

Towards Implicit Parallel Programming for Systems

Sebastian Ertel

Dr.-Ing. Defense
Chair for Compiler Construction
TU Dresden, Germany

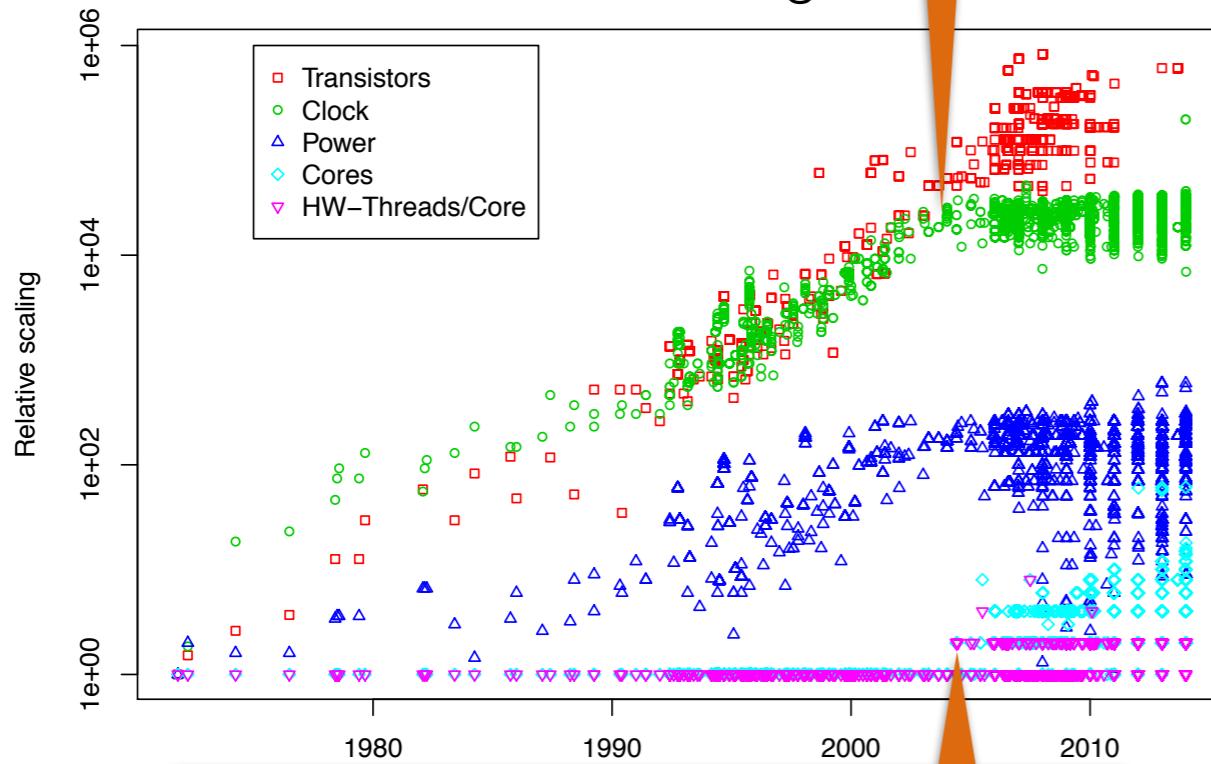
Supervisor: Prof. Dr.-Ing. Jeronimo Castrillon
Fachreferent : Prof. Dr.-Ing. Hermann Härtig

4.9.2019

The Shape Of Things To Run

The end of the single-core era

Processor scaling trends



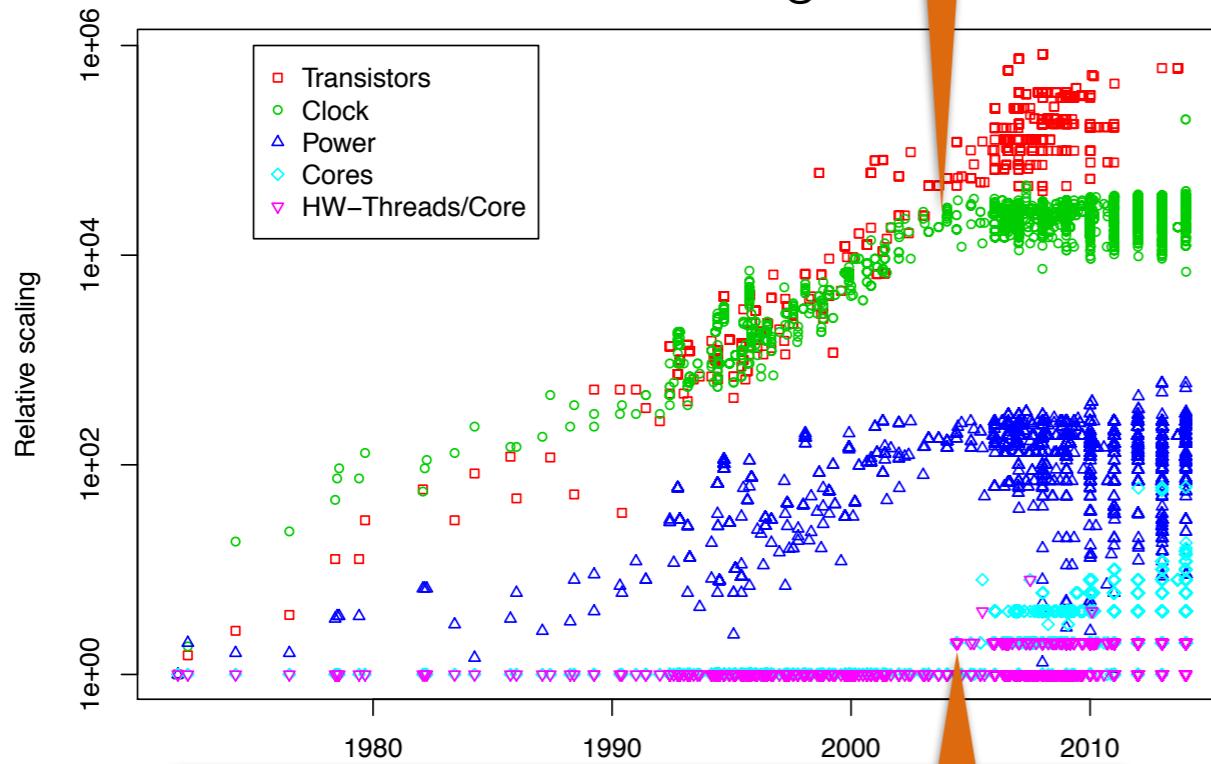
The dawn of the multi-core era

Herb Sutter. 2005. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobbs Journal.

