at a coarse-grained level, neglecting parallelization potential of multi-core processor systems. This situation results from the lack of easy-to-use generic P&F frameworks which are able to cover all kinds of P&F architectures and simultaneously provide a high performance with a low overhead caused by the framework implementation.

While there are some frameworks, such as Fastflow [2], StreamIt [9], GRAMSP [15], Spark [13], Storm [14], and Akka [1], 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. On the other hand, GRAMPS is tailored to graphics pipelines.

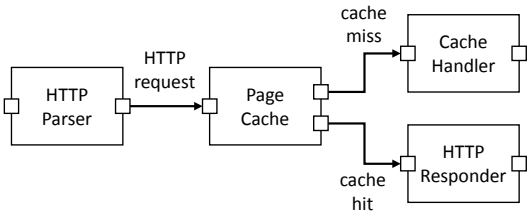
In this paper, we discuss these open challenges in more detail and present our P&F framework *TeeTime* as a solution. To the best of our knowledge, it is the first framework that is able to model and to execute arbitrary P&F architectures. Moreover, it allows to declare filters to be executed in parallel at a new level of abstraction. For example, TeeTime utilizes the notion of active and passive filters [4] to manage threads and their synchronization in a fully transparent way.

In our previous work [5, 21], we have successfully applied TeeTime to build P&F architectures with multiple ports per filter and with high throughput rates. In this work, we describe how TeeTime achieves this genericity and performance. Additionally, we present a small P&F example in Java to provide a more detailed view on the design and the usage of TeeTime.

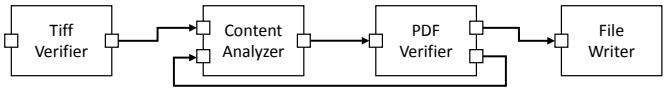
We categorize TeeTime as research tool since it allows to experiment with the P&F architectural style. Its extensible framework architecture enables to implement arbitrary (composite) filters, various synchronized and unsynchronized pipes, as well as parallel and distributed execution models. However, TeeTime also targets software architects and software developers who just want to use it within their applications. We provide a Java-based [17] and a C++-based [16] reference implementation of our framework as open-source software.

II. Open Challenges for P&F Frameworks

As mentioned in Section I, we are not aware of a framework which supports the modeling and the execution of arbitrary P&F architectures. Such a framework must at least support the dataflow structures shown in Figure 1. However, most available frameworks focus on streaming applications which allow at most one input and one output port. Hence, they cannot model branches (see Figure 1a). For the same reason, they often do not support loops (see Figure 1b). Although some frameworks provide support for loops, they only do so for particular filters. In contrast, an actor in actor-based frameworks can output to multiple receivers. However, it still has only one input port—often called mailbox. In addition, this mailbox is also untyped



(a) 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. [20]



(b) An example feedback loop: the Document Understanding and Analysis System of Gokhale et al. [8]

Fig. 1: Variations of P&F dataflow structures

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.

The P&F style can also provide a basis for parallelization. For example, multiple filters can run in parallel to increase the throughput. Allen et al. [3] 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. Moreover, 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. Research [9, 15] shows that an efficient parallel execution of P&F architectures is still work-in-progress and requires further research.

Hence, 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. GRAMPS [15], e.g., does not provide this level of flexibility.

III. The P&F Framework TeeTime

In this section, we present our P&F framework *TeeTime*. Its key features are (1) its support for all P&F variants described in Section II, (2) its parallel execution model with a high level of abstraction, and (3) its extensibility to experiment with the P&F style. The design of the framework architecture as well as our Java [17] and C++ [16] implementations follow the original definition by Shaw [12] and Allen et al. [3]. A P&F-based system consists of the four first-class entities: *pipes*, *filters*, *ports*, and *configurations*. In this way, we implement the scheduling and the synchronization within pipes, the execution logic within filters, and the type-safety within ports.

In the following Sections III-A to III-E, we describe how TeeTime addresses the challenges mentioned in Section II. We will use the term *stage* as generalization for *data sources*, *filters*, and *data sinks*, as categorized by Buschmann et al. [4].

A. Automatic Thread Management

Buschmann et al. [4] distinguish three scenarios to trigger the activity of a stage. First, the stage's successor pulls output data from the stage. Second, the stage's predecessor pushes new input data to the stage. Finally, the stage is in an active loop pulling its input from its predecessors and pushing its output to its successors. Buschmann et al. call the first two cases *passive* stages and the last one an *active* stage.

