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

Johannes Ohlemacher

November 23, 2016
1. Motivation

2. Goals

3. Foundations

4. Design and Evaluation of a SPSC Queue

5. Design and Evaluation of TeeTime for C++

6. Conclusions and Future Work
Motivation

TeeTime

- high performance, low overhead Pipe-and-Filter (P&F) Framework
- flexible and extensible
- architecture designed to be implementable in other languages
Motivation

► TeeTime
  ► high performance, low overhead Pipe-and-Filter (P&F) Framework
  ► flexibel and extensible
  ► architecture designed to be implementable in other languages

► Why C++?
  ► How does the language influence the performance of TeeTime?
  ► Compare TeeTime with an established C++ solution (FastFlow)
Goals

- G1: Basic Pipe & Filter Setup
- G2: Ready-To-Use Stages
- G3: Support for Multiple Platforms
- G4: Comparison with FastFlow
- G5: Comparison with Java-based TeeTime
The Pipe-and-Filter Architectural Style

Foundations

- The Pipe-and-Filter Architectural Style (Christian Wulf, 2016)

- The Pipe-and-Filter Framework TeeTime (Wulf, Ehmke, and Hasselbring, 2014)

TeeTime

- The C++11 Programming Language (Meyers, 2014; Stroustrup, 2013)

- The FastFlow Programming Framework (M. Aldinucci et al., 2010)
The C++11 Programming Language

Foundations

- Native code
- Manual Memory Management
- Value Semantics
- Move Semantics
Parallel Programming Framework

- Ready-to-use skeletons: *Pipe* and *Taskfarm*
- All Stages are active
- No value semantics
The FastFlow Programming Framework

Foundations

Integer Producer

allocate & write

Integer, String

read & write

Integer to String

read & free

String Sink

Pointer
Overview of existing SPSC queue implementations

Design and Evaluation of a SPSC Queue

- Lamport: (Lamport, 1983)
- FastForward: (Marco Aldinucci, Danelutto, et al., 2012; Giacomoni, Moseley, and Vachharajani, 2008)

---

1https://github.com/facebook/folly
2http://www.boost.org
Overview of existing SPSC queue implementations

Design and Evaluation of a SPSC Queue

- Lamport: (Lamport, 1983)
- FastForward: (Marco Aldinucci, Danelutto, et al., 2012; Giacomoni, Moseley, and Vachharajani, 2008)

<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>folly¹</td>
<td>Lamport</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>boost²</td>
<td>Lamport</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>FastFlow</td>
<td>FastForward</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

¹https://github.com/facebook/folly
²http://www.boost.org
Pointer-based FastForward Queue

Design and Evaluation of a SPSC Queue

Queue:

0x01 | 0x08 | 0x20 | 0x00 | 0x08

| value1 | value4 | value3 | value2 |

0x01: 0x08: 0x20: 0x45:
| true | value1 | true | value2 | true | value3 | false | true | value4 |

Queue:
Evaluation of SPSC queues

Design and Evaluation of a SPSC Queue

Test Environment

1. 2x Intel Xeon E5-2650, 2.8GHz, 8 cores, 128GB, Debian, gcc 4.9.2
2. 1x Intel i7 6700K, 4.2GHz, 4 cores, 16GB, Ubuntu, gcc 5.2.1
3. 1x Intel i7 6700K, 4.2GHz, 4 cores, 16GB, Windows 10, MSVC 2015

Test Scenarios

- Scenario 1: pointers
- Scenario 2: std::vector<int>
- Scenario 3: 4x4 double matrix
Evaluation of SPSC queues

Design and Evaluation of a SPSC Queue

- **Pointers**
  - folly::ProducerConsumerQueue
  - folly::ProducerConsumerQueue (optimized)
  - boost::spsc_queue

- **std::vector<int>**
  - ff::SWSR_Ptr_Buffer
  - ff::SWSR_Ptr_Buffer (no volatile)
  - teetime::SpscValueQueue
  - teetime::SpscPointerQueue

- **4x4 matrices**
  - Time (milliseconds)

Johannes Ohlemacher
Conception and Development of a Pipe & Filter Framework for C++
November 23, 2016
Overview

Design and Evaluation of TeeTime for C++

AbstractStage

+debugName
+execute()
+createRunnable()

#addOutputPort(): OutputPort<T>
#addInputPort(): InputPort<T>

AbstractOutputPort

OutputPort<T>
+send(value: T)