The Shape Of Things To Run

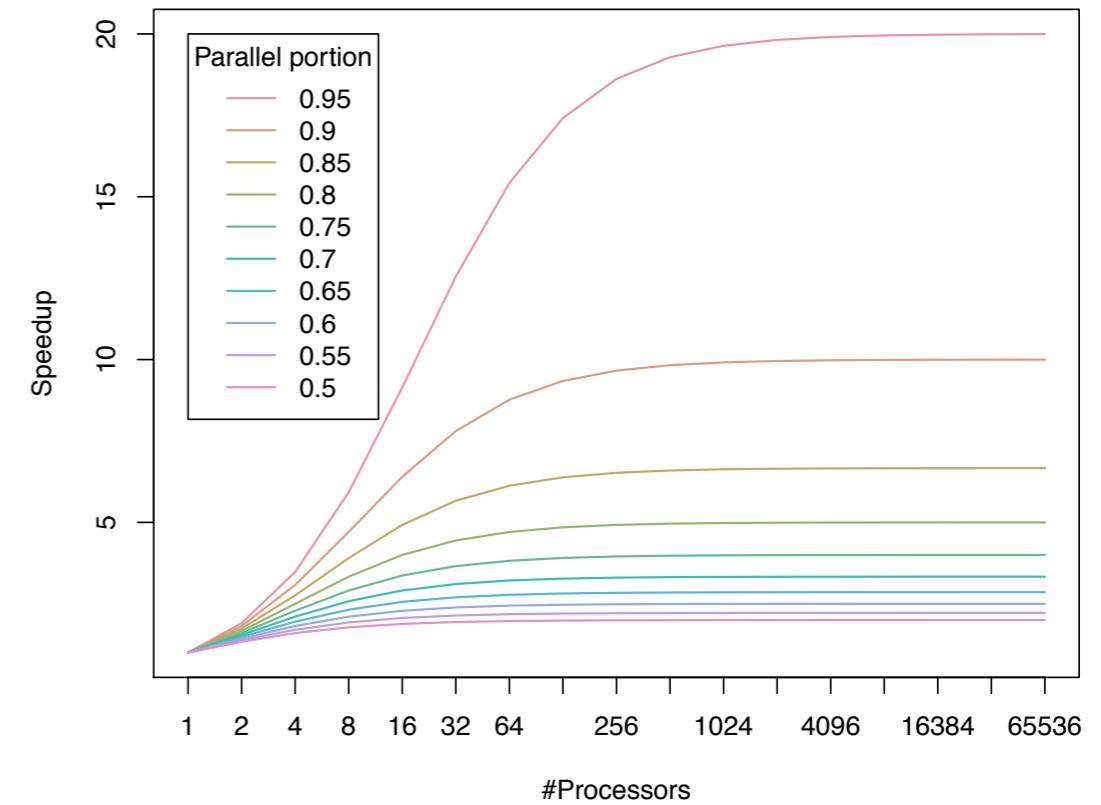
The end of the single-core era

Processor scaling trends



The dawn of the multi-core era

Amdahl's Law



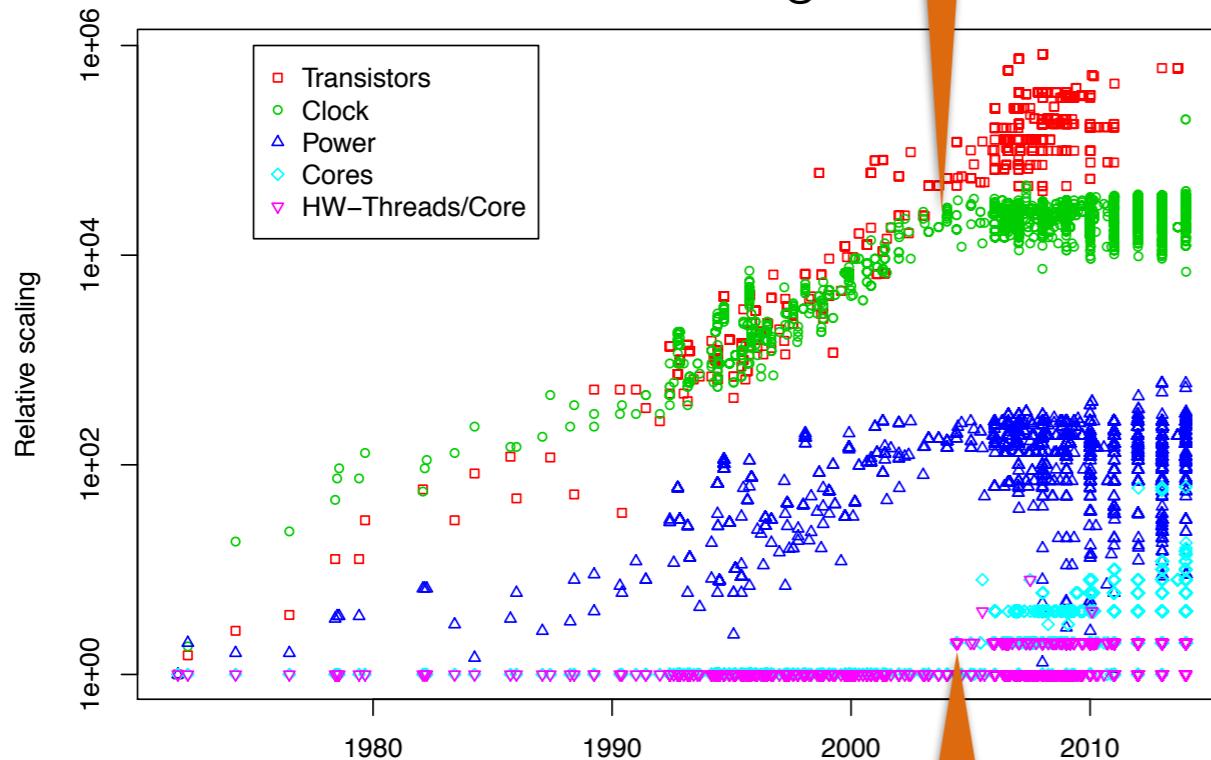
Herb Sutter. 2005. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobbs Journal.

G. M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. AFIPS '67. ACM.

The Shape Of Things To Run

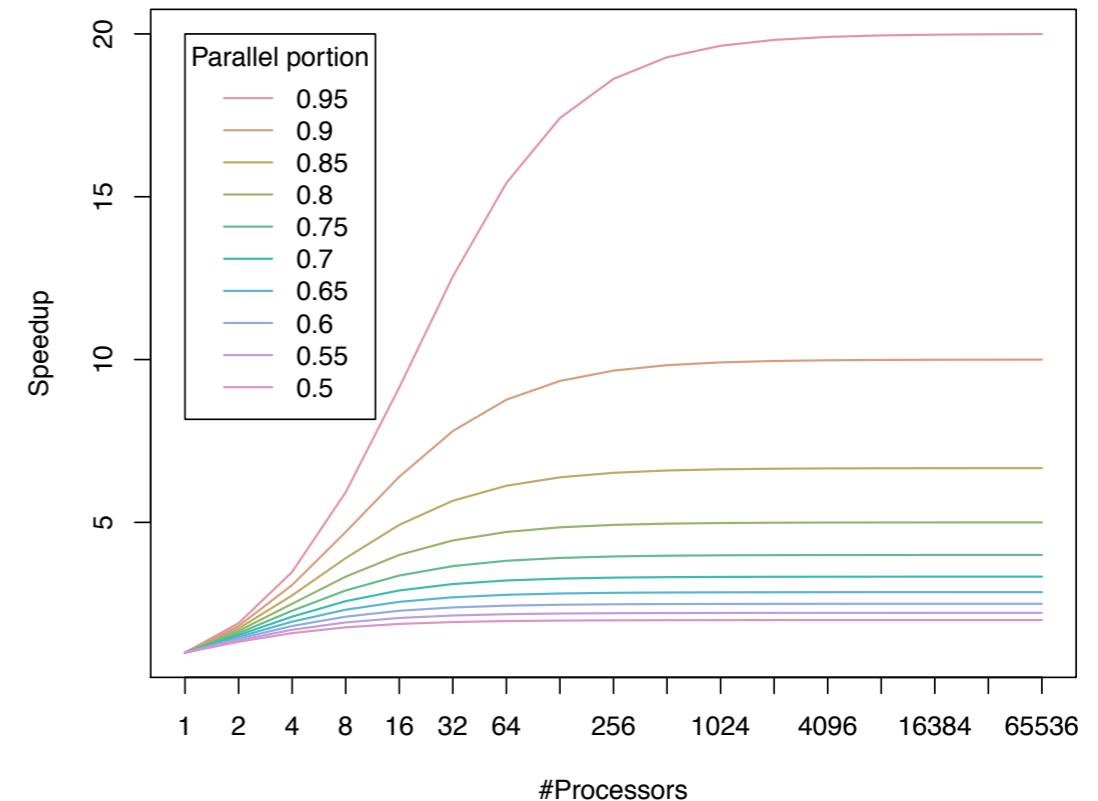
The end of the single-core era

Processor scaling trends



The dawn of the multi-core era

Amdahl's Law



Challenge	Near-Term	Long-Term
1,000-fold software parallelism	Data parallel languages and "mapping" of operators, library and tool-based approaches	New high-level languages, compositional and deterministic frameworks

Herb Sutter. 2005. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobbs Journal.

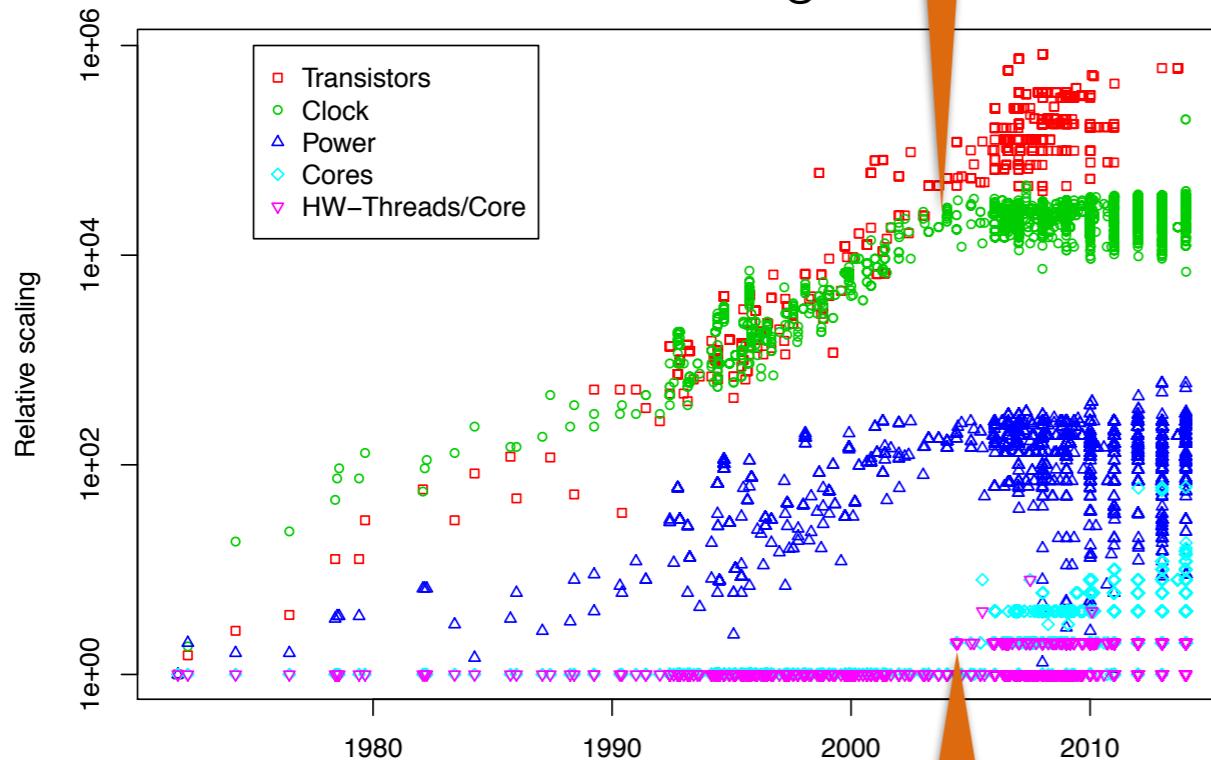
G. M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. AFIPS '67. ACM.

Shekhar Borkar and Andrew A. Chien. 2011. The future of microprocessors. Commun. ACM.