Instead of manually creating a thread for each stage that should be active, TeeTime only requires to declare a stage as *active*. Based on this information, the framework automatically associates one dedicated thread with each active stage and its passive successor stages. Thus, TeeTime completely manages the instantiation, execution, and termination of the threads.

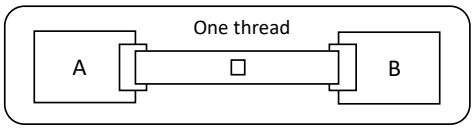
B. Efficient Pipe-based Communication

Since a distributed system typically communicates via a network which transmits data as byte sequences, data conversion and synchronization must always be performed. However, in a shared-memory system, data is usually exchanged via pointers without any data conversion. Moreover, data synchronization is only necessary between stages that are executed by two different threads. Hence, TeeTime connects stages as follows.

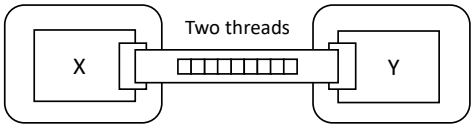
When connecting two stages within the same thread, the framework chooses an unsynchronized pipe holding a single element (see Figure 2a). Once a stage sends an element, the corresponding pipe stores it and executes the consumer stage which in turn pulls the element from the pipe. In this way, we do not only avoid an unnecessary synchronization between these stages. We also ensure back-pressure, because each data element is pushed as far as possible through the whole P&F architecture before processing the next data element. This approach allows direct method calls while simultaneously preserving stage recombination since it encapsulates the scheduling of stages within the pipes.

When connecting two stages executed within two different threads, the framework chooses a pipe that employs a synchronized queue (see Figure 2b). Once a stage sends an element, the corresponding pipe adds it to the queue either in a non-blocking or blocking mode depending on the chosen push strategy. Similarly, the consumer stage can choose between a busy-waiting or a blocking strategy to consume the element.

For achieving a high throughput and a high scalability, it is important that this synchronized queue has a low and constant overhead. We use the lock-free `SpScArrayQueue` of the JCTools library [11]. This queue is similar to the queues described by [7, 19]. It features a low overhead by using only light-weight synchronization mechanisms, e.g., memory barriers. 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 intra-producer and intra-consumer contention.



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



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

Fig. 2: Efficient pipe-based communication in TeeTime

C. Automatic Pipe Selection

TeeTime employs a two-phase approach to automatically identify the correct and most efficient pipe implementation between two ports. In the first phase, when connecting ports in the configuration, a placeholder pipe is instantiated. In the second phase, after all stages have been declared passive or active, the whole P&F architecture is traversed. Each incoming and outgoing pipe that is a placeholder, is replaced by a synchronized or unsynchronized pipe depending on whether the consuming stage is active or not. In this way, we can also automatically connect stages that are created at runtime.

D. Efficient Multi-Port Composite Stages

We distinguish between primitive and composite stages. A composite stage is only a wrapper and does not exist at runtime. It is flattened to the P&F configuration represented by its child stages. Thus, it does not have a dedicated execution logic and cannot be declared active. In this way, we allow modularization and, at the same time, ensure that a composite stage does not incur *any* runtime overhead.

E. Type Safety

TeeTime checks for type safety between interconnected ports at compile-time and at run-time: At compile-time, it employs type parameters for ports. If two ports are incompatible with each other, the compiler detects this issue and outputs an appropriate error message. At run-time, TeeTime employs a `type` attribute for each port. On initialization, TeeTime checks each connection whether its source and target port match. Run-time validation is necessary when type parameters are not available, e.g., when loading a stage via reflection.

IV. TeeTime Application Example

As mentioned in the introduction (Section I), we provide two reference implementations for TeeTime: one for Java and one for C++. Although they differ in some technical details, both base on the same, general, language-independent framework architecture introduced in Section III. Hence, for a consistent presentation, we use the same programming language in all following code listings, namely Java.

