Conception and Development of a Pipe & Filter Framework for C++

Master’s Thesis

Johannes Ohlemacher

October 30, 2016

KIEL UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
SOFTWARE ENGINEERING GROUP

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

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 30. Oktober 2016
Abstract

The Pipe-and-Filter architectural style splits the functionality of a software system into clearly separated components. This does not only improve the modularity of a software system, but also opens up possibilities to easily run parts of that software system in parallel and therefore increase the throughput of the system and it’s performance in general.

The TeeTime Pipe-and-Filter framework for Java does not only provide ways to create Pipe-and-Filter architectures, but it also contains many ready-to-use components that implement commonly used functionality.

In this thesis, we implement the TeeTime Pipe-and-Filter framework for C++. The C++ programming language is in a lot of aspects very different from Java and we take that into account. We make sure that all the features typical C++ programs rely on, are usable with TeeTime. An example of this are value semantics.

The ability to run different components in parallel is one the key benefits of Pipe-and-Filter architectures. Lockfree Single-Producer Single-Consumer queues are used to pass data from one thread to the next. Despite the fact that there are multiple ready-to-use implementations of such a queue available, we implement such a queue ourselves to achieve the best performance possible.

We do a performance evaluation of that queue to show how it outperforms other implementations. We also perform a performance evaluation of TeeTime for C++ as a whole to analyze how TeeTime for C++ scales on multicore system compared to the similar framework FastFlow. This evaluation reveals that by eliminating the need for manual memory management, TeeTime for C++ does not only allow a better modularity, but also provides a better performance in most cases.
# Contents

1 Introduction ................................................................. 1  
1.1 Motivation ...................................................................... 1  
1.2 Document Structure ...................................................... 1  

2 Foundations and Technologies .............................................. 3  
2.1 The Pipe & Filter Architectural Style ................................. 3  
2.2 Lockfree Single-Producer Single-Consumer Queues .............. 4  
2.2.1 Lamport’s Queue ...................................................... 4  
2.2.2 FastForward Queue .................................................. 5  
2.3 The Pipe & Filter Framework TeeTime ............................... 5  
2.4 The FastFlow Parallel Programming Framework .................. 8  
2.5 The C++11 Programming Language .................................... 11  
2.5.1 Value Semantics ..................................................... 11  
2.5.2 Move Semantics ..................................................... 12  
2.5.3 Dynamic Dispatch / Virtual Function Calls .................... 14  

3 Implementation of a SPSC Queue ........................................ 15  
3.1 Overview of Existing Implementations ............................... 15  
3.2 Implementing a Value based FastForward Queue ................. 17  

4 Implementation of TeeTime for C++ .................................... 21  
4.1 Overview ....................................................................... 21  
4.2 The Support Layer ........................................................ 21  
4.2.1 Platform .................................................................... 21  
4.2.2 Logging .................................................................... 23  
4.3 Basic Stages ................................................................... 23  
4.3.1 AbstractStage .......................................................... 23  
4.3.2 AbstractProducerStage .............................................. 25  
4.3.3 AbstractConsumerStage .......................................... 25  
4.3.4 AbstractFilterStage ................................................. 25  
4.3.5 Distributor ............................................................... 26  
4.3.6 Merger ..................................................................... 26  
4.4 Ready-To-Use Stages ...................................................... 27  
4.4.1 InitialElementProducer ............................................. 27  
4.4.2 CollectorSink .......................................................... 28  
4.4.3 Directory2Files ....................................................... 28
Contents

Bibliography 77
Chapter 1

Introduction

1.1 Motivation

The Pipe-and-Filter style is a widely known architectural style for applications that do streaming-like processing of data. The style splits the processing into many small filters, that can be connected via pipes. The Pipe-and-Filter architectural style provides a great deal of modularity and flexibility, and also enables leveraging contemporary distributed systems and multi-core systems for a high throughput.

TeeTime is a Pipe-and-Filter Framework for Java. It was developed at the Kiel University and supports the modeling and execution of Pipe-and-Filter architectures. TeeTime uses the more generic term stage instead of filter and we do so as well. It features not only basic tools to create custom stages with low effort, but also many ready-to-use primitive and composite stages. The framework has no single-threaded overhead and only a minimal overhead in multi-threaded scenarios. The framework has proven to be very extensible and flexible in various research projects ([Strubel 2016], [Wulf et al. 2016]).

Having such a flexible framework for Pipe-and-Filter architectures is not only valuable for Java, but also for other languages. Especially C++ is interesting in this regard. C++ provides facilities for object-oriented programming that are quite similar to those of Java, but at the same time a lot of C++ basic principles and semantics are very different from it.

This raises the question how TeeTime can be brought into the C++ world while keeping its flexibility and performance and also leveraging C++ unique features like native code, manual memory management and value semantics.

In this thesis we describe very detailed how this can be done and compare our solution not only to similar C++ frameworks, but also to the reference implementation of TeeTime for Java.

1.2 Document Structure

Chapter 2 gives an overview of the foundations and technologies that we use in this thesis. Afterwards chapter 3 describes in detail the implementation of the SPSC queue, that is used in TeeTime for C++. Chapter 4 contains the description of the implementation of TeeTime itself and focusses on the differences to the Java reference implementation. The previously mentioned SPSC queue will be evaluated in chapter 5, and chapter 6 will not only clarify
1. Introduction

how TeeTime as a whole is being evaluated and compared against a similar framework, but will also present and explain the actual evaluation results. In chapter 7 we discuss related work of this thesis, before chapter 8 draws a final conclusion and presents an outlook on possible future work in this field.
Chapter 2

Foundations and Technologies

This chapter gives an overview of foundations and technologies, which are used in the thesis.

2.1 The Pipe & Filter Architectural Style

The Pipe-and-Filter style is a widely known architectural style for applications that do streaming-like processing of data. The style splits the processing into many small filters/stages. Each stage reads a stream of inputs on an input port, does some processing on it and writes the results to one or more output ports. The output port of a stage can be connected to an input port of the subsequent stage by a pipe. This way a complex pipeline can be defined, where data flows from stage to stage while it gets processed. Such a pipeline does not have to be strictly linear, but feedback loops and branches are also allowed.

![Figure 2.1. Example of a linear Pipe-and-Filter architecture](image)

An example for a Pipe-and-Filter architecture is illustrated in Figure 2.1. Filters and their ports are represented by rectangles, while arrows indicate pipes. In the example list generates names of image files on the hard drive. Each name is passed over the the load filter to decode that image into memory. After that the resize filter will resize that image before the save filter saves the image back to hard drive. The execution of each filter depends only on its input, providing an opportunity for parallelization. The resize filter can resize an image, while the load filter already loads the next image and the save filter writes the previous image to the hard drive. An alternative approach to leverage parallelism with a Pipe-and-Filter architecture is to branch the pipeline as depicted in Figure 2.2. The part of the pipeline that does the actual processing (load, resize, and save filter) gets duplicated for every available processor. The additional dist filter distributes the incoming file names to the subsequent load filters and therefor to different processors.
2. Foundations and Technologies

Figure 2.2. Example of a branched **Pipe-and-Filter** architecture

Having multiple stages running in parallel raises the problem, that those stages have to communicate in some way to pass data. Synchronization is therefore a critical factor to achieve good performance in multi-threaded scenarios.

2.2 Lockfree Single-Producer Single-Consumer Queues

Efficient pipeline parallelism requires that the communication between two subsequent stages running in parallel induce as little overhead as possible. In the context of two subsequent stages, the first stage only produces data and the second one only consumes data, so a **Single-Producer Single-Consumer** (SPSC) queue is sufficient for our purpose here. This section describes two different approaches for implementing such a lock-free SPSC queue.

2.2.1 Lamport’s Queue

Lamport [Lamport 1983] proved that his implementation (Figure 2.3) of a Single-Producer Single-Consumer queue does not need any explicit synchronization. While this is true, it is important to note that this implementation still has some kind of implicit synchronization. To determine whether an element of the queue can be read or written, the head and tail indices must be compared. The producer writes only to the head index of queue, and the consumer only writes to the tail index of queue, but both of them always need to read the other index as well. On CPUs with caching, this will lead to continuously synchronization of caches and therefore thrashing the cache coherence protocol.
2.3. The Pipe & Filter Framework TeeTime

TeeTime is a Pipe-and-Filter Framework for Java. It provides support for the modeling and the execution of Pipe-and-Filter architectures. TeeTime targets software engineers that need to process data in a stream-oriented, throughput-optimized, and type-safe fashion.

```java
1 bool enqueue(data) {
2     if (NEXT(head) == tail) {
3         return false;
4     }
5     buffer[head] = data;
6     head = NEXT(head);
7     return true;
8 }
9 }
10
```

Figure 2.3. Lamport’s queue implementation

2.2.2 FastForward Queue

The FastForward queue [Giacomoni et al. 2008; Aldinucci et al. 2012] is based on the idea of avoiding the shared access to head and tail indices of the Lamport queue. Producer and consumer still maintain a head and tail index, but those are completely independent from each other. As can be seen on Figure 2.4 the producer only reads and writes the head index, while the consumer only reads and writes the tail index. To determine whether an element of the queue can be read or written, the element itself is being accessed. If the element of the queue is NULL it is considered empty, so the producer can write to it. The consumer reads from an element and if it is non-NULL, the consumer consumes the value and sets the element to NULL afterwards. Since producer and consumer are synchronized by the queue element they want to access and not by the head and tail indices, synchronization is distributed over the entire queue. As long as producer and consumer are not accessing elements on the same cacheline, they can operate completely independently from each other and the CPU caches don’t need to synchronize.

```java
1 bool enqueue(data) {
2     assert(data != NULL);
3     if (buffer[head] == NULL) {
4         buffer[head] = data;
5         head = NEXT(head);
6     return true;
7     }
8     return false;
9 }
10
```

Figure 2.4. FastForward queue implementation

2.3 The Pipe & Filter Framework TeeTime

TeeTime is a Pipe-and-Filter Framework for Java. It provides support for the modeling and the execution of Pipe-and-Filter architectures. TeeTime targets software engineers that need to process data in a stream-oriented, throughput-optimized, and type-safe fashion.
2. Foundations and Technologies

TeeTime is written in Java, but the framework’s software architecture (see Figure 2.5) is intended to be easily adaptable to other object-oriented programming languages that are aware of threads and type parameters [Wulf et al. 2014]. TeeTime is based on the Tee-and-Join-Pipeline design pattern, 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.

```java
public class MyDoubleToStringStage extends AbstractStage {
    private InputPort<Double> inputPort = this.createInputPort();
    private OutputPort<String> outputPort = this.createOutputPort();

    public InputPort<Double> getInputPort() {
        return this.inputPort;
    }

    public OutputPort<String> getOutputPort() {
        return this.outputPort;
    }

    @Override
    protected final void execute() {
        final Double element = this.getInputPort().receive();
        if (null == element) {
            returnNoElement();
        } else {
            String s = element.toString();
            this.getOutputPort().send(s);
        }
    }
}
```

Three design decisions of TeeTime are of peculiar interest:

1. Input and output ports are modeled as first class citizens. Every stage can provide as many input and output ports as needed. This allows to easily implement arbitrary stages and architectures with arbitrary complex branching. An example declaration of a stage with input and output ports is depicted on Figure 2.6.

2. TeeTime distinguishes between active and non-active stages. Every stage is non-active by default, but can be set to an active state. For each active stage a dedicated thread gets
created by the framework to execute that particular stage. Active stages pull data from
their input ports for processing. For non-active stages no thread gets created. Those
stages are being executed by the predecessor stage. The predecessor stage pushes data
to the non-active stage and executes the stage for that data.
This design decision allows to develop stages as encapsulated, self-contained functional
units, that do not enforce any kind of parallelism. Modularity and parallelism are fully
orthogonal to each other and can be defined and modified independently.
An active stage can have an arbitrary number of non-active subsequent stages, they get
all executed by that one active stage. If two stages get connected and the second one
is non-active, the framework creates a *unsynchronized* pipe between them. This pipe just
forwards the data to the second stage for execution. If the second stage is active the two
stages are being executed by different threads and therefore some synchronization is
required when passing data from one stage to the next. To achieve this, the framework
creates a *synchronized* pipe between them. This pipe is at its core a single-producer
single-consumer queue, that the first stage pushes data to and the second stage pulls data from.