The Shape Of Things To Run

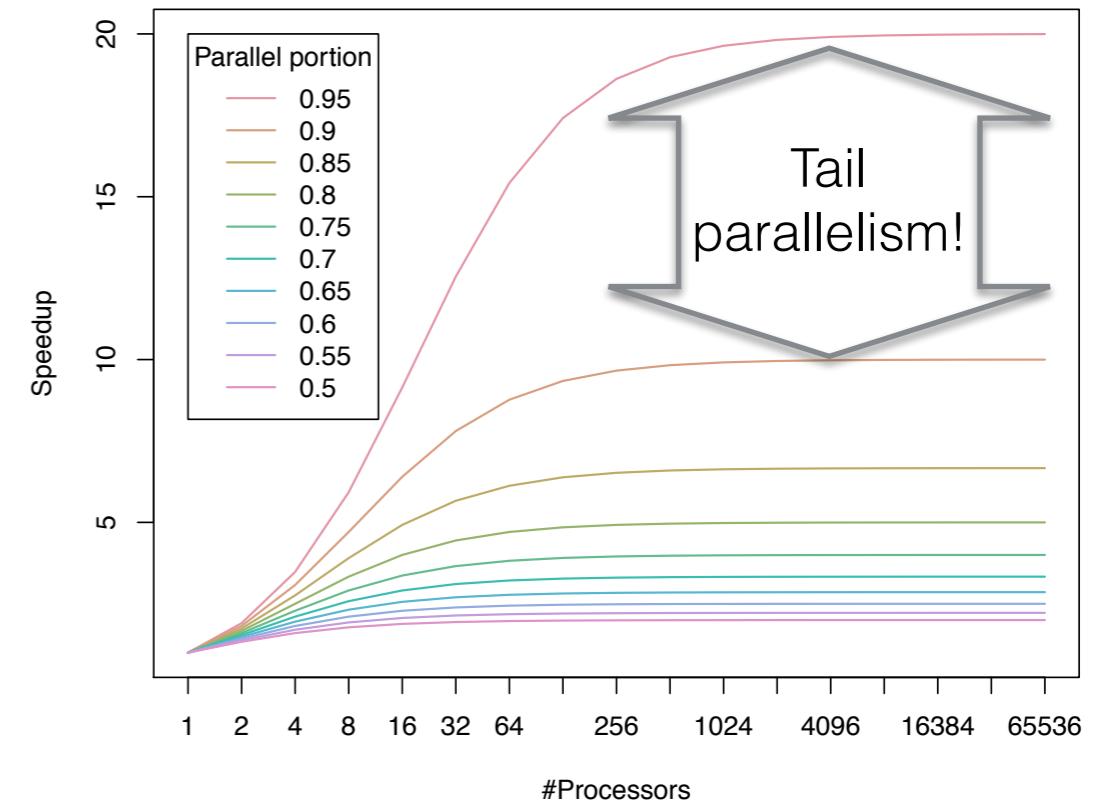
The end of the single-core era

Processor scaling trends



The dawn of the multi-core era

Amdahl's Law



Challenge	Near-Term	Long-Term
1,000-fold software parallelism	Data parallel languages and "mapping" of operators, library and tool-based approaches	New high-level languages, compositional and deterministic frameworks

Herb Sutter. 2005. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobbs Journal.

G. M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. AFIPS '67. ACM.

Shekhar Borkar and Andrew A. Chien. 2011. The future of microprocessors. Commun. ACM.

Programming Is Simple!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods

Parallel Programming Is Hard!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods



Concurrent/Parallel Programming

Transactional
Memory
Locks
Message-
Passing
Processes

compare-
and-swap
Threads

Operating System

LVars
MVars
IVars

Fork/Join

Tasks

Work-Stealing

Coroutines

Spawn

Futures

Events

Actors
Fibers

Events

OpenMP (loops)

Map/Reduce

Custom/optimized

SQL
Parallel
Haskell (pH)
Reactive
Programming
Streams
Dataflow

Low level

High level

Parallel Programming Is Hard!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods



Concurrent/Parallel Programming

Transactional
Memory
Locks
Message-
Passing
Processes

compa
and-sw

LVars
MVars

Threads

Tasks

Futures

Events

Map/Reduce

Algorithmic
Composition

Composition and state are key for systems!

Work-Stealing

Custom/optimized

Low level

High level

SQL

Parallel
Haskell (pH)

Reactive
Programming

Streams

Dataflow

Parallel Programming Is Hard!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods



Concurrent/Parallel Programming

Transactional
Memory
Locks
Message-
Passing
Processes

compare-
and-swap
Threads

Operating System

MVCC
Fork/Join
Tasks

LVars
IVars
Spaces
Futures
Events

State
Composition

Procedures
Fibers

Events

Algorithmic
Skeletons
OpenMP (loops)
Map/Reduce

Custom/optimized

SQL
Parallel
Haskell (pH)
Reactive
Programming
Streams
Dataflow

Low level

High level

Lee EA. The problem with threads. Computer. 2006.

Van Renesse R. Goal-oriented programming, or composition using events, or threads considered harmful. In ACM SIGOPS European Workshop 1998.

Parallel Programming Is Hard!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods



Concurrent/Parallel Programming

Transactional
Memory
Locks
Message-
Passing
Processes

compare-
and-swap
Threads

Operating System

LVars

MV

IVa

Fork/Join

Tasks

Work Stealing

✓ State
✓ Composition

outines

ibers

Algorithmic
Skeletons

OpenMP (loops)

Map/Reduce

Custom/optimized

SQL

Parallel
Haskell (pH)

Reactive
Programming

Streams

Dataflow

Low level

High level

Herlihy M, Moss JE. Transactional memory: Architectural support for lock-free data structures. ACM; 1993 Jun 1.

Shavit N, Touitou D. Software transactional memory. Distributed Computing. 1997 Feb 1;10(2):99-116.

Parallel Programming Is Hard!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods



Concurrent/Parallel Programming

Transactional
Memory
Locks
Message-
Passing
Processes
Threads
Operating System

LVars
MVars
IVars
Fork/Join
Threads
Tasks
Work Queue

Routines
Fibers
Custom/optimized

Algorithmic
Skeletons
OpenMP (loops)
Map/Reduce
Dataflow

SQL
Parallel
Haskell (pH)
Reactive
Programming
Streams
Dataflow

Low level

High level

✓ State
✓ Composition
✗ Composition

J. Swalens, S. Marr, J. De Koster, and T. Van Cutsem. 2014. Towards Composable Concurrency Abstractions. In Proceedings of PLACES'14.

J. Swalens, J. De Koster, and W. De Meuter. 2016. Transactional Tasks: Parallelism in Software Transactions. In Proceedings of ECOOP'16.

Parallel Programming Is Hard!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods



Concurrent/Parallel Programming

Transactional
Memory
Locks
Message-
Passing
Processes

compare-
and-swap
Threads

Operating System

Fork/Join

LVars
MVars
IVars

Spa

Tasks

? State
? Composition

Work-Stealing

oroutines
Fibers
ts

Algorithmic
Skeletons
OpenMP (loops)
Map/Reduce

Custom/optimized

SQL

Parallel
Haskell (pH)

Reactive
Programming

Streams

Dataflow

Low level

High level

Mou ZG, Hudak P. An algebraic model for divide-and-conquer and its parallelism. *The Journal of Supercomputing*. 1988.

Lämmel R. Google's MapReduce programming model—Revisited. *Science of computer programming*. 2008.

Cole MI. Algorithmic skeletons: structured management of parallel computation. London: Pitman; 1989.

Buono D, Danelutto M, Lametti S. Map, reduce and mapreduce, the skeleton way. *Procedia computer science*. 2010

Dagum L, Menon R. OpenMP: An industry-standard API for shared-memory programming. *Computing in Science & Engineering*. 1998.

Parallel Programming Is Hard!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods



Concurrent/Parallel Programming

Transactional
Memory
Locks
Message-
Passing
Processes
Threads
Operating System

compare-
and-swap
Fork/Join
Tasks

LVars
MVs
IVAs
Spaces
Work

State
Composition
Systems

Algorithmic
Skeletons
OpenMP (loops)
Map/Reduce
Custom/optimized

SQL
Parallel
Haskell (pH)
Reactive
Programming
Streams
Dataflow

Low level

High level

Chamberlin DD, Boyce RF. SEQUEL: A structured English query language. In Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control. ACM.

Nikhil RS. Implicit parallel programming in pH. Morgan Kaufmann; 2001.

Chambers C, Raniwala A, Perry F, Adams S, Henry RR, Bradshaw R, Weizenbaum N. FlumeJava: easy, efficient data-parallel pipelines. In ACM Sigplan Notices 2010. ACM.

Coutts D, Leshchinskiy R, Stewart D. Stream fusion: From lists to streams to nothing at all. In ACM SIGPLAN Notices 2007. ACM.

Thies W, Karczmarek M, Amarasinghe S. StreamIt: A language for streaming applications. In International Conference on Compiler Construction 2002. Springer.

Parallel Programming Is Hard!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods



Concurrent/Parallel Programming

Transactional
Memory
Locks
Message-
Passing
Processes

LVars
MVars

Composition and state are key for systems!