Listing 1: An example configuration in TeeTime

```
1  public class LogReaderConfig extends Configuration {
2    final CollectorSink<Record> sink;
3
4    public LogReaderConfig(List<Path> dirs) {
5      InitialElementProducer<Path> producer = new
           ↪ InitialElementProducer<>(dirs);
6      KiekerLogDirSwitch kiekerLogDirSwitch = new
           ↪ KiekerLogDirSwitch();
7      AsciiLogDirReader asciiDirReader = new
           ↪ AsciiLogDirReader();
8      BinaryLogDirReader binaryDirReader = new
           ↪ BinaryLogDirReader();
9      Merger<Record> merger = new Merger<>();
10     this.sink = new CollectorSink<>();
11
12     connectPorts(producer.getOutputPort(),
           ↪ kiekerLogDirSwitch.getInputPort());
13     connectPorts(kiekerLogDirSwitch.getAsciiPort(),
           ↪ asciiDirReader.getInputPort());
14     connectPorts(kiekerLogDirSwitch.getBinaryPort(),
           ↪ binaryDirReader.getInputPort());
15     connectPorts(asciiDirReader.getOutputPort(),
           ↪ merger.getNewInputPort());
16     connectPorts(binaryDirReader.getOutputPort(),
           ↪ merger.getNewInputPort());
17     connectPorts(merger.getOutputPort(),
           ↪ this.sink.getInputPort(),
           ↪ BufferedUnsynchronizedPipeFactory.INSTANCE);
18
19     producer.declareActive();
20     asciiDirReader.declareActive();
21     binaryDirReader.declareActive();
22     merger.declareActive();                    }}
```

We consider a P&F architecture (simplified for the sake of clarity) taken from the Kieker project [18] which processes directories containing monitoring log files either in an ASCII or a binary format. Listing 1 shows the corresponding configuration in Java. Line 1, we declare the `LogReaderConfig` by extending the class `Configuration` provided by TeeTime. We declare the stages in Line 5-10 and their interconnections in Line 12-17. Active stages are declared in Line 19-22.

First, an initial list of directories is passed to the stage `InitialElementProducer` (Line 5) which outputs each directory one by one to its output port (Line 12). Then, the stage `KiekerLogDirSwitch` (Line 6) checks the incoming directory format. Afterwards, it passes the directory (Lines 13/14) to the stage `AsciiLogDirReader` (Line 7) or, respectively, to the stage `BinaryLogDirReader` (Line 8). If the directory contains unknown files, it is not passed and thus filtered out. Subsequently, the triggered log reader reconstructs and outputs an instance of the class `Record` for each log entry in each log file. Such a record is then passed (Lines 15/16) through the stage `Merger` (Line 9) to the stage `CollectorSink` (Lines 10/17). By passing a pipe factory, TeeTime 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.

Listing 2 shows the execution of the configuration. We pass two example directories (Lines 1–4) to a new instance of `LogReaderConfig` (Line 5). We then execute the configuration by passing it to a new instance of TeeTime's `Execution` class (Line 6) and by invoking the method `executeNonBlocking()` (Line 7). Finally, we wait for its termination (Line 8) in order to receive the reconstructed records from the `CollectorSink`.

Listing 2: An example execution of the `LogReaderConfig`

```
1 Path[] dirs = { Paths.get("a/b/c-ascii"),
      ↪ Paths.get("x/y/z-binary") };
2 LogReaderConfig config = new LogReaderConfig(dirs);
3 Execution execution = new Execution(config);
4 execution.executeNonBlocking();
5 ...
6 execution.waitForTermination();
7 records = config.sink.getElements();
```

Listing 3 shows the implementation of the `KiekerLogDirSwitch`. The stage extends the class `AbstractStage` provided by TeeTime (Line 1) and declares one input port (Line 2) and three output ports (Lines 3–5). All ports have the type `Path` which represents a directory from the file system. The stage's behavior is implemented by the method `execute()` (Lines 7–17) which is invoked by TeeTime. First, it reads an element from its input port (Line 8). If the element is `null` (Line 9), the stage returns and TeeTime reschedules it accordingly. Otherwise, the stage checks the format of the directory element and passes it to the `asciiPort`, `binPort`, or `elsePort`, respectively. In Lines 19–21, it exports its input port via the public method `getInputPort()` which in turn is used in Listing 1 to connect the stage with the `producer`. The type parameter of the returning input port ensures the type-safety mentioned in Section III-E.