3. TeeTime uses Java generics to achieve type-safety. All ports, pipes and abstract base
classes for stages have type parameters, so the compiler can check whether the type
of input and output ports of subsequent stages match. This way invalid connections
between stages can be detected by the compiler and the program will fail to compile.

```
public class MyConfiguration extends Configuration {
    private ProducerStage<Long> producer;
    private FirstProcessorStage<Long> processor1;
    private SecondProcessorStage<Long> processor2;
    private SinkStage<Long> sink;

    public MyConfiguration() {
        this.producer = new ProducerStage<Long>();
        this.processor1 = new FirstProcessorStage<Long>();
        this.processor2 = new SecondProcessorStage<Long>();
        this.sink = new SinkStage<Long>();

        this.producer.declareActive();
        this.processor2.declareActive();

        connectPorts(this.producer.getOutputPort(), this.processor1.getInputPort());
        connectPorts(this.processor1.getOutputPort(), this.processor2.getInputPort());
        connectPorts(this.processor2.getOutputPort(), this.sink.getInputPort());
    }
}
```

Figure 2.7. Example of a TeeTime Pipeline Configuration

Figure 2.7 illustrates how a Pipe-and-Filter configuration can look like. Four stages
are being created (line 8 to 11) and two of them are being set as active (producer
in line 13 and processor2 in line 14). The stages are being connected in the order
producer, processor1, processor2, sink, so processor1 runs in the same thread as producer.
2. Foundations and Technologies

and sink runs in the same thread as processor2. It is important to note here, that it is not necessary to create any explicit synchronization here. The TeeTime framework detects subsequent stages in different threads and connects them with a pipe that does the synchronization automatically.

2.4 The FastFlow Parallel Programming Framework

FastFlow\(^1\) is a C++ parallel programming framework advocating high-level, pattern-based parallel programming. It chiefly supports streaming and data parallelism and is therefore similar in focus to TeeTime.

FastFlow is designed as a stack of three layers as depicted in Figure 2.8. The lowest layer provides low-level basic mechanisms for lock-free synchronization. Based on that the second layer provides distinctive communications mechanisms supporting Single-Producer/Multiple-Consumer and Multiple-Producer/Single-Consumer communication. The third layer provides programming primitives to easily implement typical streaming patterns which exploit the available parallel hardware by utilizing the mechanisms provided by the lower layers.

Especially the second layer with its pipeline and farm pattern is interesting in the context of this thesis, because the same patterns are implementable with TeeTime.

The basic building blocks of FastFlow are nodes. FastFlow’s nodes are equivalent to TeeTime’s stages. Such a stage (or node) is in the terms of TeeTime always active. There is no way to run two stages in one thread. Every stage runs in its own thread and all communication between stages is done via queues. FastFlow is primarily meant to model the parallelism of an application, not its modularity.

Another major differences between TeeTime and FastFlow is that input and output ports are not explicitly modeled. A stage always has one implicit input port and one implicit

\(^1\)http://calvados.di.unipi.it/
output port. That is even true for stages that produce no output (like a sink stage) or process no input (like a producer stage).

The pipeline pattern (or pipeline skeleton as it is called by FastFlow) consists of one or more stages that are connected in a sequential fashion. There is a way to enable a feedback loop for such a pipeline, so that the output gets feed back in as input into the pipeline, but it is not possible to create arbitrary branches.

```cpp
1 ff_node_F <char, myTask> F1(f1), F2(f2), F3(f3);
2 ff_Pipe<> pipe1(F1, F2, F3, make_unique<Deleter>());
3 pipe1.run_and_wait_end();
```

Figure 2.9. Example of a FastFlow Pipeline Configuration

The farm pattern lets you define a producer stage (also called emitter node by FastFlow) and an arbitrary number of worker stages. The output of the producer stage is distributed to the worker stages to achieve higher throughput. You can optionally also define a merger stage (or collector node as it is called by FastFlow), to merge the results of all the worker threads back into one thread.

```cpp
1 // untyped node/stagethroughput
2 class ff_node {
3 public:
4   virtual void* svc(void* data) = 0;
5 };
6
7 // wrapper type to provide type safety
8 template<typename Tin, typename TOut>
9 class ff_node_t : public ff_node {
10 public:
11   virtual void* svc(void* data) override {
12     return svc(reinterpret_cast<Tin*>(data));
13   }
14
15   virtual TOut* svc(TIn* data) = 0;
16 };
17
18 // custom stage implementation
19 class MyStage : public ff_node_t<std::string, int> {
20 public:
21   virtual int* svc(std::string* data) override {
22     int i = atoi(data->c_str());
23     delete data;
24     return new int(i);
25   }
26 }
```

Figure 2.10. Example of a FastFlow Pipeline Configuration

At its core FastFlow is not type-safe. Stages exchange data by passing untyped pointers around (void*), that must be casted to the correct type to access the data for processing. A simplified version of the basic `ff_node` class that all other stages are derived from is depicted in Figure 2.10. On top of that are several other base classes defined, that provide some amount of type-safety by wrapping up the unsafe interface with C++ templates.
2. Foundations and Technologies

As pointed out before, data has to be passed between stages as pointers. Those pointers must point to some data. The easiest solution to this is to let every stage dynamically allocate its output on the heap. Since this data must also be freed sometime, every stage also has to delete its input data after it is done with processing it. For a pipeline with \( N \) stages, this will lead to \( N - 1 \) allocations and \( N - 1 \) frees for every element that traverses the pipeline. This solution is used by most FastFlow tutorials and is depicted on Figure 2.11.

![Figure 2.11. FastFlow memory management with multiple allocations per task.](image)

Instead of allocating and freeing memory at every stage of the pipeline, an alternative solution is to allocate all memory needed by the pipeline upfront. The very first stage of the pipeline allocates a Task object, that holds all data items needed by the subsequent stages. The very last stage frees this object. All other stages in between just read and write data items from that task object. This leaves us with exactly one allocation and one free per element that traverses the pipeline, regardless of how many stages the pipeline has. Longer pipelines will probably lead to bigger and more complex task objects, but not to more allocations of them.

![Figure 2.12. FastFlow memory management with a single allocation per task.](image)
2.5 The C++11 Programming Language

The C++ programming language ([Stroustrup 2013]) was standardized 1998 (with just some minor changes in 2003 aka C++03). The next major revision of the standard was released 2011 under the name C++11 ([Meyers 2014]). C++11 brought a lot of changes not only to the language itself, but also to the standard library.

While the C++03 language provides a memory model that supports threading, the standard library does not provide any facilities to actually use threads. Programs written in C++03 had to rely on additional libraries or on platform specific (and therefore not portable) APIs to use threads. C++11 changed that by providing standardized and portable components for threads, synchronization and atomic operations.

All major C++ compilers (Microsoft Visual Studio, GCC, Clang) support most of C++11 since quite some time and major libraries like Qt, Unreal Engine, Facebook Folly, and Microsoft Casablanca are starting to not only support C++11 but to even require C++11.

C++11 is not the latest language standard. C++14 was released 2014, C++17 is expected to be released this year and a lot of its features are already available in recent compiler versions. Nevertheless relying on the latest language standard limits the number of available compilers. While offering useful new features, C++14 and C++17 do not offer any features that are critical for the thesis, so C++11 has been chosen for the thesis as a proper tradeoff between modern language features and available compilers and platforms.

2.5.1 Value Semantics

One of the major differences between Java and C++ are value semantics. With the exception of primitiv data types, all objects in Java are accessed via references. Passing an argument to a function in Java means usually copying a reference (not the actual object), which is a very cheap operation.

While the same thing can be achieved in C++ with pointers, there are some disadvantages. Despite the fact that memory allocation is often a performance bottleneck on its own, there is another catch: Unlike Java, C++ has no build-in garbage collection. Objects that were manually allocated must be manually freed or the program will run out of memory. Passing pointers to those objects around raises the question, when and where should those objects be freed. This problem is solvable by very carefully designing the used data structures and algorithms and there are tools to help with that. Nevertheless it’s still a non-trivial task and it gets very complicated and prone to errors very easily.

A different approach to this is, to not pass arguments by pointer or by reference to a function, but by value. This means the actual object is copied into a function. Depending on the type of the object, copying it might be an expensive operations (because the object is large, or it involves some non-trivial operations).

Figure 2.13 illustrates a very simple class that holds a pointer to a dynamically allocated array of integers and the size of that array. The array gets allocated in the constructor and is freed in the destructor. When we want to use objects of this class as values, we need to
2. Foundations and Technologies

```cpp
struct Array {
    Array(int size)
        : elements(new int[size])
        , size(size)
    ;

    ~Array() {
        delete[] this->elements;
    }

    int* elements;
    int size;
};
```

Figure 2.13. Simple C++ Array Class

define what copying them actually means. We do that by providing a copy constructor as is shown in Figure 2.14. The copy constructor allocates an array of the same size as the size of the copied-from array, and copies every single element in that array. Copying can be quite expensive, not only because memory needs to be allocated, but also because copying the array is \(O(n)\) complexity wise. In a lot of cases this can be improved via move semantics.

```cpp
Array(const Array& a) {
    this->elements = new int[a.size];
    this->size = a.size;
    for(int i=0; i<this->size; ++i) {
        this->arr[i] = a.arr[i];
    }
}
```

Figure 2.14. Copy-Constructor for Array Class

2.5.2 Move Semantics

Move semantics are an extension of value semantics. After copying an object into a function, the copied-from object is in many cases not needed anymore, so copying it in the first place is very wasteful. To solve that, C++11 brought so called move semantics. Move semantics allows us to move resources (like allocated memory) from one object to the other. Just like the copy constructor defines what copying an object means, the move constructor defines what moving an object means. Figure 2.15 shows a possible implementation of a move constructor for our Array class.

Note that no memory gets allocated or freed and the content of the array is not even accessed at all. The only thing the move constructor does is copying the pointer and the size from the moved-from object to itself and setting the pointer in the moved-from object to `nullptr` and the size to 0, making the moved-from object effectively empty. The last part is important, because the moved-from object must be left in a valid state, so that it can be

^2^In most cases you also want to implement an assignment operator, but this has been omitted here for brevity.
2.5. The C++11 Programming Language

```c++
Array(Array&& a) {
    this->elements = a.elements;
    this->size = a.size;
    a.elements = nullptr;
    a.size = 0;
}
```

**Figure 2.15.** Move-Constructor for Array Class

destroyed later just like all other objects. Moving an Array object is $O(1)$ in this case, no matter how many elements the array contains.

Usually a copy-constructor and a move-constructor is defined for a class, which raises the question when does an object get moved and when does it get copied? Defining copy and move constructor is possible because the constructor can be overloaded based on the argument type, just like a regular function. The argument type of the copy constructor is `const Array&`, which is a reference to a constant Array object. You are not allowed to modify that object, so moving from it is not possible. On the other hand the argument type of the move constructor is `Array&&`, which is a reference to a rvalue. The easiest way to think of rvalues are temporary values. Those values are only used once, so it’s safe to modify them and move their content somewhere else, because the value is not needed afterwards anyway. All returned values from function calls are rvalues. You can also explicity cast a value to a rvalue by calling `std::move`. All non-rvalues can be thought of beeing lvalues\(^3\). They have a name and they can be accessed multiple times, so modifying them will affect the behavior of the program. Rvalue references can not only be used with constructors, but with all functions. Figure 2.16 illustrates some examples.

```c++
void f(const Array& a); // (1)
void f(Array&& a); // (2)
Array g();
Array a(42);

f(a);  // calls (1), since a is an lvalue
f(std::move(a)); // calls (2), since a is casted to rvalue
f(g());  // calls (2), since returned value
// from g() is implicity an rvalue
```

**Figure 2.16.** Function overloading based on rvalues

If a type doesn’t define a move constructor or is not moveable in a meaningful way, moving objects of such a type is identical to copying them. Figure 2.17 depicts such a type that is not moveable, because instances of that type hold no resources that can be passed to another instance.