Algorithmic
Composition

SQL
Parallel
Haskell (pH)
Reactive
Programming
Streams
Dataflow

compa
and-sw

Threads

Tasks

Futures

Events

Map/Reduce

Operating System

Work-Stealing

Custom/optimized

Low level

High level

Explicit

Implicit

Concurrency Bugs

Optimizations
(Sequential \Rightarrow Parallel)

Parallel Programming Is Hard!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods



Concurrent/Parallel Programming

Transactional
Memory
Locks
Message-
Passing
Processes

compa
and-sw

Threads

LVars
MVars

Tasks
Futures

Events

Algorithmic
Composition

Map/Reduce

SQL

Parallel
Haskell (pH)
Reactive
Programming

Streams

Dataflow

Work-Stealing

Custom/optimized

Low level

High level

Explicit

Implicit

Concurrency Bugs

PL/compiler co-design!

Optimizations
(Sequential \Rightarrow Parallel)

Ingredients for Implicit Parallel Programming

Programming Model/ Language

- No concurrency abstractions.
- Practicality:
 - Integration into existing programming models (e.g., OOP)/languages.
 - Gradual switch/reuse of existing code base.

Ingredients for Implicit Parallel Programming

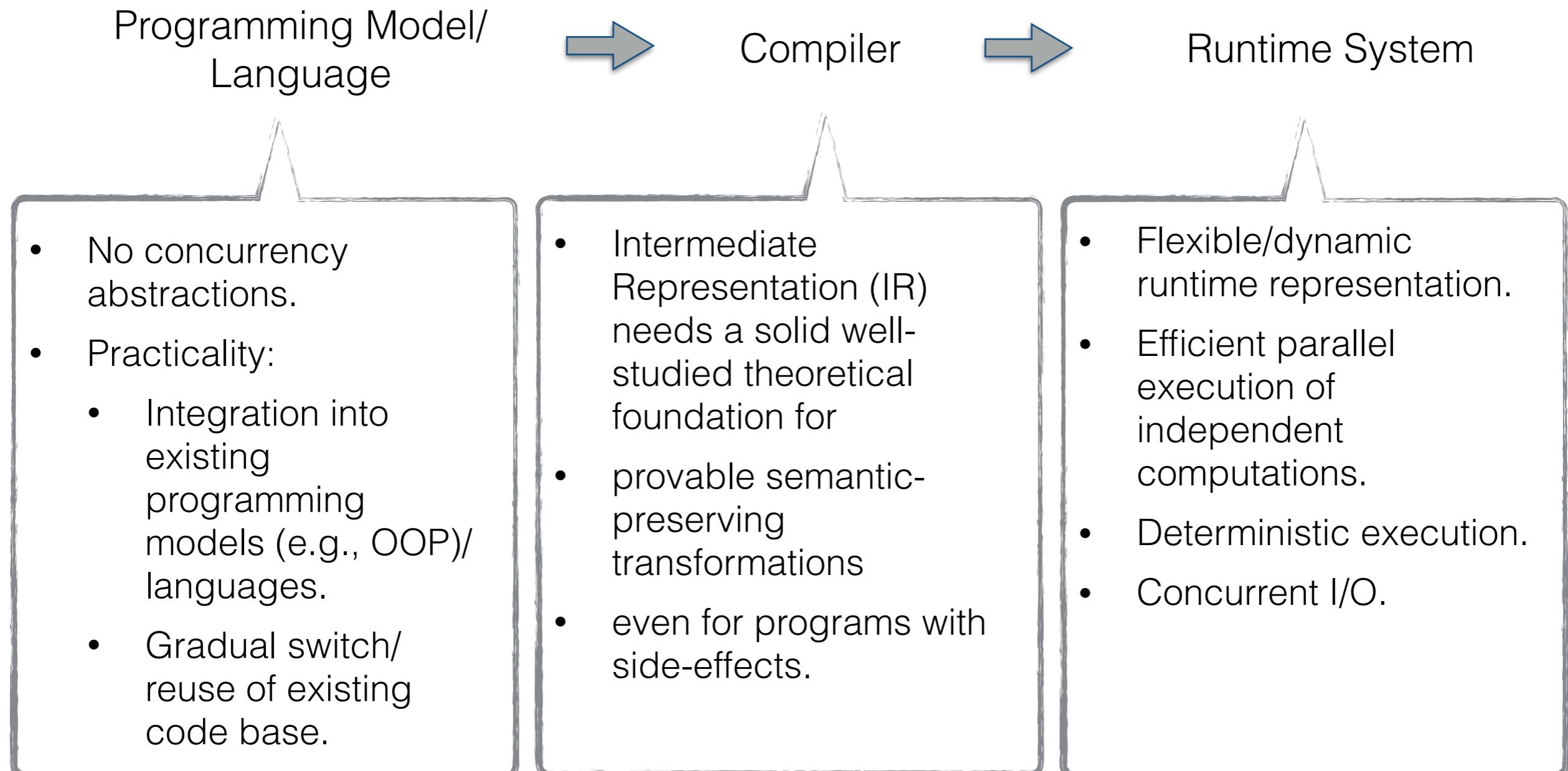
Programming Model/
Language



Compiler

- No concurrency abstractions.
- Practicality:
 - Integration into existing programming models (e.g., OOP)/languages.
 - Gradual switch/reuse of existing code base.
- Intermediate Representation (IR) needs a solid well-studied theoretical foundation for
 - provable semantic-preserving transformations
 - even for programs with side-effects.

Ingredients for Implicit Parallel Programming



Programming Model/
Language



Compiler



Runtime System

Programming Model/
Language



Compiler



Runtime System

Programming Model/
Language

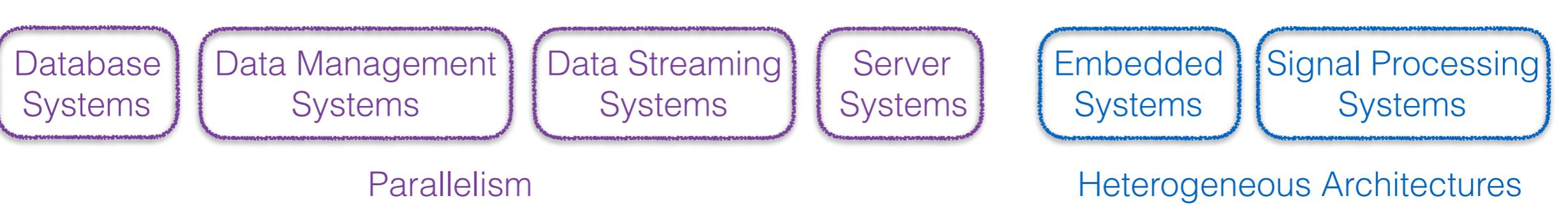


Compiler

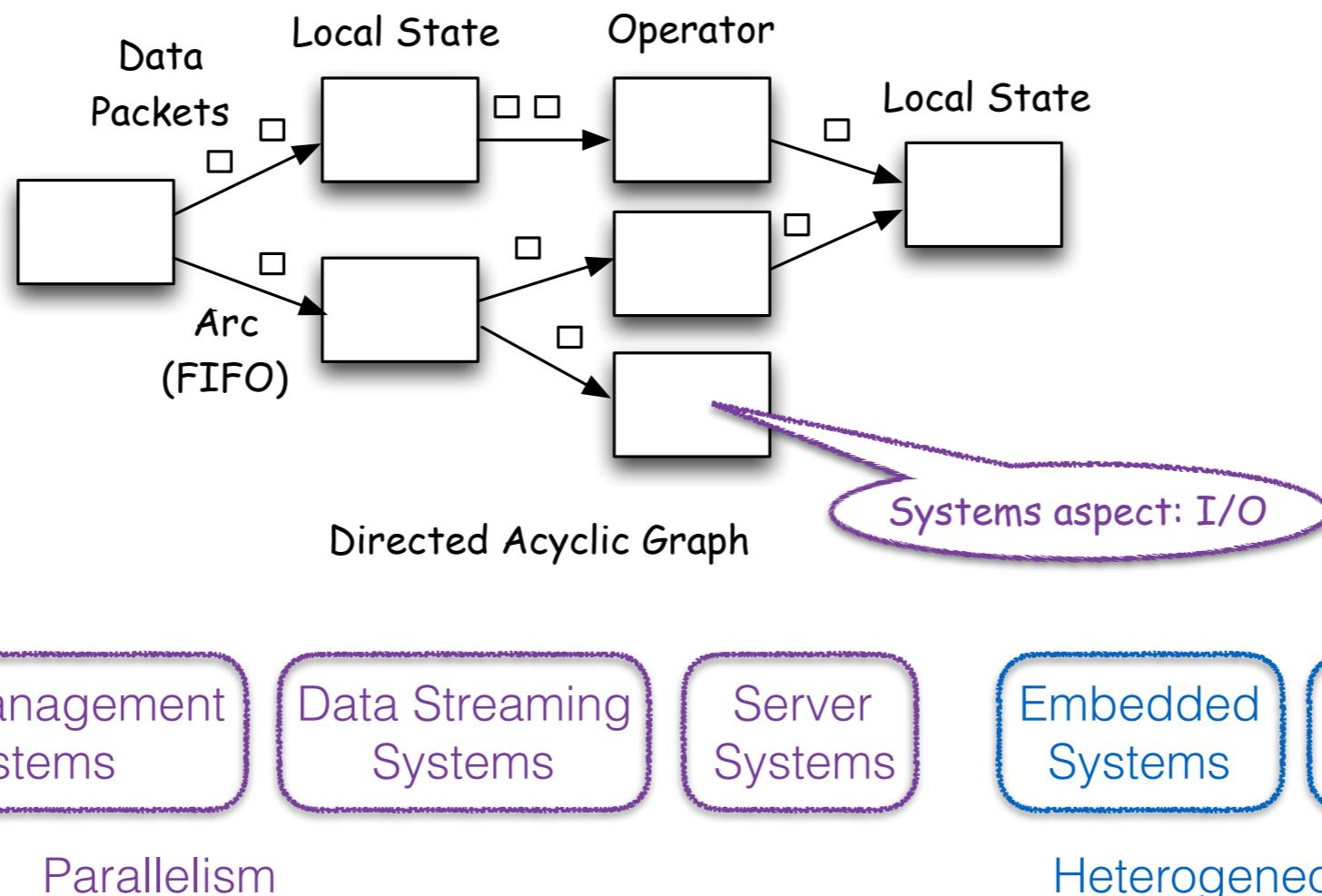


Runtime System

- Parallelism
- Mapping/Scheduling
-



Dataflow



J. B. Dennis. Data flow supercomputers. Computer, 13(11):48–56, Nov. 1980.

J. P. Morrison. Flow-Based Programming. Nostrand Reinhold, 1994

Arvind and D. E. Culler. Annual review of computer science vol. 1, 1986. chapter Dataflow architectures. Annual Reviews Inc.

