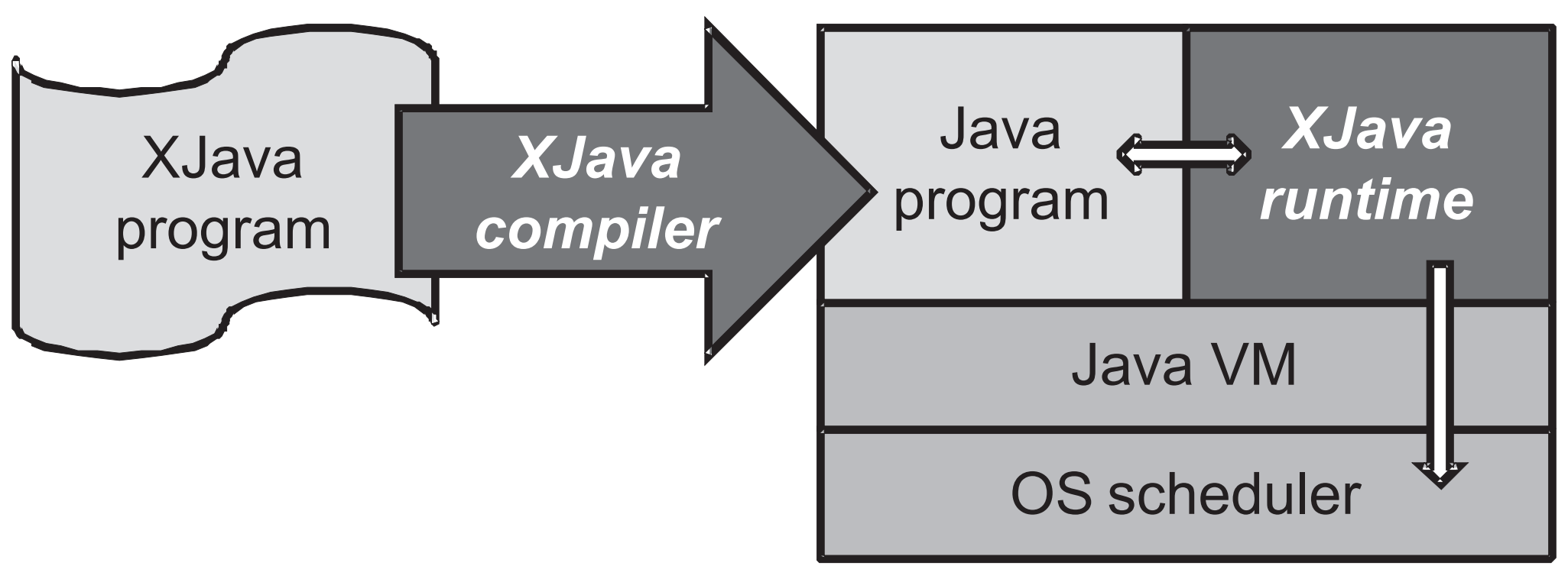
XJava

High-level Multicore Programming

XJava

- Language extension of Java
- Parallel programming without explicit threading
- **Goal:** „what you see is what you get“ parallelism
- **Idea:** unify object-orientation, stream programming, and parallel patterns
- ***XJava compiler:*** XJava to Java (or bytecode)
- ***XJava runtime:*** task pool, executor service, and scheduler



XJava Language Design

- **Tasks**
  - specialized methods that are concurrently executable
  - declared in classes or interfaces
  - inherit or override other tasks
  - can be private, static, abstract, final, …
  - **typed input and output ports** for receiving and generating **streams of data**

```
public Object => String foo() {
  /* task body */
}
```

  - **Periodic tasks** define exactly one **`work`** block for repeated execution
  - **Non-periodic tasks** do not define a **`work`** block, but may contain parallel expressions for introducing nested parallelism
  - a **push** statement puts an object to the output stream

- Combine tasks through operators to **parallel expressions**:
  - „=>“ creates a **pipe expression** (e.g. pipelines, master/worker, producer/consumer)
  - „|||“ creates a **concurrent expression**, i.e. makes independent tasks run concurrently (e.g. task or data parallelism)
  - **type safety:** compiler checks if combined tasks (i.e. input and output types) „fit together“

A simple pipeline example

```
public void => Block read(File f) {
  Iterator i = f.blockIterator();
  while(i.hasNext()) { push (Block) i.next(); }
}

public Block => Block compress() {
  work(Block b) { push b.compressBlock (); }
}

public Block => void write(File f) {
  work(Block b) { f.add(b); /* no push */ }
}
```

```
read(inFile) => compress() => write(outFile);
```