\(^3\)The C++ standard defines a couple more value types, but those are mainly of interest for compiler writers. TODO: cite actual C++ ISO standard here
2. Foundations and Technologies

```
struct pod {
  float f;
  int i;
  double d;
  char s[128];
};
```

Figure 2.17. Non-moveable plain old data (pod) type

2.5.3 Dynamic Dispatch / Virtual Function Calls

The key abstractions of TeeTime are stages, pipes and ports. For each one of them exist multiple implementations, that override a method from an abstract base class or an interface. Dynamic method dispatch is a mechanism by which a call to a overridden method is resolved at runtime. This allows a great deal of flexibility because its possible to swap out every implementation at runtime. The same thing is possible in C++ thanks to virtual member functions. This flexibility comes at a cost, even if you don’t need it:

1. Additional pointer per object.
2. Extra indirection (pointer dereference) for each call to a virtual member function.
3. Virtual function calls usually can’t be inlined by the compiler.

While the first two issues are probably neglectable, the third one might cause a performance hit [Bendersky 2013]. It hinders the compiler from doing one of its most important optimizations (inlining function calls). Unlike Java there is no virtual machine or just-in-time-compilation to fix this at runtime.
Chapter 3

Implementation of a SPSC Queue

As mentioned in Section 2.2 communication between threads is crucial for the performance of a Pipe-and-Filter architecture. In case of stages running in different threads, bounded lock-free queues are the way to go and choosing such a lock-free queue implementation is an important decision for a framework like TeeTime.

3.1 Overview of Existing Implementations

There exist various different ready-to-use implementations of lockfree Single-Producer-Single-Consumer (SPSC) queues, so before implementing such a queue ourselves, we should check if we could use one of those.

- **boost**\(^1\) is a collection of C++ libraries, providing functionality for tasks and structures such as linear algebra, multithreading, file system and memory management. Among its lock-free data structures is also SPSC queue (boost::spsc_queue).

- **folly**\(^2\) is also a collection of C++ libraries. It was originally developed internally at Facebook\(^3\), but has been released as an open source project. Folly as a whole does not support the Windows platform, but it contains a SPSC queue (folly::ProducerConsumerQueue) that is self-contained and implemented with standard C++11. Therefore that SPSC queue can be extracted easily and be used on all platforms.

- **FastFlow** implements its own SPSC queue, called ff:SWSR_Ptr_Buffer. This implementation of a SPSC queue is tailored towards the specific needs of FastFlow and is based on FastFlows own platform abstraction layer to make use of atomic operations and memory fences. In contrast to the queues from folly and boost, ff:SWSR_Ptr_Buffer is not a class template. Just like the name suggests, it is a buffer for arbitrary un-typed pointers (void*). It is not type-safe and does not support arbitrary values (only pointers). In fact there are two variants of the queue available: one where the pointers in the buffer are treated as volatile and one where the indices into the buffer are marked volatile. Depending on whether the macro NO_VOLATILE_POINTERS is defined, the first

---

\(^1\)http://boost.org
\(^2\)https://github.com/facebook/folly
\(^3\)www.facebook.com
3. Implementation of a SPSC Queue

```c
class SWSR_Ptr_Buffer {
...
#if defined(NO_VOLATILE_POINTERS)
  unsigned long pread;
  long padding1[longxCacheLine-1];
  unsigned long pwrite;
  long padding2[longxCacheLine-1];
#else
  ALIGN_TO_PRE(CACHE_LINE_SIZE)
  volatile unsigned long pread;
  ALIGN_TO_POST(CACHE_LINE_SIZE)
  ALIGN_TO_PRE(CACHE_LINE_SIZE)
  volatile unsigned long pwrite;
  ALIGN_TO_POST(CACHE_LINE_SIZE)
#endif
  const unsigned long size;
  void ** buf;
...}
inline bool empty() {
#if defined(NO_VOLATILE_POINTERS)
  return (*((volatile unsigned long *)&buf[prend]) == 0);
#else
  return (buf[prend] == NULL);
#endif
}
inline bool available() {
#if defined(NO_VOLATILE_POINTERS)
  return (*((volatile unsigned long *)&buf[pwrite]) == 0);
#else
  return (buf[pwrite] == NULL);
#endif
...}
```

*Figure 3.1. FastFlow’s NO_VOLATILE_POINTERS macro*

or the second variant is being used. Unfortunately the macro and its intention is not documented. Figure 3.1 depicts how the macro controls the implementation of the queue. By default the macro is not defined, so not the pointers, but the indices are treated as volatile. When we evaluate the different queue implementations (see Chapter 5) we consider both variants.

The table in Figure 3.2 summarizes the available SPSC queue implementation and lists their implementation type and their available features. Several papers claim FastForward based queues to be significant better for performance [Aldinucci et al. 2012; Giacomoni et al. 2008]. Unfortunately the only implementation of such a queue is the one from FastFlow, but that one suffers from missing type-safety and missing support for value semantics.

There is no implementation of a FastForward queue available that is based on modern C++11 and does support value semantics, so the following section describes how we can implement such a queue ourselves.
3.2. Implementing a Value based FastForward Queue

Like mentioned in Section 2.2.2, the FastForward queue is based on the idea, that a special value marks an element as empty. In case of pointers (like FastFlows pointer buffer), the NULL pointer is usually used for that. Java implementations of such a queue use the null reference. The memory layout of such implementations is depicted in Figure 3.3. The fourth element of that queue is empty, because it contains the NULL pointer. All other elements contain a non-NULL pointer, that is pointing to a value somewhere in memory.

Copying pointers in and out of such a queue is very cheap, but there are three disadvantages to this approach:

1. The queue itself contains only pointers to the values, but it does not manage or provide memory to actually store values. Memory management has to happen manually outside of the queue. Memory for the values must be allocated and deallocated at the right place and at the right time. This is not a trivial task and faults can lead to fatal errors.

2. To access the actual values, the pointer has to be dereferenced. Depending on how the memory is managed, the individual values can be very distributed in memory, resulting in bad cache locality.

3. Extra effort is required to make sure no NULL pointers are being inserted (maybe unintentional) into the queue. Because of the special meaning of the NULL pointer, this destroys the semantics of the queue.

Some of these issue can be solved by storing the values directly inside the queue, but this raises the question of how an element of the queue should be marked as empty. In the

<table>
<thead>
<tr>
<th>Queue</th>
<th>type</th>
<th>type safe</th>
<th>value semantics</th>
<th>move semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>folly::ProducerConsumerQueue</td>
<td>Lamport</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>boost::spsc_queue</td>
<td>Lamport</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>ff::SWSR_Ptr_Buffer</td>
<td>FastForward</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Figure 3.2. Feature Overview of existing SPSC queue implementations

Figure 3.3. Memory layout example of a pointer-based queue.

3.2 Implementing a Value based FastForward Queue

Like mentioned in Section 2.2.2, the FastForward queue is based on the idea, that a special value marks an element as empty. In case of pointers (like FastFlows pointer buffer), the NULL pointer is usually used for that. Java implementations of such a queue use the null reference. The memory layout of such implementations is depicted in Figure 3.3. The fourth element of that queue is empty, because it contains the NULL pointer. All other elements contain a non-NULL pointer, that is pointing to a value somewhere in memory.

Copying pointers in and out of such a queue is very cheap, but there are three disadvantages to this approach:

1. The queue itself contains only pointers to the values, but it does not manage or provide memory to actually store values. Memory management has to happen manually outside of the queue. Memory for the values must be allocated and deallocated at the right place and at the right time. This is not a trivial task and faults can lead to fatal errors.

2. To access the actual values, the pointer has to be dereferenced. Depending on how the memory is managed, the individual values can be very distributed in memory, resulting in bad cache locality.

3. Extra effort is required to make sure no NULL pointers are being inserted (maybe unintentional) into the queue. Because of the special meaning of the NULL pointer, this destroys the semantics of the queue.

Some of these issues can be solved by storing the values directly inside the queue, but this raises the question of how an element of the queue should be marked as empty. In the
3. Implementation of a SPSC Queue

| true | value1 | true | value2 | true | value3 | false | true | value4 |

Figure 3.4. Memory layout example of a value-based queue.

In the case of pointers the NULL pointer is pretty convenient for this, because it already has a special meaning. If we want to store arbitrary values (like strings or matrices) in the queue, finding such a special value is very hard. Depending on the actual use case, it might be possible to define the empty string or the null matrix as this special value, but even then it is very prone to errors. Another issue is the actual comparison. The check whether a queue element is empty or not, you have to compare it against the given null value, but this comparison can be quite expensive (e.g. if the queue contains $N \times M$ matrices, each comparison requires the comparison of $N \times M$ matrix elements).

Our approach avoids those issues by splitting each queue entry into two parts: a flag to store whether the entry is empty or not, and a block of memory to store the actual value (if non-empty). The depicted queue on Figure 3.4 demonstrates how the memory layout of such a queue looks like. This queue contains the same data as the previously mentioned pointer based queue on Figure 3.3. The fourth entry is empty (the flag is false), while all other entries have a value (the flag is true). All values are directly embedded into the queue and the sequential layout of the queue provides optimal cache locality.

Figure 3.5 depicts how such a queue entry is implemented in C++. hasValue is the flag used to store whether the entry contains a value or not. The flag is being read and written atomically, making use of C++11 atomics. buffer provides storage memory for an optional value. Two things are important to note here: The first thing is, that buffer is not of type T. The reason is, that we don’t want to construct a value of type T if there actually is no value present. Instead we want to provide a buffer of raw memory to hold a value of type T and we do that with an array of bytes (char). The second thing is a direct consequence of that. Each data type in C++ requires a certain memory alignment, which means the memory address of an instance of that type must be a multiple of that alignment. Usually the compiler takes care of that by inserting padding bytes into structures and allocating correctly aligned memory, but since buffer is not of type T, we have to take care of memory alignment ourselves. The alignas(T) directive makes sure, the array is aligned as it were an instance of type T. The compiler will insert padding bytes between the flag and the buffer to make this happen. This also means the entry object gets bigger. For example let’s assume the queue holds 32bit integers. On most platforms such an integer has an alignment of four bytes. As a result, the entry object is eight bytes big: one byte for the flag, three padding bytes (created by the compiler) and 4 bytes for the actual value.

Figure 3.6 depicts how the actual queue is structured. The read index is accessed by the consumer and the write index by the producer. Those indices are separated from each other and from other attributes by explicitly added padding bytes. The number of padding bytes equals the size of a CPU cacheline (usually 64) to make sure, no other data is stored on the same cacheline. This guarantees that the index can be read and written
3.2. Implementing a Value based FastForward Queue

```cpp
template<typename T>
struct Entry {
    atomic<bool> hasValue;
    alignas(T) char buffer[sizeof(T)];

    T* ptr() {
        return reinterpret_cast<T*>(&buffer[0]);
    }
};
```

**Figure 3.5. SpscValueQueue Entry**

```cpp
template<typename T>
class SpscValueQueue {
    char _padding1[CACHELINE_SIZE];
    size_t m_readIndex;
    char _padding2[CACHELINE_SIZE];
    size_t m_writeIndex;
    char _padding3[CACHELINE_SIZE];
    const unique_ptr<Entry<T>[]> m_entries;
    const size_t m_capacity;
};
```

**Figure 3.6. SpscValueQueue data structure**

without causing any cache synchronization, because other CPUs don’t access data on the same cacheline. The m_entries pointer points to a dynamically allocated array of queue entries and m_capacity holds the size of that array. Both attributes are initialized once in the constructor of the queue and cannot be changed later (the queue has a fixed size). This is enforced by declaring them const.

The write function depicted in Figure 3.7 demonstrates the enqueuing of a value to the queue. The implementation resembles the original pseudo code from Figure 2.4 very closely. The function takes the element that should be added to the queue as a rvalue reference to avoid unnecessary copies. If loading the hasValue flag reveals that the entry is empty, the function calls constructAt to construct an object of type T at the memory location p and initializes it with value. constructAt itself is a little helper function that either calls placement new to actually create an object by calling the move constructor of T or that does a plain copy of the memory from the source value to the destination buffer p. Which one of the two is called depends on whether type T is trivially copyable or not and is completely determined at compile time, so that no branching occurs at run time. The full implementation of constructAt is depicted in Appendix X.