Programming Model/
Language



Compiler



Runtime System

- Parallelism
- Mapping/Scheduling
-

Programming Model/
Language



Compiler



Runtime System

Dataflow

- Parallelism
- Mapping/Scheduling
- Dynamic Software Evolution

Sebastian Ertel and Pascal Felber. 2014. A framework for the dynamic evolution of highly-available dataflow programs. In Proceedings of the 15th International Middleware Conference (Middleware '14). ACM.

Sebastian Ertel, Christof Fetzer, and Pascal Felber. 2015. Ohua: Implicit Dataflow Programming for Concurrent Systems. In Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ '15). ACM.

Programming Model/
Language



Compiler



Runtime System

Dataflow

- Parallelism
- Mapping/Scheduling
- Dynamic Software Evolution

Programming Model/
Language



Compiler



Runtime System
Dataflow

- Flexible/dynamic runtime representation.
- Efficient parallel execution of independent computations.
- Deterministic execution.
- Concurrent I/O.

Programming Model/
Language



Compiler



Runtime System
Dataflow

Dataflow:

- Not concise
- Limited scalability
- Concurrency abstraction

Programming Model/
Language



Compiler

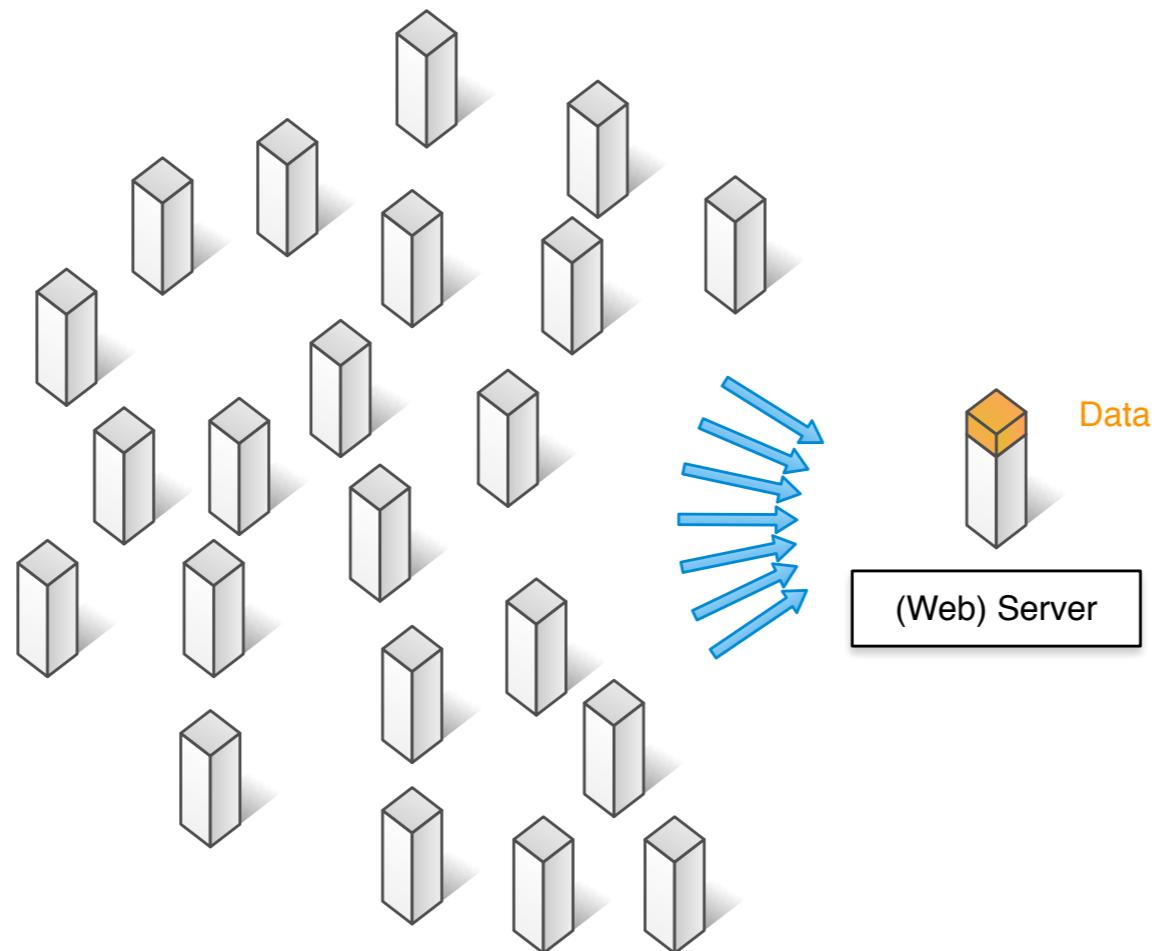


Runtime System
Dataflow

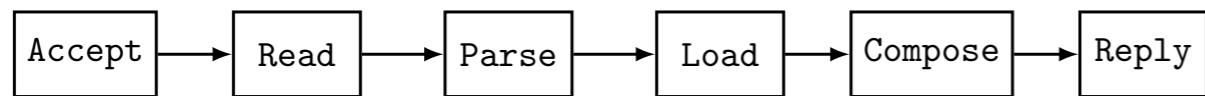
SFP:

- Variables
- Functions
- State

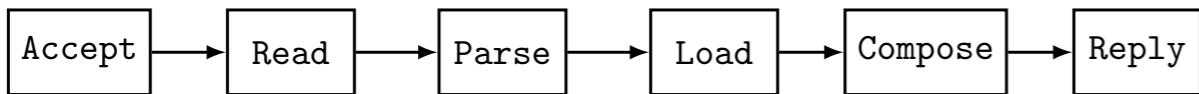
Server Architecture



Explicit Dataflow Construction



Explicit Dataflow Construction

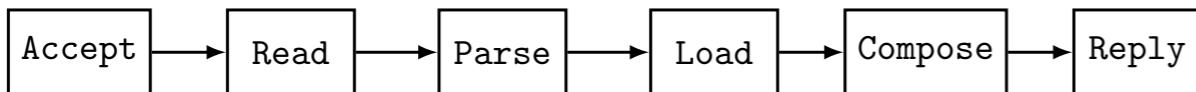


```
public class WebServer extends StreamFlexGraph {
    private Filter a, r, p, l, c, rep;

    public WebServer() {
        a = makeFilter(Accept.class);
        r = makeFilter(Read.class);
        p = makeFilter(Parse.class);
        l = makeFilter(Compose.class);
        rep = makeFilter(Reply.class);
        connect(a, r);
        connect(r, p);
        connect(p, l);
        connect(l, c);
        connect(c, rep);
        validate();
    }

    public void start() {
        new Synthesizer(a).start();
        super.start();
    }
}
```

Implicit Dataflow Construction

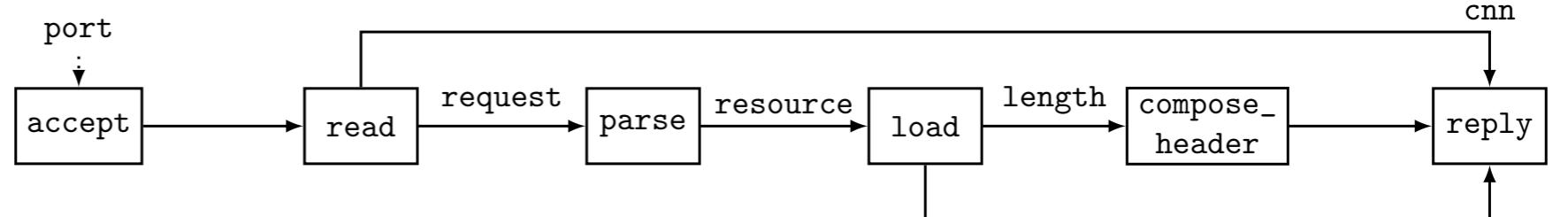


```
public class WebServer extends StreamFlexGraph {  
    private Filter a, r, p, l, c, rep;
```

```
    public WebServer() {  
        a = makeFilter(Accept.class);  
        r = makeFilter(Read.class);  
        p = makeFilter(Parse.class);  
        l = makeFilter(Compose.class);  
        rep = makeFilter(Reply.class);  
        connect(a, r);  
        connect(r, p);  
        connect(p, l);  
        connect(l, c);  
        connect(c, rep);  
        validate();  
    }
```

```
    public void start() {  
        new Synthesizer(a).start();  
        super.start();  
    }  
}
```

```
fn server(port) {  
    let (cnn, request) = read(accept(port));  
    let resource = parse(request);  
    let (content, length) = load(resource);  
    reply(cnn, compose_header(length), content)  
}
```