Listing 3: An example stage used by the `LogReaderConfig`

```
1  public class KiekerLogDirSwitch extends AbstractStage {
2    InputPort<Path> inputPort = super.createInputPort();
3    OutputPort<Path> asciiPort = super.createOutputPort();
4    OutputPort<Path> binPort = super.createOutputPort();
5    OutputPort<Path> elsePort = super.createOutputPort();
6
7    @Override protected void execute() {
8      Path path = inputPort.receive();
9      if (path == null) { return; }
10     if (path.endsWith("ascii")) {
11       asciiPort.send(path);
12     } else if (path.endsWith("binary")) {
13       binPort.send(path);
14     } else {
15       elsePort.send(path);
16     }
17   }
18
19   public InputPort<Path> getInputPort() {
20     return inputPort;
21   }
22   ... // getter for each output port
23 }
```

V. Conclusion

In this paper, we briefly presented our Pipe-and-Filter (P&F) framework TeeTime. We explained how TeeTime is able to model and to execute arbitrary P&F architectures and how it schedules multiple stages in parallel. Moreover, we presented a small P&F example built with TeeTime. In particular, it highlights that TeeTime hides the thread management and the internal use of different pipe implementations from the user.

As future work, we plan to combine TeeTime with Akka in order to support fault-tolerant, distributed P&F architectures. Additionally, we work on a graphical live visualization of running TeeTime-based P&F architectures. We also experiment with further execution models to automatically optimize the throughput of P&F architectures. Finally, we plan a tool evaluation by comparing TeeTime with other frameworks.

References

[1] *Akka Framework*. http://akka.io.

[2] Marco Aldinucci et al. "FastFlow: high-level and efficient streaming on multi-core". In: *Programming Multi-core and Many-core Computing Systems*. Wiley, 2014.

[3] Robert Allen and David Garlan. "Towards Formalized Software Architectures". In: *Recent Trends and Developments*. Vol. 1000. LNCS. Springer, 1992.

[4] Frank Buschmann et al. *Pattern-oriented Software Architecture: A System of Patterns*. Wiley & Sons, 1996.

[5] Gunnar Dittrich and Christian Wulf. "Extraction of Operational Workflow-based User Behavior Profiles for Software Modernization". In: *Proc. of the Symposium on Software Performance*. 2016.

[6] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. "Software Landscape and Application Visualization for System Comprehension with ExplorViz". In: *Information and Software Technology* (2016).

[7] John Giacomoni, Tipp Moseley, and Manish Vachharajani. "FastForward for Efficient Pipeline Parallelism: A Cache-optimized Concurrent Lock-free Queue". In: *the Proc. of the PPoPP*. 2008.

[8] Swapna S. Gokhale and Sherif M. Yacoub. "Reliability Analysis of Pipe and Filter Architecture Style." In: *the Proc. of the 18th SEKE*. 2006.

[9] Michael I. Gordon et al. "Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs". In: *the Proc. of ASPLOS*. 2006.

[10] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis". In: *Proc. of the ICPE*. ACM, 2012.

[11] *JCTools Library*. https://github.com/JCTools/JCTools.

[12] M. Shaw. "Larger Scale Systems Require Higher-level Abstractions". In: *ACM SIGSOFT SEN* 14.3 (1989).

[13] *Spark Framework*. http://spark.apache.org/streaming.

[14] *Storm Framework*. http://storm.apache.org.

[15] Jeremy Sugerman et al. "GRAMPS: A Programming Model for Graphics Pipelines". In: *ACM Transactions on Graphics* 28.1 (2009).

[16] *TeeTime (C++)*. https://git.io/vDHmm.

[17] *TeeTime (Java)*. https://teetime-framework.github.io.

[18] *The Kieker Project*. http://kieker-monitoring.net.

[19] Junchang Wang et al. "B-Queue: Efficient and Practical Queuing for Fast Core-to-Core Communication". In: *Int. Journal of Parallel Programming* 41.1 (2013).

[20] Matt Welsh, David Culler, and Eric Brewer. "SEDA: An Architecture for Well-conditioned, Scalable Internet Services". In: *SIGOPS Oper. Syst. Rev.* 35.5 (2001).

[21] Christian Wulf, Christian Claus Wiechmann, and Wilhelm Hasselbring. "Increasing the Throughput of Pipe-and-Filter Architectures by Integrating the Task Farm Parallelization Pattern". In: *Proc. of the CBSE*. 2016.