Analogous to the write function is the read function in Figure 3.8 depicted. This function also resembles the pseudo code from Figure 2.4 very closely. If loading the hasValue flag reveals the presence of a value (line 5), that value is moved to the destination (line 7). After that and before marking the entry as empty, the destructor of the entry value is explicitly called (line 8). This makes sure, that the object is correctly destroyed and releases all resources it eventually holds.
3. Implementation of a SPSC Queue

```cpp
template<typename T>
void constructAt(T&& value, void* p);

bool SpscValueQueue::write(T&& t)
{
    const auto index = m_writeIndex;
    auto& entry = m_entries[index];
    if (!entry.hasValue.load(memory_order_acquire)) {
        constructAt<T>(std::move(t), entry.ptr());
        entry.hasValue.store(true, memory_order_release);
        const auto next = index + 1;
        m_writeIndex = (next != m_capacity) ? next : 0;
        return true;
    }
    return false;
}

bool SpscValueQueue::read(T& value) {
    const auto index = m_readIndex;
    auto& entry = m_entries[index];
    if (entry.hasValue.load(memory_order_acquire)) {
        auto ptr = entry.ptr();
        value = std::move(*ptr);
        ptr->~T();
        entry.hasValue.store(false, memory_order_release);
        const auto next = index + 1;
        m_readIndex = (next != m_capacity) ? next : 0;
        return true;
    }
    return false;
}
```

**Figure 3.7.** SpscValueQueue write access

**Figure 3.8.** SpscValueQueue read access
4.1 Overview

As can be seen on Figure 4.1 TeeTime is based on four core abstraction: Stages, Ports, Pipes, and Configurations.

A stage represents a single processing unit within a Pipe-and-Filter architecture. Most stages read data from an input port, process it and write the result to an output port. Stages can provide an arbitrary number of different input and output ports. Some stages don’t have input ports, because they don’t need any input. They just produce some output and are normally called producer stages and are the first stage in a Pipe-and-Filter architecture. Other stages don’t produce an output, because they are just consuming data and storing it somewhere, those stages are usually called sink stages.

Stages are connected by creating a Pipe between an output port of one stage and the input port of another stage. The pipe is the actual communication channel between the stages and makes sure the output of the first stage is available as input for the second stage.

A Configuration represents a concrete composition of different stages and their connections. An instance if a configuration is responsible for instantiating all required stages and for connecting and finally executing them.

In addition to these core abstractions, the C++ version of TeeTime also features a support layer and its own implementation of a lock-free single-producer single-consumer queue (see Chapter 3 for details).

4.2 The Support Layer

The support layer provides some utilities and tools that are not crucial building blocks of a Pipe-and-Filter architecture, but are still very useful or even required, because the needed functionality is not provided by the C++ standard library.

4.2.1 Platform

Thanks to C++11 a lot of things can be implemented in a platform agnostic way, but unfortunately the C++11 standard library is still pretty limited in what it can do.
4. Implementation of TeeTime for C++

C++11 provide components to work with threads and to implement communication between them, but it does not provide a way to set the CPU affinity of a thread to restrict its execution to certain CPUs.

C++11 also doesn’t provide any functions to access the file system (other than reading and writing files). There is currently no way to list the content of a directory or to delete files. C++17 will have a file system module that supports all these things in a platform agnostic way, but since TeeTime should be compatible with any C++11 compiler, we need to provide the necessary abstractions ourselves. Those abstractions will be used to implement some of the planned ready-to-use stages and test scenarios.

```cpp
namespace teetime {
namespace platform {
  setThreadAffinity(uint64 mask);

  void listFiles(
    const char* directoryPath ,
    std::vector<std::string>& files ,
    bool recursive);

  void listDirectories(
    const char* directoryPath ,
    std::vector<std::string>& directories ,
    bool recursive);

  bool createDirectory(const char* path);
  bool removeDirectory(const char* path);
  bool removeFile(const char* path);
}
}
```

Figure 4.2. Platform Layer

The interface of the platform layer is depicted in Figure 4.2 and consists of a couple of global function to access the file system and to set the thread affinity. The platform layer is implemented for Windows (Win32 API) and Linux (POSIX API).
4.3 Basic Stages

4.2.2 Logging

TeeTime produces a lot of different messages that can be helpful to analyze possible problems and issues. This feature is not only useful for the development of TeeTime itself, but also for applications that use TeeTime. There are a lot of different frameworks available for C++ that provide such a logging functionality (Boost.Log\(^1\), spdlog\(^2\), log4cxx\(^3\), just to name a few), but there is no standardized API and a lot of projects have implemented their own logging framework.

TeeTime does not depend on and therefore does not enforce a particular logging framework. Instead TeeTime itself provides a very simple logging API to set a log level and a log callback. Such a callback can forward all log message to whatever logging framework the application is using.

The API is depicted in Figure 4.3. TeeTime includes a simple and ready-to-use logging function (simpleLogging), that can be used as a callback and that prints all messages to standard out. That function is not meant to be used in actual production code. There is no way to customize formatting or to redirect the output and it is not tuned for optimal performance.

4.3 Basic Stages

TeeTime provides a couple of base classes for stages, that either provide some basic functionality required by all stages or are required to implement certain patterns (like the task farm pattern [Wulf et al. 2016]). The available base classes are depicted on Figure 4.4 and are described in more detail in the following sections.

4.3.1 AbstractStage

The AbstractStage is the common base class for all stages. The class has two pure virtual member functions that must be implemented by derived classes:

createRunnable(): Creates a Runnable to run the stage in its own thread.

execute(): Defines the actual execution of the stage.

Unlike most other base classes and stages, the AbstractStage class doesn’t have any type parameter. In contrast to Java Generics, C++ templates don’t use type erasure, so type information is not lost [Garcia et al. 2003]. The downside is that the type parameter cannot be ignored either. Two instantiations of the same class template, but with different type parameters, are totally distinct types for the type system of C++. There are no raw types in

---

\(^1\)http://www.boost.org/doc/libs/1_62_0/libs/log/
\(^2\)https://github.com/gabime/spdlog
\(^3\)https://logging.apache.org/log4cxx/
C++ and classes don’t always share a common base class like `Object` in Java. For that reason, the `AbstractStage` class explicitly provides such a common base class, that is common to all stage classes, regardless of the types the derived stages might be parameterized with.

The `AbstractStage` base class also manages all input and output ports that are needed by derived stage classes. It provides functions to create input and output ports and to access all available ports. The class also `owns` all ports in terms of memory management, so it makes sure all ports are probably destroyed when the lifetime of a stage ends. The class purposely doesn’t provide any functionality to remove ports from a stage. Being able to remove ports from a stage would complicate the implementation, because it would be necessary to handle a couple of edge cases (e.g. if a port is already connected to some other port, that connection must be invalidated). Since there is currently no use case for removing ports, we omitted it to avoid needless complication.

The `AbstractStage` class also stores a `debug name`, that can be set by derived stages to an arbitrary string value. This name can be used for debugging purposes and in log messages to help identify certain stage instances or to differentiate between stages. Since the name is optional and doesn’t have to be unique, the name should be used for debugging and logging purposes only. It is highly discouraged to use it for anything else and the name
4.3. Basic Stages

Figure 4.4. Basic TeeTime Stages

emphasizes this.

4.3.2 AbstractProducerStage

TODO

4.3.3 AbstractConsumerStage

This class template can be used as a base class for most stages that consume some form of input. The type parameter T specifies the type of data the stage consumes. The class template provides a ready-to-use input port, a `createRunnable` implementation that pulls data from all available input ports, and a `execute` function, that is tailored towards processing a single data element. Derived classes only need to implement the `execute(T&&)` function to consume and process a single data element. The class does not provide any output ports, but derived classes are free to add output ports or to add more input ports.

4.3.4 AbstractFilterStage

This class template is a simple extension of the before mentioned AbstractConsumerStage and models the common case of a stage with exactly one input port and one output port. It has two type parameters: `TIn` and `TOut`. `TIn` is the type of data to be consumed and `TOut`
is the type of data sent to the next stage. The second type parameter is optional and can be omitted. If the second parameter is omitted, it defaults to the same type as the input type (TIn).

### 4.3.5 Distributor

A distributor provides one input port and distributes its input to one or more output ports. The class template provides that input port and functionality to add new output ports, but it doesn’t define how exactly the input is being distributed to these output ports. That’s where the distribution policy comes in. The policy is set as a second type parameter to the Distributor class template. The policy is a callable type that when called with a list of output ports and a value will do the actual distribution.

TeeTime for Java is slightly different in that aspect. The Distributor of the java version makes use of the strategy pattern [Gamma et al. 1995] and takes an instance of a IDistributorStrategy to do the distribution. Policies in C++ are very similar to the strategy pattern [Alexandrescu 2001]. The strategy pattern is usually based on dynamic dispatch, while C++ policies are based on templates and are completely resolved at compile time, saving the cost of the dynamic dispatch. Figure 4.5 gives an impression of how such a policy is implemented. If actual dynamic dispatch at runtime is needed, it can be easily implemented on top of policies, by providing a DynamicPolicy that takes a function pointer or some other kind of interface that can be resolved at runtime.

TeeTime for C++ includes three predefined policies:

- **BlockingRoundRobin**: This policy selects one of the output ports in round robin order and blocks until the data element was successfully sent to that output port.

- **RoundRobin**: This policy cycles through all output ports in round robin order until it successfully sent the data element to an output port. For the next data element, it will continue cycling at the port it left off from the previous one.

- **Copy**: This policy duplicates all incoming data elements for each output port. The equivalent strategy of TeeTime for Java is called CloneStrategy and calls the clone method on an input data element to duplicate it. C++ has no clone function, but the copy constructor fulfills a similar purpose, so that’s why this policy is called Copy. With N output ports, this policy will create N – 1 copies and sent them to the first N – 1 output ports. The last port gets the original input data element, saving one copy this way. The type T must be copy constructable, otherwise the compiler will not compile this stage.

### 4.3.6 Merger

The Merger is the counterpart to the Distributor. It provides just one output port, but an arbitrary amount of input ports can be added to it. All data from all the input ports is forwarded to that one output port, merging the data streams.
4.4 Ready-To-Use Stages

While Section 4.3 describes the basic stages that TeeTime includes and that are required to build more sophisticated stages or implement common patterns, this section introduces the available ready-to-use stages that provide commonly used functionality. Figure 4.6 gives a graphical overview of all these stages.

4.4.1 InitialElementProducer

This stage is a pure producer stage. It doesn’t process any input and has therefore no input port. The constructor takes a list of data elements, that will be sent one by one to it’s only output port when the stage gets executed. The type parameter T specifies the type of data elements.

```
template<typename T> 
class RoundRobinDistribution 
{
public:
    RoundRobinDistribution() : m_next(0) {

    }

    void operator()(const std::vector<unique_ptr<AbstractOutputPort>>& ports,
    T& & value) {
        ...
        }

    private:
        size_t m_next;
    }

    template<typename T, typename TDistributionPolicy = RoundRobinDistribution<T>>
class DistributorStage final : public AbstractConsumerStage<T>
{
public:
    explicit DistributorStage(const char* debugName = "DistributorStage",
    TDistributionPolicy policy = TDistributionPolicy())
    : AbstractConsumerStage<T>(debugName)
    , m_policy(policy) {
    }

    OutputPort<T>& getNewOutputPort() {
        OutputPort<T>* p = AbstractStage::addNewOutputPort<T>();
        return *p;
    }

    private:
        virtual void execute(T&& value) override {
            m_policy(AbstractStage::getOutputPorts(), std::move(value));
        }

        TDistributionPolicy m_policy;
};
```

Figure 4.5. Policy-based Distribution Stage (full implementation of RoundRobinDistribution policy class omitted for brevity)
4. Implementation of TeeTime for C++

4.4.2 CollectorSink

The CollectorSink is a pure consumer stage. It doesn’t produce any output and has therefore no output port. All incoming data elements are collected in an internal container. There are two ways to access these collected elements afterwards: Either by calling `getElements()` or by calling `takeElements()`. The first variant returns a copy of all the collected elements, while the second one moves the elements out of the stage (emptying the stage).

4.4.3 Directory2Files

The Directory2Files takes a directory path as input and sends a file path to the output port for each file in that directory.