Classic Operators

```
class FileLoad extends Filter {  
  
    private Channel<String> in, out;  
  
    public void work() {  
        // explicit channel control  
        String resource = in.take();  
        String contents = load(resource);  
        out.put(contents);  
    }  
  
}
```



Explicit dataflow

Classic Operators

```
class FileLoad extends Filter {  
  
    private Channel<String> in, out;  
  
    public void work() {  
        // explicit channel control  
        String resource = in.take();  
        String contents = load(resource);  
        out.put(contents);  
    }  
  
    private Map<String, String> cache = new HashMap<>();  
  
    private String load(String resource){  
        String content = null;  
        if(!cache.containsKey(resource)){  
            content = new String(  
                Files.readAllBytes(Paths.get(resource))  
            );  
            cache.put(resource,content);  
            // cache eviction emitted for brevity  
        } else {  
            content = cache.get(resource);  
        }  
        return content;  
    }  
}
```



Explicit dataflow



Computation

Stateful Functions

```
class FileLoad extends Filter {  
    private Channel<String> in, out;  
  
    public void work() {  
        // explicit channel control  
        String resource = in.take();  
        String contents = load(resource);  
        out.put(contents);  
    }  
  
    private Map<String, String> cache = new HashMap<>();  
  
    private String load(String resource){  
        String content = null;  
        if(!cache.containsKey(resource)){  
            content = new String(  
                Files.readAllBytes(Paths.get(resource))  
            );  
            cache.put(resource,content);  
            // cache eviction emitted for brevity  
        } else {  
            content = cache.get(resource);  
        }  
        return content;  
    }  
}
```



Computation

Stateful Functions

```
class FileLoad extends Filter {  
    private Channel<String> in, out;  
  
    public void work() {  
        // explicit channel control  
        String resource = in.take();  
        String contents = load(resource);  
        out.put(contents);  
    }  
  
    private Map<String, String> cache = new HashMap<>();  
  
    private String load(String resource){  
        String content = null;  
        if(!cache.containsKey(resource)){  
            content = new String(  
                Files.readAllBytes(Paths.get(resource))  
            );  
            cache.put(resource,content);  
            // cache eviction emitted for brevity  
        } else {  
            content = cache.get(resource);  
        }  
        return content;  
    }  
}  
  
class FileLoad {  
    Map<String, String> cache = new HashMap<>();  
  
    String load(String resource) {  
        String contents = null;  
        // load file data from disk or cache (omitted)  
        return contents;  
    }  
}
```



The diagram illustrates a computation process. On the left, there is a block of Java code representing stateful computation. An arrow points from this code to the right, where the word "Computation" is written above a large upward-pointing arrow. This visual metaphor suggests that the computation described in the code results in a value or state that can be retrieved.

Computation

Computation is (almost) a function:

- ✗ Some loss of dataflow expressiveness.
- ✓ Solid theoretical foundation.
- ✓ Deterministic.

Stateful Functions

```
class FileLoad extends Filter {  
  
    private Channel<String> in, out;  
  
    public void work() {  
        // explicit channel control  
        String resource = in.take();  
        String contents = load(resource);  
        out.put(contents);  
    }  
  
    private Map<String, String> cache = new HashMap<>();  
  
    private String load(String resource){  
        String content = null;  
        if(!cache.containsKey(resource)){  
            content = new String(  
                Files.readAllBytes(Paths.get(resource))  
            );  
            cache.put(resource,content);  
            // cache eviction emitted for brevity  
        } else {  
            content = cache.get(resource);  
        }  
        return content;  
    }  
}
```



```
class FileLoad {  
    Map<String, String> cache = new HashMap<>();  
  
    String load(String resource) {  
        String contents = null;  
        // load file data from disk or cache (omitted)  
        return contents;  
    }  
}
```

Stateful Functions

```
class FileLoad extends Filter {  
  
    private Channel<String> in, out;  
  
    public void work() {  
        // explicit channel control  
        String resource = in.take();  
        String contents = load(resource);  
        out.put(contents);  
    }  
  
    private Map<String, String> cache = new HashMap<>();  
  
    private String load(String resource){  
        String content = null;  
        if(!cache.containsKey(resource)){  
            content = new String(  
                Files.readAllBytes(Paths.get(resource))  
            );  
            cache.put(resource,content);  
            // cache eviction emitted for brevity  
        } else {  
            content = cache.get(resource);  
        }  
        return content;  
    }  
}
```



```
class FileLoad {  
    Map<String, String> cache = new HashMap<>();  
  
    String load(String resource) {  
        String contents = null;  
        // load file data from disk or cache (omitted)  
        return contents;  
    }  
}  
  
struct FileLoad {  
    cache : HashMap,  
};  
  
impl FileLoad {  
    fn load(&self, resource:String) -> String {  
        let contents : String = {  
            // load file data from disk or cache (omitted)  
        };  
        contents  
    }  
}
```



Stateful Functions

```
class FileLoad extends Filter {  
  
    private Channel<String> in, out;  
  
    public void work() {  
        // explicit channel control  
        String resource = in.take();  
        String contents = load(resource);  
        out.put(contents);  
    }  
  
    private Map<String, String> cache = new HashMap<>();  
  
    private String load(String resource){  
        String content = null;  
        if(!cache.containsKey(resource)){  
            content = new String(  
                Files.readAllBytes(Paths.get(resource))  
            );  
            cache.put(resource,content);  
            // cache eviction emitted for brevity  
        } else {  
            content = cache.get(resource);  
        }  
        return content;  
    }  
}
```



```
class FileLoad {  
    Map<String, String> cache = new HashMap<>();  
  
    String load(String resource) {  
        String contents = null;  
        // load file data from disk or cache (omitted)  
        return contents;  
    }  
}  
  
struct FileLoad {  
    cache : HashMap,  
};  
  
impl FileLoad {  
    fn load(&self, resource:String) -> String {  
        let contents : String = {  
            // load file data from disk or cache (omitted)  
        };  
        contents  
    }  
}  
  
char* load(char* resource) {  
    static GHashTable* cache = g_hash_table_new();  
    char* contents = NULL;  
    // load file data from disk or cache (omitted)  
    return contents;  
}
```



Stateful Functions

➤ ?

```
class FileLoad extends Filter {  
  
    private Channel<String> in, out;  
  
    public void work() {  
        // explicit channel control  
        String resource = in.take();  
        String contents = load(resource);  
        out.put(contents);  
    }  
  
    private Map<String, String> cache = new HashMap<>();  
  
    private String load(String resource){  
        String content = null;  
        if(!cache.containsKey(resource)){  
            content = new String(  
                Files.readAllBytes(Paths.get(resource))  
            );  
            cache.put(resource,content);  
            // cache eviction emitted for brevity  
        } else {  
            content = cache.get(resource);  
        }  
        return content;  
    }  
}
```



```
class FileLoad {  
    Map<String, String> cache = new HashMap<>();  
  
    String load(String resource) {  
        String contents = null;  
        // load file data from disk or cache (omitted)  
        return contents;  
    }  
}  
  
struct FileLoad {  
    cache : HashMap,  
};  
  
impl FileLoad {  
    fn load(&self, resource:String) -> String {  
        let contents : String = {  
            // load file data from disk or cache (omitted)  
        };  
        contents  
    }  
}  
  
char* load(char* resource) {  
    static GHashTable* cache = g_hash_table_new();  
    char* contents = NULL;  
    // load file data from disk or cache (omitted)  
    return contents;  
}
```



Stateful Functions



```
class FileLoad extends Filter {  
  
    private Channel<String> in, out;  
  
    public void work() {  
        // explicit channel control  
        String resource = in.take();  
        String contents = load(resource);  
        out.put(contents);  
    }  
  
    private Map<String, String> cache = new HashMap<>();  
  
    private String load(String resource){  
        String content = null;  
        if(!cache.containsKey(resource)){  
            content = new String(  
                Files.readAllBytes(Paths.get(resource))  
            );  
            cache.put(resource,content);  
            // cache eviction emitted for brevity  
        } else {  
            content = cache.get(resource);  
        }  
        return content;  
    }  
}
```

State encapsulation



```
class FileLoad {  
    Map<String, String> cache = new HashMap<>();  
  
    String load(String resource) {  
        String contents = null;  
        // load file data from disk or cache (omitted)  
        return contents;  
    }  
}  
  
struct FileLoad {  
    cache : HashMap,  
};  
  
impl FileLoad {  
    fn load(&self, resource:String) -> String {  
        let contents : String = {  
            // load file data from disk or cache (omitted)  
        };  
        contents  
    }  
}  
  
char* load(char* resource) {  
    static GHashTable* cache = g_hash_table_new();  
    char* contents = NULL;  
    // load file data from disk or cache (omitted)  
    return contents;  
}
```



Local State

```
fn server(port) {  
    let (cnn, req)      = read(accept(port));  
    let resource        = parse(req);  
    let (content, length) = load(resource);  
    reply(cnn, compose(length), content)  
}
```



```
struct FileLoad {  
    cache : HashMap,  
};  
  
impl FileLoad {  
    fn load(&self, resource:String) -> String {  
        let contents : String = {  
            // load file data from disk or cache (omitted)  
        };  
        contents  
    }  
}
```

State is local to each call-site!

Programming Model/
Language



Compiler



Runtime System
Dataflow

SFP:

- Variables
- Stateful Functions
- Composition

Programming Model/
Language



Compiler



Dataflow
Runtime System

SFP:

- Variables
- Stateful Functions
- Composition

SFP: Stateful Functional Programming

Programming Model/
Language



Compiler



Dataflow
Runtime System

SFP:

- Variables
- Stateful Functions
- Composition
- Code reuse!

