Toward a Generic and Concurrency-Aware Pipes & Filters Framework

Symposium on Software Performance 2014

Christian Wulf,
Nils Christian Ehmke, and
Wilhelm Hasselbring — 28.11.2014

Software Engineering Group
Kiel University, Germany
Current State of Kieker

already optimized [Wa14]

Analysis component originally made for a modular offline analysis
⇒ Too slow for live analysis and more complex offline analysis
⇒ Not that modular anymore due to software erosion
Design Limitations of Kieker’s Analysis Component

- Determination of target stage at runtime
  - Using Java’s Reflection API
  - Causing unnecessary and high synchronization overhead
- Check for type safety on each deliver
  - Using annotation-based ports
- Only readers are executed in a separate thread
- Limited support for stage composition

pipes are no first-class entities [Shaw93]
Related P&F Frameworks

• ExplorViz’ trace processing component [FWBH13]
  – Tailored to trace processing
  – Not intended as generic P&F framework

• Java 8 streams [Oracle14]
  – No support for ports
  – Limited configuration options

• Akka [Akka14]
  – No direct support for ports
  – Less efficient due to its actor-based design
    • e.g., due to MpSc mailbox
Requirements

- Abstraction from platform/implementation details (Abs)
- Focus on efficiency and scalability (E&S)
- Type safety (TS)
- Compositional, reusable stages (CompS)
Agenda

• Motivation
• Proposal for a P&F Software Architecture
• Preliminary Performance Results
• Conclusion
Pipe as First-Class Entity

- **Pros:**
  - Encapsulates stage scheduling
  - Encapsulates synchronization
  - Allows the compiler to perform further optimizations

- **Cons:**
  - Increases runtime overhead due to delegation
Stage Scheduling & Synchronization

Toward a Generic and Concurrency-Aware Pipes & Filters Framework

- Unsynchronized
- Direct method call
- Backpressure technique

No additional global scheduler necessary: each thread decides for itself what to execute

- Lock-free SpSc queue\(^1\)
- All IPC protocols possible (e.g., asynch send + polling)

\(^1\) Java Concurrency Tools: https://github.com/JCTools/JCTools
Multi-Threaded Execution

Our sample pipeline using a hybrid thread assignment approach

- Flexible thread assignment: distinct [SQKP10], shared [SLY+11], and duplicated execution
- Minimal communication overhead by using intra- and inter-thread pipes
A stage composed of multiple other stages
Basic, Reusable Stages

- **Distributor/Merger**
- **File system stages:**
  - Reader/writer, text file line processing
- **Generic stages:**
  - Repeater, delay, throughput, instanceOf
- **Further stages:**
  - Clock, word counter, en-/decryption, (de)compress
## Preliminary Performance Evaluation

### Kieker’s analysis component

<table>
<thead>
<tr>
<th></th>
<th>Monitoring (S1)</th>
<th>TCP Reading (S2)</th>
<th>Reconstruction (S3)</th>
<th>Reduction (S4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>2.7370</td>
<td>16.2016</td>
<td>1578.8127</td>
<td>1566.1206</td>
</tr>
<tr>
<td>Ci95</td>
<td>0.0326</td>
<td>0.0395</td>
<td>38.2491</td>
<td>37.9230</td>
</tr>
<tr>
<td>Throughput</td>
<td>365360</td>
<td>61722</td>
<td>633</td>
<td>638</td>
</tr>
</tbody>
</table>

Performance results of Kieker’s analysis component with MooBench [Wa14] as load driver

### TeeTime

<table>
<thead>
<tr>
<th></th>
<th>Monitoring (S1)</th>
<th>TCP Reading (S2)</th>
<th>Reconstruction (S3)</th>
<th>Reduction (S4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>2.7594</td>
<td>17.5330</td>
<td>17.1577</td>
<td>17.3352</td>
</tr>
<tr>
<td>Ci95</td>
<td>0.0075</td>
<td>0.0095</td>
<td>0.0202</td>
<td>0.0245</td>
</tr>
<tr>
<td>Throughput</td>
<td>362395</td>
<td>57035</td>
<td>91x</td>
<td>58282</td>
</tr>
</tbody>
</table>

Performance results of TeeTime with MooBench [Wa14] as load driver

### TeeTime

<table>
<thead>
<tr>
<th></th>
<th>Monitoring (S1)</th>
<th>TCP Reading (S2)</th>
<th>Reconstruction (S3)</th>
<th>Reduction (S4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>-</td>
<td>2.3995</td>
<td>9.3671</td>
<td>7.4789</td>
</tr>
<tr>
<td>Ci95</td>
<td>-</td>
<td>0.0062</td>
<td>0.0463</td>
<td>0.0389</td>
</tr>
<tr>
<td>Throughput</td>
<td>-</td>
<td>416748</td>
<td>168x</td>
<td>106756</td>
</tr>
</tbody>
</table>

Performance results of TeeTime with a faster load driver
• **Flexible** P&F software architecture including an *open source implementation*
• **Optimized** in terms of abstraction, efficiency & scalability, type safety, and reusability
• **90-200x** performance increase w.r.t. Kieker’s current analysis component

Future work:
• Further performance experiments
• Automatize as much as possible:
  – pipe instantiation, thread assignment, and exception handling
• Support for Kieker’s WebGUI via save/load of pipeline structures [E13]
• Migration of legacy P&F architectures [Wu14]
References


Two common thread assignment strategies:

- Assign each worker thread on a **distinct set** of stages [e.g., SQKP10]
- Assign each worker thread to **all** stages sharing the pipes [e.g., SLY+11]
Signal Concept

- Automatic passing of signals by the framework
- Signal encapsulates the logics
- Integrated signals for starting, validating, terminating
- Arbitrary signals possible
```java
public class Directory2FilesFilter extends ConsumerStage<File> {

    private final OutputPort<File> outputPort = this.createOutputPort();

    private FileFilter filter;
    private Comparator<File> fileComparator;

    // omitted constructors

    @Override
    protected void execute(final File inputDir) {
        final File[] inputFiles = inputDir.listFiles(this.filter);

        if (inputFiles == null) {
            this.logger.error("Directory " + inputDir + " does not exist or an I/O error occurred.");
            return;
        }

        if (this.fileComparator != null) {
            Arrays.sort(inputFiles, this.fileComparator);
        }

        for (final File file : inputFiles) {
            this.send(this.outputPort, file);
        }

    // omitted getter and setter

    public OutputPort<File> getOutputPort() {
        return outputPort;
    }

    }
```
Performance Evaluation - Setup

- Kieker (11.07.2014) vs. TeeTime (01.09.2014)
- Load: 2 resp. 200 million traces each with a depth of 10
- Four scenarios: S1-S4
- Single-threaded execution of the analysis side
More Efficient and Scalable

Static connection of ports
  – No dynamic search of annotation-based ports

Single type check phase on initialization
  – No dynamic type check on each delivery

Direct stage invocation
  – No reflection call

Flexible thread assignment
  – No limitation to readers