4.4.4 File2FileBuffer

The File2FileBuffer takes a file path as input and reads that file from disk into memory. The stage sends a FileBuffer object to its output port, containing the file content and the original file path.
4.4.5 FileExtensionSwitch

This stage redirects file paths from its input port to different output ports, based on their file extension. Files with an unrecognized file extension are either send to a default output port, or are being ignored (if no default output port is configured).

4.4.6 Md5Hashing

The Md5Hashing stage creates a MD5 hash for every incoming data element and sends that to its output port. The type parameter T specifies the input type.

The Java version of TeeTime has a similar stage called MD5Stage. While that stage does the same thing, it has a completely different interface. The Java version only accepts strings as input and produces a string representation of a MD5 hash as output. There are multiple disadvantages of this approach. First of all it is required to create a string representation of the data to be hashed, which can get very costly for a lot of data types (e.g. binary data like images). Second, the stage has to create a string from the actual hash value. Third and last a lot of type-safety is lost, because a string does not necessarily represent a valid MD5 hash.

The C++ version avoids all these issues and is depicted on Figure 4.7. It accepts all types of input (specified by the type parameter T), as long as the global md5hash function is overloaded for that particular type. If such an overload does not exist, the compiler won’t compile the stage for that type.

TeeTime already provides implementations of md5hash for most common data types like int, float, std::string, and others. The type representing the hash value is provided by TeeTime itself and is called Md5Hash. It encapsulates an embedded 128bit value, so it does not dynamically allocate any memory and is relatively cheap to create, copy and compare with other hash values. It also provides convenience functions to create a string representation of it or to parse an existing string representation.

4.4.7 FunctionStage

TBD

4.5 Configuration

Configuration is the base class for all concrete configurations of a Pipe-and-Filter architecture. The public interface of a configuration is very simple and it’s only purpose is to execute the instance of a configuration. Most of it’s functionality is geared towards defining the structure and the behaviour of a Pipe-and-Filter architecture. Configuration provides functions to create stages, to connect stages (or more specifically their ports) and to declare which stages are active. Figure 4.8 depicts such a sample configuration,
4. Implementation of TeeTime for C++

```cpp
class Md5Hash final
|
public:
Md5Hash();
explicit Md5Hash(const uint8 bytes[16]);
Md5Hash(const Md5Hash&) = default;
Md5Hash& operator=(const Md5Hash&) = default;
bool operator==(const Md5Hash& other) const;
bool operator!=(const Md5Hash& other) const;
std::string toHexString() const;

static Md5Hash generate(const void* data, size_t dataSize);
static Md5Hash generate(const std::string& s);
static Md5Hash parseHexString(const std::string& s);
static Md5Hash parseHexString(const char s[32]);

private:
alignas(int) uint8 value[16];

Md5Hash md5hash(int i);
Md5Hash md5hash(float f);
Md5Hash md5hash(const char* s);
Md5Hash md5hash(const std::vector<char>& bytes);

template<
typename T>
class Md5Hashing final : public AbstractFilterStage<T, Md5Hash>
|
public:
explicit Md5Hashing(const char* debugName = "Md5Hashing")
: AbstractFilterStage(debugName)
{};

private:
virtual void execute(T& value) override
|
  getOutputPort().send(md5hash(value));
};
```

**Figure 4.7. Implementation of the Md5Hashing stage**

implemented with TeeTime for C++. For the most part the configuration is identical to the previously mentioned sample configuration with TeeTime for Java (see Figure 2.7). Taking a closer look at it and comparing the two configuration reveals some differences. Despite purely syntactical differences, some of the differences result from the fact that C++ needs manual memory management and other differences are the consequence of deliberate design decisions.

To manage the lifetime of stages, TeeTime makes use of so called *shared pointers*, that are part of the C++11 standard library (*shared_ptr*). A shared pointer manages a (raw) pointer to an object and maintains a reference count for that object. The reference count gets increased with every copy of the shared pointer, and decreased with every shared pointer
4.5. Configuration

```cpp
class MyConfiguration : public Configuration {
public:
  MyConfiguration() {
    this->producer = createStage<ProducerStage<long>>();
    this->processor1 = createStage<FirstProcessorStage<long>>();
    this->processor2 = createStage<SecondProcessorStage<long>>();
    this->sink = createStage<SinkStage<long>>();
    declareStageActive(this->producer);
    declareStageActive(this->processor2);
    connectPorts(this->producer->getOutputPort(), this->processor1->getInputPort());
    connectPorts(this->processor1->getOutputPort(), this->processor2->getInputPort());
    connectPorts(this->processor2->getOutputPort(), this->sink->getInputPort());
  }
private:
  shared_ptr<ProducerStage<long>> producer;
  shared_ptr<FirstProcessorStage<long>> processor1;
  shared_ptr<SecondProcessorStage<long>> processor2;
  shared_ptr<SinkStage<long>> sink;
};
```

Figure 4.8. Example of a TeeTime Pipeline Configuration in C++

that gets destroyed. The object that the smart pointer points to is finally destroyed, when
the reference count drops to zero. The Configuration class provides various functions (like
createStage), that create a stage and return a shared pointer to it. Those factory functions
exist for convenience only. Figure 4.9 depicts some alternative ways to create a stage, that
are all equivalent and valid and could be used instead of the factory function. Only the last
two variants (c and d) are slightly less efficient, because stage object and reference count
are allocated separately [Meyers 2014]. Nothing of this is important to the Java version of
TeeTime, since Java uses a built-in garbage collector and all objects are created by calling
new.

```cpp
auto a = createStage<StageType>(arg1, arg2);
auto b = std::make_shared<StageType>(arg1, arg2);
auto c = std::shared_ptr<StageType>(new StageType(arg1, arg2));
std::shared_ptr<StageType> d;
d.reset(new StageType(arg1, arg2));
```

Figure 4.9. Different ways of instatiating a stage in TeeTime for C++

Another difference is the way stages are being declared active. The AbstractStage
class of the Java version has a method called declareActive() to set the stage active. The
C++ version uses a different approach. The C++ Configuration class provides a function
declareStageActive(Stage) to set a stage active. The information whether a stage is active
or not, is not part of the stage itself, but of the configuration the stage is used in and that
information is also stored in the configuration. This change has two advantages: First of all,
the interface to define the behavior of a configuration is more consistent this way. Ports
4. Implementation of TeeTime for C++

are already beeing connected by calling the `connectPorts` function of the configurations class (or `connectPorts` method in case of Java). Declaring a stage active now follows the same principle, because it’s also done by a function of the configuration class, and not by calling a function on the stage object itself. Like mentioned above, stage instances can also be created by calling the according factory functions of the configuration class.

The second and more important advantage is, that this approach avoids some problematic edge cases. Since in TeeTime for Java the `declareActive()` method is part of the stage class, nothing prevents a stage from declaring itself active. A stage can declare itself active when beeing created (in the constructor) or even while the stage is beeing executed. This doesn’t necessarily lead to fatal errors, but a configuration can behave quite unexpectedly and errors are more likely to happen.

This problem is even more aggravating due to another difference to the Java version. In Java a stage is either active or not active and there is nothing more to that. In C++ an active stage can be restricted to run on certain CPUs only. To do this, the `declareStageActive` function of the configuration takes an optional bit mask to specify on which CPUs that stage can be executed. There is no way for a stage to set this CPU affinity mask to a reasonable value itself, because such a mask value is highly dependent on the configuration the stage is used in and on the available hardware concurrency.

As mentioned before, the `Configuration` class provides some convenience functions to instantiate stages. Among those functions are also functions to create stages from function pointers and lambdas. Examples of this are depicted in Figure 4.10. This allows the creation of extremly light weight stages, because it’s not required anymore to declare a heavy weight class for each type of stage. Especially smaller helper stages can be defined inline as simple lambdas. Those stages are still fully type-safe, because the input and output types are automatically deduced by the factory functions.
4.5. Configuration

```cpp
double half(int i) {
    return i * 0.5;
}

struct ToInt {
    int operator()(const std::string& s) {
        return atoi(s.c_str());
    }
};

class FunctionConfig : public Configuration {
public:
    FunctionConfig() {
        ...
        auto toIntStage = createStageFromLambda.ToInt();
        auto halfStage = createStageFromFunctionPointer(half);
        auto doubleStage = createStageFromLambda([](double d) {
            return d * 2;
        });
        connectPorts(toIntStage.getOutputPort(), halfStage.getInputPort());
        connectPorts(halfStage.getOutputPort(), doubleStage.getInputPort());
        ...
    }
};
```

**Figure 4.10.** Creating lightweight stages from functions and lambdas
Chapter 5

Evaluation of SPSC Queues

5.1 Methodology

To create a meaningful evaluation of our SPSC queue implementation, we test our implementation together with other implementations in multiple scenarios on different systems. The various implementations are described in Section 5.3 and the systems are covered in Section 5.2. While the different scenarios are described in Section 5.4, the methodology behind those scenarios is always the same. Our goal is to measure and compare the throughput of the queue. To achieve that, we create a producer that pushes a fixed number of data elements to the queue. While the producer is running, a consumer tries to pull the same number of data elements from that queue. Producer and consumer are running in parallel and we measure the time it takes for both of them to finish. Figure 5.1 gives an impression of how producer and consumer are implemented. This example pushes only pointers through the queue, but the other scenarios (see Section 5.4) look very similar to that.

```cpp
1 auto produce = [&] () {
2   const size_t local_num = numValues;
3   for (size_t i = 0; i < local_num; ++i) {
4     void* p = reinterpret_cast<void*>(i + 1);
5     auto p = reinterpret_cast<volatile*>(i + 1);
6     while (!queue.write(p)) {
7       std::this_thread::yield();
8     }
9   }
10 }
11
1 auto consume = [&] () {
2   const size_t local_num = numValues;
3   void* tmp;
4   for (size_t i = 0; i < local_num; ++i) {
5     void* tmp;
6     while (!queue.read(tmp)) {
7       std::this_thread::yield();
8     }
9     dest.push_back(tmp);
10   }
11 }
```

Figure 5.1. Producer (left) and Consumer (right) of Queue Benchmark

To reduce the impact of statistical spikes, we repeat the measurement ten times (per scenario, per system and per implementation) and compute the arithmetic mean of it. All queues have a fixed size and are instantiated with the same capacity (1024) for a fair comparison.
5. Evaluation of SPSC Queues

5.2 Environment

On Figure 5.2 an overview of the test systems and their configuration is shown. For our evaluation of a SPSC queue, we use three test systems: *i7-windows*, *i7-linux*, and *Xeon*. The two *i7* based systems have a lot less hardware threads than the *Xeon* system, but are still well suited for testing single-producer single-consumer queues. Since the hardware of *i7-windows* and *i7-linux* are exactly the same, testing on these systems gives us the opportunity to evaluate the influence of different operating systems and compilers on the results.

<table>
<thead>
<tr>
<th></th>
<th>i7-windows</th>
<th>i7-linux</th>
<th>Xeon</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUs</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel Core i7 6700k</td>
<td>Intel Core i7 6700k</td>
<td>Intel Xeon E5-2650</td>
</tr>
<tr>
<td>Architecture</td>
<td>x86_64</td>
<td>x86_64</td>
<td>x86_64</td>
</tr>
<tr>
<td>Clock/Core</td>
<td>4.2GHz</td>
<td>4.2GHz</td>
<td>2.8GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>4</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Threads</td>
<td>8</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>RAM</td>
<td>16GB</td>
<td>16GB</td>
<td>128GB</td>
</tr>
<tr>
<td>Disk</td>
<td>SATA/SSD</td>
<td>SATA/SSD</td>
<td>SATA/HDD</td>
</tr>
<tr>
<td>OS</td>
<td>Windows 10</td>
<td>Ubuntu 16.04</td>
<td>Debian 8</td>
</tr>
<tr>
<td>Compiler</td>
<td>MSVC 2015</td>
<td>GCC 5.2.1</td>
<td>GCC 4.9.2</td>
</tr>
</tbody>
</table>

The *Xeon* system is part of the Software Performance Engineering Lab 1 of the University of Kiel. It’s clockrate is significantly lower than the one of the *i7* systems, but it is especially suited for tests with a lot of parallelism due to it’s two processors with 32 hardware threads. The *Xeon* system is equipped with a lot more RAM, but our tests don’t need that much RAM, so the 16GB of the *i7* systems are not disadvantageous.