SFP: Stateful Functional Programming

Algorithms

```
fn server(port) {
    let (cnn, req)      = read(accept(port));
    let resource        = parse(req);
    let (content, length) = load(resource);
    reply(cnn, compose_header(length), content)
}
```

Top-level Algorithms

Abstraction

```
fn respond(cnn, content, length){
    reply(cnn, compose_header(length), content)
}
```

```
fn server(port) {
    let (cnn, req)      = read(accept(port));
    let resource        = parse(req);
    let (content, length) = load(resource);
    respond(cnn, content, length)
}
```

Lambda Algorithms

```
fn server(port) {
    let respond = |cnn, content, length|
        reply(cnn, compose_header(length), content);
    let (cnn, req)      = read(accept(port));
    let resource        = parse(req);
    let (content, length) = load(resource);
    respond(cnn, content, length)
}
```

Application

Programming Model/
Language



Compiler



Runtime System
Dataflow

SFP:

- Variables
- Stateful Functions
- Composition
- Algorithms

Programming Model/
Language



Compiler

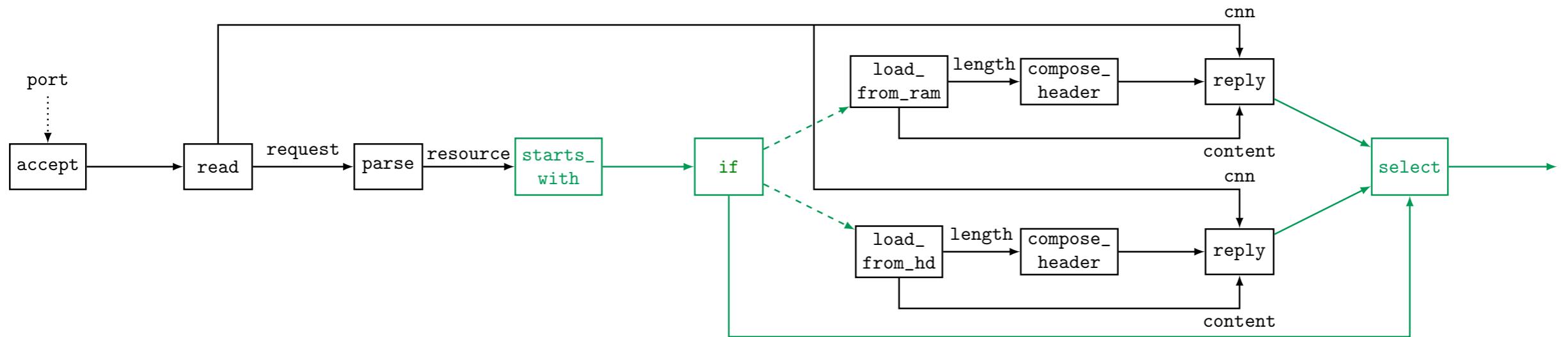


Runtime System
Dataflow

SFP:

- Variables
- Stateful Functions
- Composition
- Algorithms
- Control Flow

Control Flow



```

fn respond(cnn, content, length){
  reply(cnn, compose_header(length), content)
}
  
```

```

fn server(port) {
  let (cnn, req)      = read(accept(port));
  let resource        = parse(req);
  let (content, length) = load(resource);
  respond(cnn, content, length)
}
  
```

```

fn respond(cnn, content, length){
  reply(cnn, compose_header(length), content)
}
  
```

```

fn server(port) {
  let (cnn, req) = read(accept(port));
  let resource   = parse(req);
  if startsWith(resource, "news/") {
    let (content, length) = load_from_ram(resource);
    respond(cnn, content, length)
  } else {
    let (content, length) = load_from_hd(resource);
    reply(cnn, content, length)
  }
}
  
```

Programming Model/
Language



Compiler



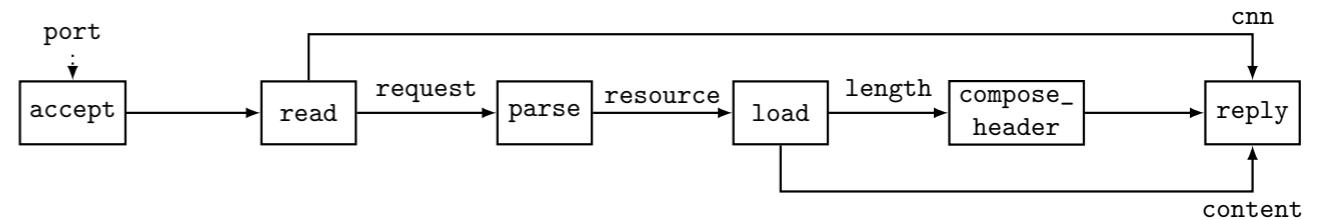
Runtime System
Dataflow

SFP:

- Variables
- Stateful Functions
- Composition
- Algorithms
- Control Flow

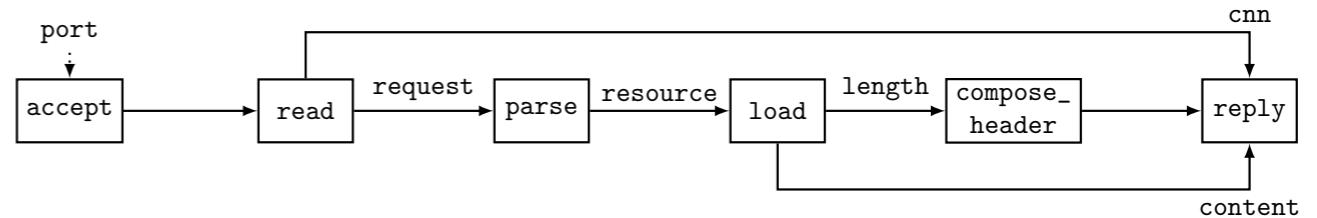
Semantic Gap

```
fn server(port) {  
    let cnn = accept(port);  
    let (cnn, req) = read(cnn);  
    let resource = parse(req);  
    let (content, length) = load(resource);  
    reply(cnn, compose_header(length), content)  
}
```



Semantic Gap

```
fn server(port) {  
    let cnn = accept(port);  
    let (cnn, req) = read(cnn);  
    let resource = parse(req);  
    let (content, length) = load(resource);  
    reply(cnn, compose_header(length), content)  
}
```



Typical PL semantics:

The **server** function executes **once** for each invocation.

Dataflow semantics:

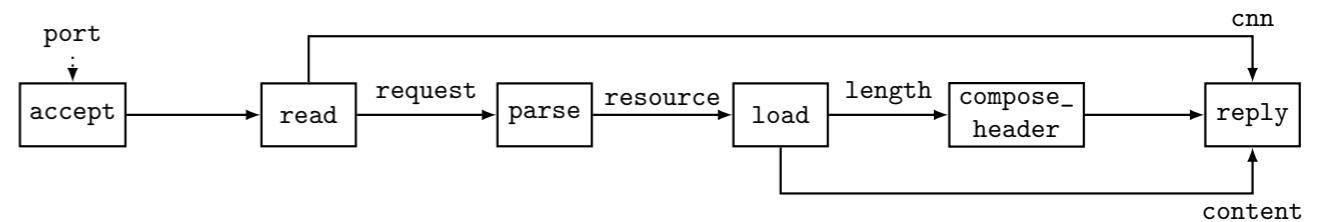
The **server** algorithm executes **forever** for each invocation.



The **accept** node is an I/O source operator, i.e., it finishes when all requests have been exhausted!

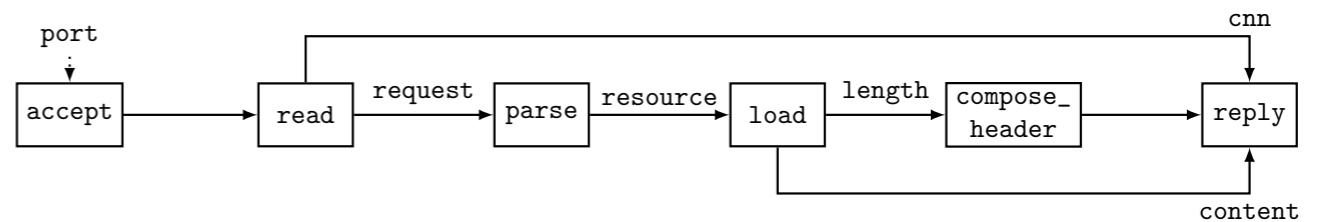
SMap

```
fn server(port) {  
    let cnn = accept(port);  
    let (cnn, req) = read(cnn);  
    let resource = parse(req);  
    let (content, length) = load(resource);  
    reply(cnn, compose_header(length), content)  
}
```



SMap

```
fn server(port) {  
    let cnn = accept(port);  
    let (cnn, req) = read(cnn);  
    let resource = parse(req);  
    let (content, length) = load(resource);  
    reply(cnn, compose_header(length), content)  
}
```

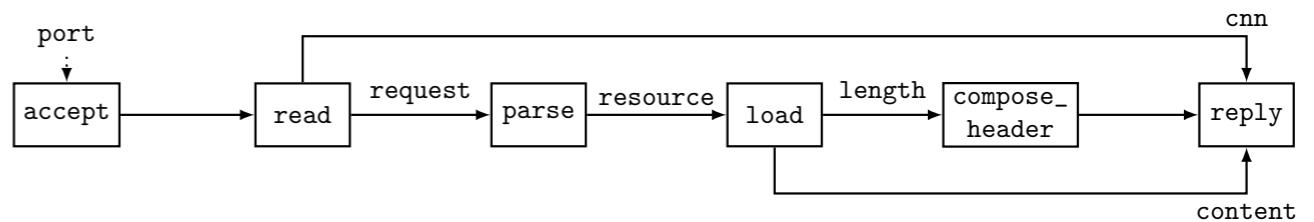


Abstract over processing:

```
fn request_handling(cnn) {  
    let (cnn, req) = read(cnn);  
    let resource = parse(req);  
    let (content, length) = load(resource);  
    reply(cnn, compose_header(length), content)  
}  
  
fn server(port) {  
    let cnn = accept(port);  
    request_handling(cnn)  
}
```