AbstractInputPort

InputPort<T>
+receive(): T

Pipe<T>

UnsynchedQueue<T>

SynchedPipe<T, TQueue>

UnsynchedQueue<T>

Configuration

#declareStageActive(stage, cpuAffinity)
#declareStageNonActive(stage)
#connectPorts(outputport, inputport, capacity)
#createStage<T>(args...): Stage
#createStageFromFunctionPtr(FunctionPtr): Stage
#createStageFromLambda(lambda): Stage
+executeBlocking()
```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;
};
```
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()
    {
        ...
        autotoIntStage = createStageFromLambda(ToInt());
        auto halfStage = createStageFromFunctionPointer(half);
        auto doubleStage = createStageFromLambda([](double d) {
            return d * 2;
        });
        connectPorts(toIntStage->getOutputPort(), halfStage->getInputPort());
        connectPorts(halfStage->getOutputPort(), doubleStage->getInputPort());
        ...
    }
};
Evaluation of TeeTime for C++

Test Environment

1. 2x Intel Xeon E5-2650, 2.8GHz, 8 cores, 128GB, Debian, gcc 4.9.2

Test Scenarios

- Scenario 1: CPU intensive
- Scenario 2: IO intensive
- Scenario 3: CPU and IO intensive

Test Configurations

- 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)
CPU intensive Scenario

Design and Evaluation of TeeTime for C++
Design and Evaluation of TeeTime for C++

**IO intensive Scenario**

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

November 23, 2016 20 / 38
CPU and IO intensive Scenario

Design and Evaluation of TeeTime for C++
Overview

Design and Evaluation of TeeTime for C++

MD5 Benchmark, 20µs

- Ideal
- 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)

Number of worker threads vs. Speedup
Overview

Design and Evaluation of TeeTime for C++

MD5 Benchmark, 500ns

- Ideal
- 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)

Number of worker threads vs. Speedup
Design and Evaluation of TeeTime for C++

IO Benchmark, 20000 * 1MB

- Ideal
- 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)

Number of worker threads vs. Speedup
**Overview**

Design and Evaluation of TeeTime for C++

**IO Benchmark, 20000 * 1MB**

- 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)
Overview

Design and Evaluation of TeeTime for C++

Mipmaps Benchmark, 100 * 512x512

Number of worker threads

Speedup

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

Conclusions and Future Work

- TeeTime successfully ported to C++
- TeeTime for C++ supports modern use of the C++ programming language
- Implemented a Value-based FastForward SPSC queue
- Speedup very similar to FastFlow
  - in many cases even better due to less memory management
  - TeeTime supports better modularity and stages are more reusable than with FastFlow
- TeeTime for C++ added features currently not available for the Java version (CPU affinity, Lambdas)
Future Work

Conclusions and Future Work

- Extent evaluation on more diverse platforms
- Adopt C++14 and C++17
- Adaptive Taskfarm Pattern (Wulf, Wiechmann, and Hasselbring, 2016)
- Targeting distributed systems (Marco Aldinucci, Campa, et al., 2013)


```c++
struct Array {
    Array(int size)
        : elements(new int[size]), size(size) {
    }

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

    int* elements;
    int size;
};
```
```cpp
Array(const Array& a) {
    this->elements = new int[a.size];
    this->size = a.size;
    for (int i = 0; i < this->size; ++i) {
        this->elements[i] = a.elements[i];
    }
}
```
```cpp
Array(Array&& a) {
    this->elements = a.elements;
    this->size = a.size;
    a.elements = nullptr;
    a.size = 0;
}
```
class String2Int : public ff_node_t<std::string, int> {
public:
    virtual int* svc(std::string* data) override {
        int i = atoi(data->c_str());
        delete data;
        return new int(i);
    }
};

String2Int stringToInt;
ff_node_F<char, std::string> producer(f1);
ff_node_F<int, char> sink(f2);
ff_Pipe <> pipe(producer, stringToInt, sink);
pipe.run_and_wait_end();
value-based FastForward Queue

Appendix

Johannes Ohlemacher

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

November 23, 2016 37 / 38

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

    T* ptr () {
        return reinterpret_cast<T*>(&buffer[0]);
    }
};
```cpp
bool enqueue(data) {
    assert(data != NULL);
    if(buffer[head] == NULL) {
        buffer[head] = data;
        head = NEXT(head);
        return true;
    }
    return false;
}

bool dequeue(data) {
    if(buffer[tail] != NULL) {
        data = buffer[tail];
        buffer[tail] = NULL;
        tail = NEXT(tail);
        return true;
    }
    return false;
}
```