5.3 SPSC Implementations

Like mentioned in Section 3.1 there exist several different implementations of SPSC queues. To evaluate our own implementation of such a queue, we need to compare it with those implementations in various different scenarios. While those scenarios are described in Section 5.4, this section describes which implementations (and variants thereof) are actually compared to each other.

*boost::spsc_queue*: SPSC queue implementation from *boost* (version 1.61.0).

*folly::ProducerConsumerQueue*: SPSC queue implementation from *Folly*.
5.4 Scenarios

*folly::ProducerConsumerQueue (cache optimized):* Looking at the implementation of the *Folly* queue revealed an opportunity for optimization. Our optimized *Folly* queue is still a Lamport queue, but making sure read and write indices are on different cachelines could improve performance. The implementation of the *boost* queue makes use of the same optimization and applying that to the *Folly* queue is quite simple, as is shown in Figure 5.3.

*ff::SWSR_Ptr_Buffer:* Default queue implementation of FastFlow (macro `NO_VOLATILE_POINTERS` is not defined). This implementation does not support arbitrary values and can only be used with pointers.

*ff::SWSR_Ptr_Buffer (no volatile):* Alternative queue implementation of FastFlow (macro `NO_VOLATILE_POINTERS` is defined, see Section 2.4 for details on the differences to the default implementation). This implementation does not support arbitrary values and can only be used with pointers as well.

*teetime::SpscValueQueue:* Our implementation of a FastForward queue with support for value semantics. The implementation of this queue is described in Chapter 3.

*teetime::SpscPointerQueue:* A slightly modified version of our SpscValueQueue, that stores only pointers and no arbitrary values. It's a classic FastForward queue that uses a NULL pointer to mark an element empty. It's basically the same queue, as the one from FastFlow, but implemented with modern language features. This implementation does not support arbitrary values and can only be used with pointers.

```cpp
namespace folly {
    template <class T>
    struct ProducerConsumerQueue {
        ...;
        const uint32_t size_;
        T* const records_;
    };
}

namespace folly {
    template <class T>
    struct ProducerConsumerQueue_Optimized {
        ...;
        const uint32_t size_;
        T* const records_;
    };
}
```

**Figure 5.3.** Original folly Queue (left) and Cache Optimized folly Queue (right)

5.4 Scenarios

Since has been stated before, the FastFlow queue supports only pointers and Java implementations of a FastForward queue support only references. In those cases the type
5. Evaluation of SPSC Queues

of the data elements doesn’t matter very much for the performance of the queue. From the perspective of the queue, all pointers (or all references) can be treated the same. For the queue it doesn’t matter what data type a pointer or a reference points to. With value semantics this is a bit different. Data types vary in size, in complexity and operations on those data types cannot be optimized the same way in every case. As a consequence of that, the performance of a queue implementation is highly dependent on the data type the queue is used with.

To take this into account, we will benchmark the queues in different scenarios, each using a different data type. The following scenarios (or data types) are used:

**Pointers:** The producer pushes fifty million pointers to the queue, while the consumer consumes those pointers and writes them to a local buffer. It’s important to note, that those pointers don’t point to actual objects that must be allocated or freed. The pointers are not dereferenced, so this doesn’t matter for the benchmark. The scenario focusses only on pushing pointers through the queue and not on processing actual data. This scenario is the only scenario that can be executed with all implementations of a SPSC queue. The pointer-only implementations by FastFlow and the SpscPointerQueue support pointers naturally, and the other implementations treat them just like regular values.

**`std::vector<int>`:** The producer creates ten million instances of `std::vector<int>` and sends them to the consumer one by one. `std::vector` is a class template of the C++ standard library that manages a dynamically allocated array. In this case, each instance allocated an array of 64 `int` values. `std::vector` supports move semantics, so in the ideal case, the instances should be moved through the queue without any reallocations at all. The goal of this senario is to evaluate the queues with a complex data type, that manages internal resources and profits from move semantics. All instances of `std::vector<int>` are created before the benchmark starts, so the initial allocations are not beeing measured. For the same reason the consumer stores all those instances in a local buffer to avoid any deallocations.

**4x4 matrix:** The producer pushes ten millions 4 × 4 matrices of `double` values to the consumer. The goal of this scenario is to evaluate the queues with a data type that is significantly larger than a primitive data type (so more bytes must be copied) and that does not profit from move semantics. The used matrix data type is shown on Figure 5.4. Each instance of that matrix type is 128 bytes big and must always be fully copied, because those bytes are directly embedded into the matrix object.

```
1 struct Matrix4x4 {
2     double m[16];
3 };
```

**Figure 5.4. 4x4 Matrix**
5.5. Results and Discussion

The benchmark results on the Xeon system are depicted on Figure 5.5.

Looking at the pointers scenario reveals that the boost queue was the slowest of them all, but followed closely by the original folly queue. The optimized folly queue performed significantly better than the original, but not as good as all the FastForward queues. All the FastForward queues (FastFlows queues and our own implementations) are very close together. The NO_VOLATILE version of the FastFlows queue performed slightly better than the non-NO_VOLATILE version, and our own implementations of a FastForward queue are in between the two (our queue with value semantics performs slightly better than the pointer-only version). What stands out from the results of the std::vector<int> scenario is the performance of the boost queue. It’s performance significantly worse than all the other queues. The two Folly queues performs basically the same, while our own FastForward queue with value semantics is slightly ahead of them. This scenario requires value semantics, so there are no results available for queues, that only support pointers. The matrix scenario looks roughly the same as the pointer scenario. The only noticeable difference is the better performance of the boost queue. The boost queue performs very similar to the optimized folly queue and our own queue is slightly ahead of them both.

Putting it all together it is safe to say, that all FastForward queues achieves a higher throughput than Lamport queues in all scenarios. Optimizing the Folly queue by putting the indices on different cachelines results in a pretty big performance win. In the pointers scenario, this puts the folly queue pretty close to the FastForward queues (but not ahead of them). The extremely bad performance of the boost queue in the std::vector<int> scenario can be explained by the missing support for move semantics. The boost queue does not support this, so all std::vector instances are being copied in to and out of the queue, involving a lot of memory allocations and deallocations. This result strongly emphasizes the
importance of move semantics to achieve good performance with complex data types. The optimized folly queue and the boost queue are very close together in the matrix scenario, which is not that surprising, since both are implemented very similarly and use the same cache optimization for their indices.

The benchmark results of the i7 systems are shown on Figure 5.6 and Figure 5.7 and look for the most part very similar to the results from the Xeon system. Two differences are worth being mentioned. First of all, the default FastFlow queue (NO_VOLATILE_POINTERS is not defined) performs significantly worse than all other queues on the Windows system. Second, our own implementations of a FastForward queue are slightly ahead of the ones from FastFlow. The bad performance of the FastFlow queue on Windows can be explained by the way, the Microsoft compiler generates code for volatile variables. According to the
5.6. Threats to Validity

While the results in Section 5.5 are very interesting, there are some threats to validity we have to consider. For one, only two different hardware systems are used for our evaluation. While these two system use different micro architectures, both system are still from the same vendor and implement the same instruction set. The micro architecture of systems from other vendors might differ more radically and could therefore produce different results. Systems that implement a different instruction set and have a different memory model (for example ARM) can produce even more different results.

Another point is the software environment. The i7-windows system and the i7-linux system are based on exactly the same hardware, but produce different results. The reason for that is the used software environment. i7-windows system does not only use a different operating system than the i7-linux system, but our benchmark program is also compiled with a different compiler and uses a different implementation of the C++ standard library. All these things can influence the behaviour and the performance of a SPSC queue implementation.

A third point are the used scenarios or data types. We use three scenarios to cover different characteristics of data types. While all three scenarios give plausible results, we cannot ignore the possibility, that a different data type can lead to different results.
6.1 Methodology

It is important to evaluate how the implementation of TeeTime for C++ compares to similar C++ frameworks and to the TeeTime reference implementation in Java. We consider the throughput and the speedup we get by distributing the workload to multiple threads as the default metric for assessing the performance of Pipe-and-Filter architectures.

The speedup $S$ is defined as $S = \frac{T_1}{T_n}$, where $T_1$ specifies the time it takes to execute the benchmark with just one thread and $T_n$ specifies the time it takes to execute the same benchmark with $n$ threads. In the ideal case the speedup equals the number of used threads $S = \frac{T_1}{T_n} = n$, but usually some speedup is lost due to the required communication between threads. A speedup below one means the program has gotten slower.

![Diagram](image)

**Figure 6.1.** Approach to compare C++ TeeTime with FastFlow

The approach to compare TeeTime for C++ (1) with FastFlow (2) is depicted in Figure 6.1. The implementation of all test scenarios is written in C++ (3), so that it can be shared and executed with TeeTime (4) and FastFlow (5). Both evaluations will generate results (6,7) that can be directly compared (8), because they differ only in the framework being used.
6. Evaluation of TeeTime for C++

Comparing TeeTime for C++ with TeeTime for Java is a lot more complicated, since it is not possible to directly share any code between them. The goal of the comparison between the C++ and the Java version of TeeTime is to find out, whether there are major differences in the way the TeeTime architecture scales on multicore systems. Implementing complex scenarios in Java and C++ dilutes the comparability of the results, because not only the framework being used is different, but also the implementation of the scenario is based on very different languages and libraries. For that reason we restrict the evaluation of TeeTime for Java to very basic scenarios that focus on CPU usage and IO.

To create a meaningful evaluation, we evaluate all configurations in multiple scenarios with different workloads. While all those scenarios are described in Section 6.3, the configurations are described in Section 6.4.

We measure the time it takes for a configuration to finish a scenario. Each scenario is executed for an increasing number of threads. Since the test system supports 32 hardware threads, we execute each scenario with up to 34 threads. Thats two more threads than supported by the hardware, which allows us to see what happens when the number of threads exceeds the available hardware parallelism. For each number of threads, each scenario is executed ten times and we build the arithmetic mean from those ten results to reduce the impact of statistical spikes.

This leads us to 340 executions per scenario and per configuration.

6.2 Environment

For the evaluation we use the Xeon system that is described in the evaluation chapter of the SPSC queue, see Section 5.2 for details.

As can be seen on Figure 5.2, the Xeon system has 2 processors with 16 cores and 32 threads. Some of the used configuration in Section 6.4 assign individual threads to specific (logical) CPUs or hardware threads. To get a better understanding of how this works, the CPU layout of the Xeon system is depicted on Figure 6.2. The red boxes represent the two CPUs of that system. Each green box is a single core and each grey box represents a
6.3 Scenarios

To evaluate the performance of Pipe-and-Filter architectures we need different test scenarios. Since FastFlow does not support the execution of subsequent stages in one thread (see Section 2.4), we have to combine a lot of functionality into single stages that would normally be split into multiple stages.

6.3.1 CPU-intensive Scenario

The first scenario is focussed on creating a CPU-intensive workload. For this purpose a Pipe-and-Filter architecture will be implemented like depicted in Figure 6.3.

The first stage will send a certain number of pre-generated MD5 hash values to the distributor stage. Each hash value has been generated from the same initial integer value. The distributor distributes all incoming hash values to worker stages that try to find that initial integer. They do that by brute force and generate hash values for all integers (starting at zero) and compare each generated hash with the input hash. If those hashes match, the current integer value is send as a result to the next stage. The merger stages merges all result together, so the final sink stage can save them in a local buffer. The larger the initial integer is, the more iterations the worker stages need to find it. By choosing the initial integer we can create different CPU workloads.

This scenario will be tested with four different workloads:

**coarse grained:** This scenario will use 100 as the integer value, so worker stages will need to run 101 iteration to find that value. The resulting workload for each worker stage is about 18µs. The Java version will use 52 as initial integer, resulting in roughly the same workload. One million hashes will be produced and must be reversed by the worker stages.

**medium grained:** This scenario will use 20 as the integer value, so worker stages will need to run 21 iteration to find that value. The resulting workload for each worker stage is about 4µs. The Java version will use 11 as initial integer, resulting in roughly the same workload. Five million hashes will be produced and must be reversed by the worker stages.