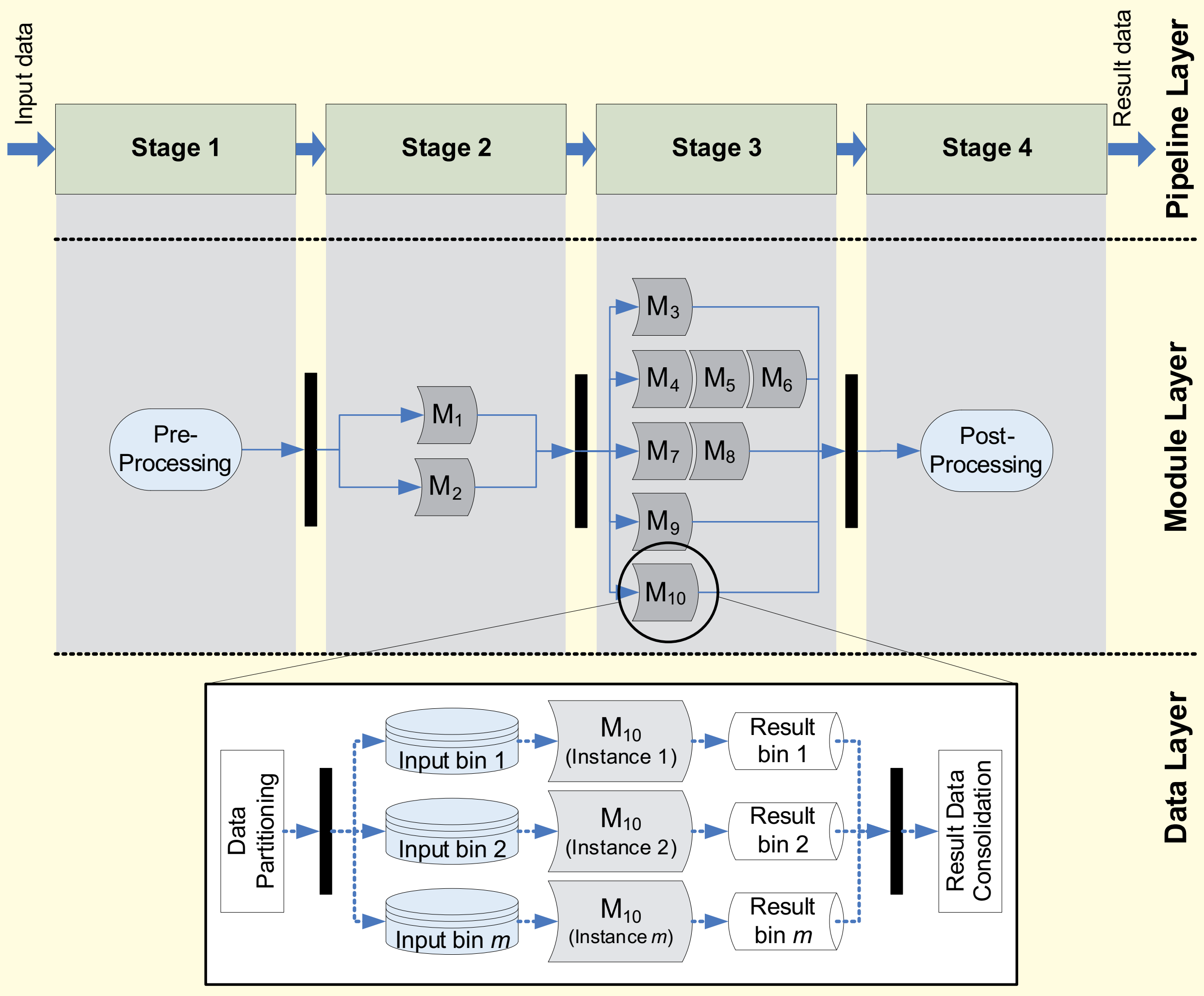
Tasks are declared like methods; instead of a return type, they have input and output types

A work block is repeatedly executed until all elements are processed

„=>“ connects tasks and generates parallelism

- **`read(File f)`** reads a file **`f`**, divides it into **`Blocks`**, and puts them (in form of a data stream) on the output
- **`compress()`** expects a stream of **`Blocks`** and produces a stream of compressed **`Blocks`**
- **`write(File f)`** expects a stream of **`Blocks`** and stores them to a file **`f`**
- The pipeline expression (1 LOC!) creates the whole parallelized file compression
  - similar to Unix filters
- The programmer does not need to care about synchronization

A more complex example: a real-world parallel application (biological data analysis)



Different layers of abstraction for nested parallelism: pipeline, task and data parallelism

Split-join data flow

Implementing this in common threaded code is painful

Skeleton of this architecture in XJava code:

```
void => X stage1() {...}
X => Y stage2() {m1() ||| m2(); }
Y => Z stage3() {m3() ||| m456() ||| m78() ||| m9() ||| m10(); }
Z => void stage4() {...}
...
Y => Z m456() {m4() => m5() => m6(); }
Y => Z m78() {m7() => m8(); }
...
X => Y m1() {...}
...
Y => Z m10() {m101() ||| m102() ||| m103(); }
```

```
stage1() =>* stage2() =>* stage3() => stage4();
```

Preliminary Results

- Benchmark programs
  - open-source desktop search application
  - several smaller programs: text transformation, sorting algorithms, etc.
- Code savings **up to 40%** over threaded Java
- Good speedups over sequential Java (e.g. **between 2 and 3.5** on a quadcore, **up to 31.5** on a Niagara2)

Improvements for Software Engineering

- Write parallel general-purpose applications in a „what you see is what you get“ style
  - Better code understanding, „less indeterminism“
- Performance gains
  - Exploit object-oriented parallelism on all fronts
- Abstraction
  - Hide confusing details wherever it is possible
- Fewer bugs & easier debugging
  - Intuitive language constructs & implicit parallelism: less error-prone
  - Compiler/debugger: more knowledge about semantics
- Code savings & higher productivity

Frank Otto, Victor Pankratius, Walter F. Tichy
Institute for Program Structures and Data Organization / University of Karlsruhe, Germany
Contact: Frank Otto (otto@ipd.uka.de)

Date: May 2009