**fine grained:** This scenario will use 10 as the integer value, so worker stages will need to run 11 iteration to find that value. The resulting workload for each worker stage is about 2µs. The Java version will use 5 as initial integer, resulting in roughly the same workload. Ten million hashes will be produced and must be reversed by the worker stages.
6. Evaluation of TeeTime for C++

very fine grained: This scenario will use 2 as the integer value, so worker stages will need to run 3 iteration to find that value. The resulting workload for each worker stage is about 500ns. The Java version will use 1 as initial integer. Twenty million hashes will be produced and must be reversed by the worker stages.

Figure 6.3. Scenario 1: CPU-intensive Pipe-and-Filter architecture

6.3.2 IO-intensive Scenario

The goal of the second scenario is to create an I/O-intensive workload by constantly reading and writing files. The architecture for this scenario is depicted in Figure 6.4.

The first stage generates a certain number of predefined integer values. The Distributor stages distribute those numbers to the Write & Read stages. Each Write & Read stage creates a file, and writes a certain number of bytes to that file. The input number specifies how many bytes are written. In a second step, this stage reads that file back into memory and returns the number of bytes read to the next stage before deleting the file. The output number should therefore always equal the input number. The Merger stage merges all input number, so the Sink stage can collect them.

This scenario will be tested with four different workloads:

coarse grained: This scenario will write and read 2000 files, each 10MB big.
medium grained: This scenario will write and read 20000 files, each 1MB big.
fine grained: This scenario will write and read 100000 files, each 128KB big.
very fine grained: This scenario will write and read 1000000 files, each 1KB big.
6.3. Scenarios

6.3.3 Mixed Scenario

The first two scenarios are very artificial and focused on creating a very specific kind of workload. This scenario aims at creating a workload that is similar to the workload of real-world applications. The idea is to generate so-called *mipmaps* for image files. Mipmaps are sequences of images, each of which is a progressively lower resolution representation of the same image. The height and width of each image in the mipmap is a power of two smaller than the previous image. Figure 6.5 illustrates such a mipmap.

![Figure 6.5. Example of a Mipmap](image)

The architecture to achieve this is depicted in Figure 6.3. The *MipMapTask Producer* reads a list of image files from the hard drive. For each image it calculates the number of required mipmap levels. For each level it sends a *MipMapTask* to the next stage. A *MipMapTask* contains a reference to the source image and the mipmap level that should be generated. The Distributor distributes those tasks to the *Resize & Save* stages. A *Resize & Save* stage creates a resized copy of the source image and saves it to disk. The filename of the image
6. Evaluation of TeeTime for C++

file is being sent to the next stage. The Merger stage merges all incoming filenames, so the Sink stage can collect them.

![Diagram](image)

**Figure 6.6. Scenario 3: Realistic Pipe-and-Filter architecture**

This scenario will create a very mixed workload, since the initial *MipMapTask Producer* stage and the *Resize & Save* stages will cause I/O workload when reading and writing files, while decoding, resizing and encoding the images are very CPU-intensive operations.

This scenario will be tested with four different workloads:

- **coarse grained**: This scenario will generate mipmap levels for 20 images, each with a width and height of 1024 pixels. For each image 11 mipmap levels are being created.

- **medium grained**: This scenario will generate mipmap levels for 100 images, each with a width and height of 512 pixels. For each image 10 mipmap levels are being created.

- **fine grained**: This scenario will generate mipmap levels for 200 images, each with a width and height of 256 pixels. For each image 9 mipmap levels are being created.

- **very fine grained**: This scenario will generate mipmap levels for 2000 images, each with a width and height of 64 pixels. For each image 7 mipmap levels are being created.

This scenario is not implemented for the Java version of TeeTime. Loading and processing image files are comparatively complex tasks. The available implementations of this functionality in Java and C++ are so different, that we could not conclude anything useful from the results.
6.4 Configurations

While all three frameworks (TeeTime for C++, TeeTime for Java and FastFlow) provide some basic building blocks and guidance to implement a Pipe-and-Filter architecture, there are still a lot of little details that can affect the performance and can be implemented in various different ways. For a fair comparison of FastFlow and TeeTime it makes sense to test out various configurations to make sure, we don’t choose a configuration that is very bad for a particular framework. Unfortunately FastFlow and TeeTime support very different configuration options, so the different configurations of TeeTime and FastFlow are not directly comparable.

TeeTime for C++ supports different strategies of how to assign threads to CPUs, a feature that is not present in FastFlow and TeeTime for Java.

FastFlow on the other hand, relies on pointers to pass data between different stages, forcing the user to implement some kind of manual memory management. Since TeeTime for C++ use value semantics all memory is preallocated and the user doesn’t need any manual memory management.

All three implementations (TeeTime for C++, TeeTime for Java and FastFlow) use bounded Single-Producer-Single-Consumer queues to communicate between different active stages. The implementations of TeeTime (the existing Java one and our new C++ one) let the user choose an arbitrary size of those queues. FastFlow does not allow this, FastFlow always uses a fixed queue size of 4096. To keep things simple and better comparable all TeeTime configurations (C++ and Java) will use the same queue capacity as FastFlow.

FastFlow doesn’t have explicit Distributor and Merger stages. The FastFlow versions of all scenarios are implemented with the task farm skeleton, which includes a distributor and a merger implicitly [Aldinucci et al. 2010].

6.4.1 C1: TeeTime (no explicit CPU affinity)

This is the simplest TeeTime configuration possible. None of the threads are assigned to a specific CPU, instead the operating system has full control over where the threads are being executed and is free to even migrate threads between different CPUs if it makes sense.

6.4.2 C2: TeeTime (prefer same CPU)

This configuration assigns each active stage to a different (hardware) thread. The strategy is to fully utilize all hardware threads of the first CPU before using the second CPU. In terms of the CPU layout depicted in Figure 6.2, this configuration uses the available hardware threads in the order T0, T1..T31.
6. Evaluation of TeeTime for C++

6.4.3 C3: TeeTime (avoid same core)

This configuration also assigns each active stage to a different (hardware) thread, but in this case the configuration tries to use a dedicated core for each stage as long as possible, before assigning a second stage to a core (and thereby using the second hardware thread of that core). In terms of the CPU layout depicted in Figure 6.2, this configuration uses the available hardware threads in the order $T_0..T_7, T_{16}..T_{23}, T_8..T_{15}, T_{24}..T_{31}$.

6.4.4 C4: FastFlow (multiple allocations)

This configuration is used by most FastFlow tutorials. Each stage frees its input object and allocates a new object for the output. TeeTime for Java basically uses the same approach, but it doesn’t need to explicitly free objects since Java has a garbage collector. The benefit of this approach is, that the functionality of each stage is perfectly encapsulated. Each stage just frees it’s input and doesn’t need to worry about where or when that input was allocated. The disadvantage is that you have a lot of allocations and deallocations. For single data element to traverse a linear pipeline with $n$ stages requires $n - 1$ allocations and $n - 1$ deallocations.

6.4.5 C5: FastFlow (single allocation)

This configuration tries to solve the problem with many allocations and deallocations and is used by most FastFlow demos. The core idea behind this approach is to allocate all memory needed by the pipeline upfront and not by each single stage. The very first stage allocates one big Task object, that holds all data for all following stages. Instead of freeing the input and creating a new output, all stages read and write to that one object. The last stage in the pipeline has to finally deallocate the object. While this reduces to number of allocations and deallocations to 1, it fully destroys the encapsulation and composibility of the stages. The Task object and with it all the stages must be perfectly designed for one very specific Pipe-and-Filter architecture and can not be re-used for others.

6.4.6 C6: TeeTime (Java)

TeeTime for Java doesn’t require any manual memory management, but it doesn’t support different strategies for CPU affinity either. So the configuration of TeeTime for Java is pretty simple and straight-forward. The specialty is, that the capacity of all queues is set to 4096, just like all others.
6.5 Result and Discussion

6.5.1 CPU-intensive Scenario

Figure 6.7 shows the diagram for our CPU-intensive scenario with a coarse workload. With up to six worker threads, all configurations scale nearly the same, with only minor differences. TeeTime based configurations (C1, C2, and C3) are slightly ahead of FastFlow based ones (C4 and C5) and the Java based TeeTime configuration C6 shows first signs of weakness at five worker threads. With more than six worker threads, the speedup of our configurations start to get more diverse. While the TeeTime configuration C3 and both FastFlow configurations (C4 and C5) continue to scale almost linearly (with TeeTime still slightly ahead), the speedup of the TeeTime configuration C2 decreases noticeable. The TeeTime configuration without any CPU affinity (C1) scales a bit better, but still not as good as the C3 configuration. The Java based configuration C6 scales very similar to that, but results are a more fluctuating. Configurations C3, C4, and C5 all show a significant speedup decrease with more than 14 worker threads. Increasing the number of worker thread to 17, results in a speedup decrease for all three of them. With 17 worker thread all configurations are very close together again. When the number of worker thread is increased even more, the configuration C3, C4, and C5 regain their speedup and continue to scale almost linearly. All configurations reach their maximum speedup with 30 worker thread, the only exception is the Java based configuration C6, that reaches it’s maximum with 31 worker threads. For all C++ based configurations, increasing the number of worker threads beyound 30 leads to a dramatic speedup drop. Compared to that, the speedup of the java configuration stays a lot more stable with more than 31 worker threads.

The fact that we reach a maximum speedup larger than sixteen, although the system only has sixteen physical CPUs, is related to the HyperThreading feature of the Xeon CPU. HyperThreading is an implementation of Simultaneous Multithreading [Tullsen et al. 1998] and maximizes the on-chip parallelism.

To understand these results it is important to remember that there are two more threads active in addition to our worker threads. With six worker threads, there are actually eight threads active and that also the number of cores a single CPU in the system has. With more than 8 threads, the C2 configuration does not start to use the cores of the second CPU. The C2 configuration tries to fully utilize a single CPU before taking an additional CPU into account, so it starts to assign a second thread to each core of the first CPU. This is still beneficial and results in a speedup, but that speedup is not as big as with additional actual CPU cores. The C3 configuration does exactly that. It tries to use dedicated cores as long as possible, before assigning a second thread to single core. This explains the better scaling of the C3 configuration compared to C2. The C1 configuration uses no explicit CPU affinity, and it’s results are roughly between C2 and C3 so far. The FastFlow based configuration C4 and C5 seem to use a similar CPU affinity as C3, because they scale very similar (but not as good). With more than 14 worker thread (or 16 threads in total), all physical cores of the system are used and the configurations C3, C4, and C5 are forced to assign a second
6. Evaluation of TeeTime for C++

![Figure 6.7. Speedup of CPU-intensive scenario with a coarse workload granularity](image)

Choosing a more fine granular workload reveals more significant differences between the used configurations. The results of the fine grained workload are shown on Figure 6.8. The TeeTime configurations C1, C2, and C3 scale roughly the same as with the coarse grained workload. The maximum speedup is lower (16-17) and the C1 configuration starts to fall behind a bit earlier, but the overall curvature is the same. The results from the FastFlow based configurations C4 and C5 look very different now. The C4 configuration reaches a maximum speedup of 6 with 8 worker threads. Using more worker threads lets the speedup drop to 4 to 5, and it never recovers from that. The C5 configuration does a lot better and reaches a maximum speedup of 12 with 15 worker threads, but beyond that, speedup decreases continuously with more worker threads. The Java based configuration C6 scales basically the same as with the coarse workload, but it’s maximum speedup is distinctly lower with 14.

The observed changes from the coarse to the fine grained workload become more extrem, when we make the workload even more fine grained. Figure 6.9 shows the results with a very fine grained workload. The C4 configuration doesn’t even reach a speedup of
3, and the C5 configuration reaches only a speedup of five with five worker threads and quickly decreases afterwards. The TeeTime configuration are doing a lot better and the C3 configuration reaches a maximum speedup of seven with 10 worker threads, but starts to decrease with more worker threads as well.

The big differences between the TeeTime and FastFlow configurations are related to the manual memory management the FastFlow configurations have to do. The C5 configuration that does only one allocation and deallocation per data element performs alot better than the C4 configuration that does one allocation and deallocation per data element and per stage. The TeeTime configuration doesn’t even need to do that single allocation and scales therefore even better.

### 6.5.2 I/O-intensive Workload

The results from the coarse grained I/O-intensive workload are depicted on Figure 6.10. As we see, the speedup of the Java based configuration C6 is significantly better than the speedup of all other configurations. The FastFlow configuration C4, C5 and the TeeTime configuration C2 scale nearly same. All three of them suffer from a decreased speedup when the number of worker threads gets bigger than six or seven. The speedup of these configuration reaches values below one, so all these worker threads are making the program run slower. The same effect is visible for C1 and C3 configuration, but the effect is not that strong and the speedup does not drop below one.
6. Evaluation of TeeTime for C++

![Figure 6.9. Speedup of CPU-intensive scenario with a very fine workload granularity](image)

To understand those results better, it is useful to take a look at the actual time values that have been measured. Those values are depicted on Figure 6.11. As can be seen there, with just one worker thread, the Java based configuration C6 takes a lot more time to finish than the C++ ones. This partly explains the better speedup of the C6 configuration that can be seen on Figure 6.10. With up to 6 worker threads the Java based configuration and all the C++ based configurations scale very similar. With more than 6 worker threads the C++ based configurations start to slow down. The FastFlow C4 and C5 configurations and the TeeTime configuration C3 suffers the most from this. The other two TeeTime configurations show the same slowdown, but less distinctive. With more than twelve worker thread, all configurations start to get faster again.

Using a more fine-granular workload reveals a pattern here, as can be seen on Figure 6.12. With more than 6 worker threads the programs gets slower, but as soon as more than twelve worker thread are involved the program gets faster again. This repeats itself when we add more threads. With more than 17 worker threads the program gets slower again, before it suddenly gets faster again with more than 28 worker threads. This pattern cannot be explained with the architecture or the implementation of TeeTime and FastFlow alone. The reasons for the observed behaviour are probably related to the way file operations are implemented in the C++ standard library and in the way the standard library interacts with the operating system. If you look very closely, you see a similar wave-like trend in the runtimes of the Java configuration C6. It’s not clear at this point, if this is related to the issues we observe with the C++ implementations.
6.5. Result and Discussion

Figure 6.10. Speedup of IO-intensive scenario with a coarse workload granularity

Memory management is in this scenario not an issue, since both FastFlow configurations (C4 and C5) scale the same.

Without digging deeper into the way file operations are implemented in the standard libraries of C++ and Java, the only things we can conclude from this scenario is that TeeTime is at least as good as FastFlow. Thanks to the ability to specify a CPU affinity, TeeTime can scale even better than FastFlow in this scenario.

6.5.3 Mixed Workload

Looking at the results from the Mixed workload scenario on Figure 6.13 reveals that TeeTime and FastFlow scale very similar. With ten to fifteen worker threads FastFlow scales better in general and reaches a higher speedup. FastFlow suffers from a speedup drop when switching from 15 to 16 worker threads. Compared to FastFlow TeeTime scales more evenly and its maximum speedup is slightly higher.

When the number of worker threads reaches 11, 22 or 33 the speedup of all configurations drops significantly. With 12 and 23 worker threads this drop is suddenly gone and speedup is even higher than before. The reasons for this remarkable effect is the used round robin strategy to distribute data elements to the worker threads. In this case the size of all input images of the scenario is 1024 by 1024. This means eleven mipmap levels must be generated (1024x1024, 512x512,...,1x1). When the number of worker threads is a multiple of the number of mipmap levels, each worker thread always gets levels of the same size.
6. Evaluation of TeeTime for C++

Since the workload of a level depends on the size of that level, the overall workload is not distributed evenly over all the worker threads. The fact that this effect is exactly the same with every configuration, lets us conclude that FastFlow uses the same round robin distribution as TeeTime.

Looking at results from the same scenario with a finer grained workload (see Figure 6.14), does not result in a significant difference between TeeTime and FastFlow. Either the CPU affinities of TeeTime nor the memory management strategies of FastFlow make a big difference here. All configuration scale roughly the same and reach very similar maximum speedups. The TeeTime configurations seem to be slightly ahead of the FastFlow ones as long as the workload is more coarse grained. With a more fine grained workload, FastFlow takes the lead as can be seen on Figure 6.14 and Figure 6.15.

6.6 Threats to Validity

While we evaluated TeeTime for C++ thoroughly, there are still some threats to the validity of our results. First of all, all threats to validity from for our evaluation of SPSC queues (see Section 5.6) apply to this evaluation as well. In addition to that we did the evaluation only on one system, so running the same benchmark on other systems (that differ in hardware or software) can lead to different results. Another possible threat to validity is that we might not have covered enough use cases. There might still be scenarios with unexpected
6.6. Threats to Validity

Figure 6.12. Runtimes of IO-intensive scenario with a fine grained workload granularity

behaviour due to the way those scenarios are implemented.

With the exception of the IO-intensive scenario, all other scenarios generally conform to our expectations.
6. Evaluation of TeeTime for C++

![Graph](image1.png)

**Figure 6.13.** Speedup of mixed scenario with a coarse workload granularity

![Graph](image2.png)

**Figure 6.14.** Speedup of mixed scenario with a fine workload granularity
6.6. Threats to Validity

Figure 6.15. Speedup of mixed scenario with a very fine workload granularity
Both TeeTime (for Java) and FastFlow have been the subject of various research projects to experiment with new approaches to utilize the available parallelism of modern computer systems. There exist an extension for FastFlow to support distributed systems [Aldinucci et al. 2013]. This extension targets clusters of multicore systems by splitting up the FastFlow framework into two tiers:

- a lower tier to implement patterns like Pipeline and Task Farm on a shared-memory multicore workstation.
- a upper tier to coordinate a set of distributed nodes, each executing computations implemented with the lower tier.

The lower tier is essentially what currently exists in FastFlow and what we use to implement our scenarios for the evaluation. The upper tier adds special stages that have an external channel (or Pipe). This external channel is used to send data elements over the network to other nodes of the cluster.

For the purpose of this thesis, we implemented the Task farm pattern manually by creating a distributor stage and a merger stage. FastFlow provides a so called skeleton for that to make the initial setup easier. This skeleton take a producer stage, a list of worker stages and a collector stage and connects them accordingly. This could be implemented for TeeTime as well, so the user doesn’t need to create the distributor and the merger and connect all those stages manually. The existing reference implementation of TeeTime for Java even takes a step further and provides a self-adapting stage for task farming [Wulf et al. 2016]. The idea is to provide a composite stage that wraps an existing stage to increase it’s throughput. This composite stage automatically adapts the number of instances of that existing stage to achieve the highest possible throughput.

As can be seen in our evaluation of the mixed scenario (see Section 6.5.3), certain scenarios can lead to a very unfavourable distribution of workload to the worker stages. TeeTime already has support for specifying a different distribution strategy to overcome this problem, but a different and probably more general applicable approach to this is work stealing [Sanchez et al. 2011; Asenjo et al. 2009]. The idea is to let worker stages steal input data from other worker stages when they run out of work. This way these stages could help out a stage, that is becoming the bottleneck.
Chapter 8

Conclusions and Future Work

8.1 Conclusion

This thesis presents an approach to implement the TeeTime Pipe-and-Filter framework for C++. At first we discussed the required foundations to clarify what Pipe-and-Filter architectures actually are and what the important design decision of TeeTime are. In addition to that we also described in what ways the C++ programming language differs from Java and how these things must be supported by a framework like TeeTime to be usable in the context of typical C++ programs.

In Chapter 3 we described how to implement a lockfree Single Producer Single Consumer queue, before we described in Chapter 4 the actual implementation of TeeTime for C++.

In Chapter 5 we evaluated our implementation of a SPSC queue together with other implementations and came to the conclusion that our implementation is suited best for our needs. In Chapter 6 we finally evaluated TeeTime itself in various different scenarios to compare it with FastFlow and with the reference implementation of TeeTime for Java. In that evaluation we not only showed that TeeTime can be implemented for C++, but also that this implementation is able to achieve good performance gains. These performance gains are very similar to what an established framework like FastFlow achieves and in some cases, the performance gains are even better. The reason for that is that TeeTime for C++ supports a more modern and value based usage of the C++ programming language and therefore doesn’t need manual memory management. A side effect of this is that TeeTime allows a much more modular and self-contained design of stages, that are more reusable and improve the overall design of programs.

8.2 Future Work

One of the big advantages of the C++ programming language is it’s wide availability on nearly all platforms. Like has been stated in the threats to validity sections (see Section 5.6 and Section 6.6) this version of TeeTime was only tested and evaluated on one system. Extending that to other systems and platform is an obvious thing to make TeeTime for C++ available on more platforms.

In the context of this thesis we implemented only the most basic functionality of TeeTime.
8. Conclusions and Future Work

The reference implementation of TeeTime for Java implements a lot more functionality and even more sophisticated features to utilize parallelism and to distribute workload. Those features can be implemented for C++ as well.
Appendix A

Detailed Evaluation Diagrams

![Diagram 1: MD5 Benchmark, 20µs Speedup vs Number of worker threads]

![Diagram 2: MD5 Benchmark, 20µs Time vs Number of worker threads]
A. Detailed Evaluation Diagrams
MD5 Benchmark, 1600ns

- Speedup vs. Number of worker threads
- C1: TeetTime (no affinity)
- C2: TeetTime (prefer same CPU)
- C3: TeetTime (avoid same core)
- C4: FastFlow (multi alloc)
- C5: FastFlow (single alloc)
- C6: TeetTime (Java)

MD5 Benchmark, 1600ns

- Time (milliseconds) vs. Number of worker threads
- C1: TeetTime (no affinity)
- C2: TeetTime (prefer same CPU)
- C3: TeetTime (avoid same core)
- C4: FastFlow (multi alloc)
- C5: FastFlow (single alloc)
- C6: TeetTime (Java)
A. Detailed Evaluation Diagrams

![Graph 1: MD5 Benchmark, 500ns Speedup](image1)

![Graph 2: MD5 Benchmark, 500ns Time](image2)
A. Detailed Evaluation Diagrams

![IO Benchmark, 20000 * 1MB Speedup Diagram](image)

![IO Benchmark, 20000 * 1MB Time (milliseconds) Diagram](image)
IO Benchmark, 100000 * 128kB

- C1: TeeTime (no affinity)
- C2: TeeTime (prefer same CPU)
- C3: TeeTime (avoid same core)
- C4: FastFlow (multi alloc)
- C5: FastFlow (single alloc)
- C6: TeeTime (Java)

Speedup vs. Number of worker threads

IO Benchmark, 100000 * 128kB

- C1: TeeTime (no affinity)
- C2: TeeTime (prefer same CPU)
- C3: TeeTime (avoid same core)
- C4: FastFlow (multi alloc)
- C5: FastFlow (single alloc)
- C6: TeeTime (Java)

Time (milliseconds) vs. Number of worker threads
A. Detailed Evaluation Diagrams

IO Benchmark, 1000000 * 1kB

- C1: TeeTime (no affinity)
- C2: TeeTime (prefer same CPU)
- C3: TeeTime (avoid same core)
- C4: FastFlow (multi alloc)
- C5: FastFlow (single alloc)
- C6: TeeTime (Java)

Speedup vs. Number of worker threads

IO Benchmark, 1000000 * 1kB

- C1: TeeTime (no affinity)
- C2: TeeTime (prefer same CPU)
- C3: TeeTime (avoid same core)
- C4: FastFlow (multi alloc)
- C5: FastFlow (single alloc)
- C6: TeeTime (Java)

Time (milliseconds) vs. Number of worker threads
Mipmaps Benchmark, 20 * 1024x1024

Number of worker threads

Mipmaps Benchmark, 20 * 1024x1024

Number of worker threads
A. Detailed Evaluation Diagrams

![Graph 1](image1)

![Graph 2](image2)
A. Detailed Evaluation Diagrams

Mipmaps Benchmark, 2000 * 64x64

C1: TeeTime (no affinity)
C2: TeeTime (prefer same CPU)
C3: TeeTime (avoid same core)
C4: FastFlow (multi alloc)
C5: FastFlow (single alloc)

Mipmaps Benchmark, 2000 * 64x64

C1: TeeTime (no affinity)
C2: TeeTime (prefer same CPU)
C3: TeeTime (avoid same core)
C4: FastFlow (multi alloc)
C5: FastFlow (single alloc)
Bibliography


Bibliography