SMap

```
fn server(port) {  
    let cnn = accept(port);  
    let (cnn, req) = read(cnn);  
    let resource = parse(req);  
    let (content, length) = load(resource);  
    reply(cnn, compose_header(length), content)  
}
```

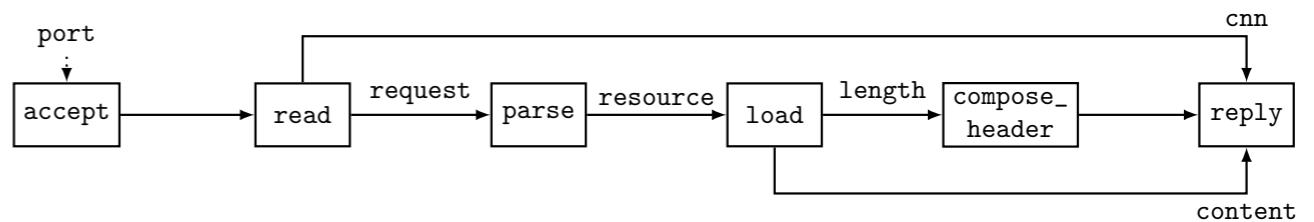


Abstract over processing:

```
fn request_handling(cnn) {  
    let (cnn, req) = read(cnn);  
    let resource = parse(req);  
    let (content, length) = load(resource);  
    reply(cnn, compose_header(length), content)  
}  
  
fn server(port) {  
    let cnn = accept(port);  
    request_handling(cnn) f  
}
```

SMap

```
fn server(port) {
    let cnn = accept(port);
    let (cnn, req) = read(cnn);
    let resource = parse(req);
    let (content, length) = load(resource);
    reply(cnn, compose_header(length), content)
}
```



Abstract over processing:

```
fn request_handling(cnn) {
    let (cnn, req) = read(cnn);
    let resource = parse(req);
    let (content, length) = load(resource);
    reply(cnn, compose_header(length), content)
}
```

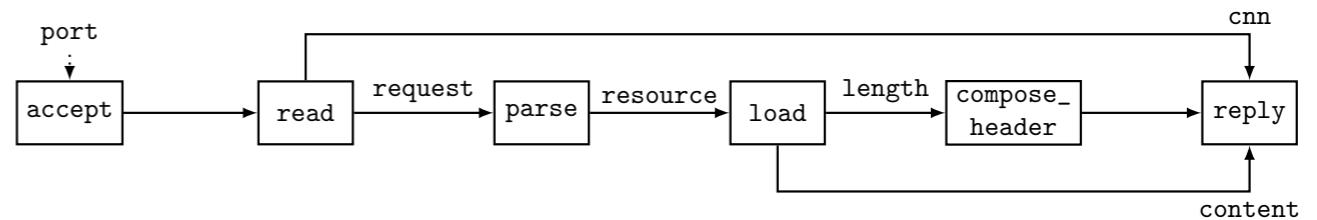
```
fn server(port) {
    let cnn = accept(port);      I = [a1, a2]
    request_handling(cnn)       f
}
```

$$I' = [b_1, b_2] = \text{map}(f, I)$$

$$\begin{aligned} b_1 &= f(a_1) \\ b_2 &= f(a_2) \end{aligned}$$

SMap

```
fn server(port) {
    let cnn = accept(port);
    let (cnn, req) = read(cnn);
    let resource = parse(req);
    let (content, length) = load(resource);
    reply(cnn, compose_header(length), content)
}
```



Abstract over processing:

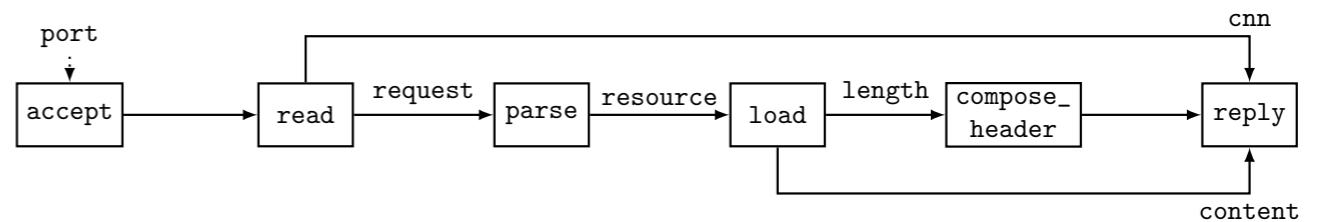
```
fn request_handling(cnn) {
    let (cnn, req) = read(cnn);
    let resource = parse(req);
    let (content, length) = load(resource);
    reply(cnn, compose_header(length), content)
}
```

```
fn server(port) {
    let cnn = accept(port);      I = [a1, a2]
    request_handling(cnn)       f
}
```

$$\begin{array}{lll}
 I' = [b_1, b_2] = \text{map}(f, I) & = & I' = [b_1, b_2] = \text{smap}(f, I) \\
 b_1 = f(a_1) & & (s'_f, b_1) = f(s_f, a_1) \\
 b_2 = f(a_2) & & (s''_f, b_2) = f(s'_f, a_2)
 \end{array}$$

SMap

```
fn server(port) {
    let cnn = accept(port);
    let (cnn, req) = read(cnn);
    let resource = parse(req);
    let (content, length) = load(resource);
    reply(cnn, compose_header(length), content)
}
```



Abstract over processing:

```
fn request_handling(cnn) {
    let (cnn, req) = read(cnn);
    let resource = parse(req);
    let (content, length) = load(resource);
    reply(cnn, compose_header(length), content)
}
```

```
fn server(port) {
    let cnn = accept(port);      I = [a1, a2]
    request_handling(cnn)       f
}
```

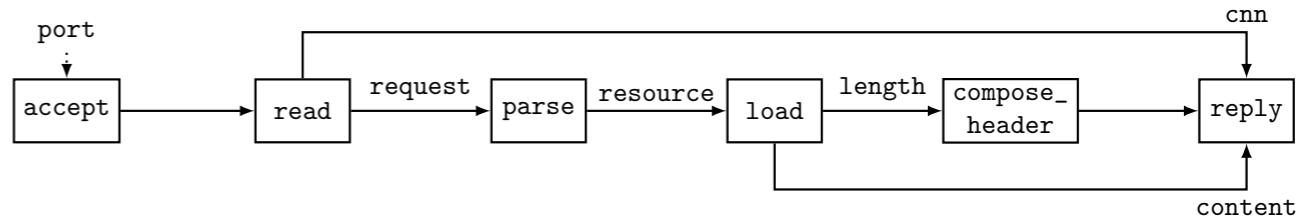
$$\begin{array}{lll}
 I' = [b_1, b_2] = \text{map}(f, I) & = & I' = [b_1, b_2] = \text{smap}(f, I) \\
 b_1 = f(a_1) & & (s'_f, b_1) = f(s_f, a_1) \\
 b_2 = f(a_2) & & (s''_f, b_2) = f(s'_f, a_2)
 \end{array}$$

data parallel

strictly sequential

SMap

```
fn server(port) {
  let cnn = accept(port);
  let (cnn, req) = read(cnn);
  let resource = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```



Abstract over processing:

```
fn request_handling(cnn) {
  let (cnn, req) = read(cnn);
  let resource = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```

```
fn server(port) {
  let cnn = accept(port);      I = [a1, a2]
  request_handling(cnn)       f
}
```

$$\begin{array}{lll} I' = [b_1, b_2] = \text{map}(f, I) & = & I' = [b_1, b_2] = \text{smap}(f, I) \\ b_1 = f(a_1) & & (s'_f, b_1) = f(s_f, a_1) \\ b_2 = f(a_2) & & (s''_f, b_2) = f(s'_f, a_2) \end{array}$$

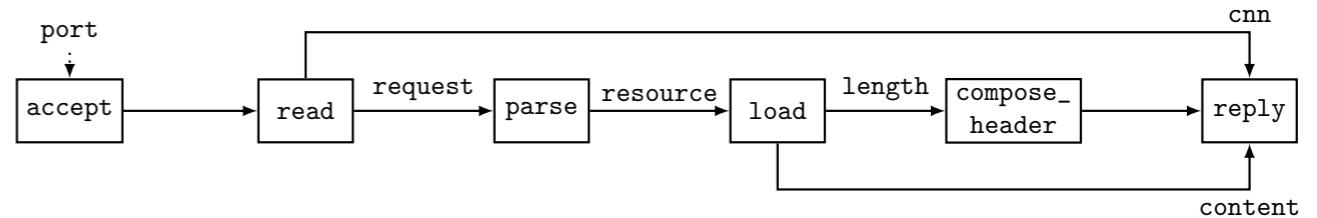
data parallel

```
fn f(a){  
  let x = g(a);  
  h(x)  
}  
→  
f = h ∘ g  
sf = (sg, sh)
```

strictly sequential

SMap

```
fn server(port) {
  let cnn = accept(port);
  let (cnn, req) = read(cnn);
  let resource = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```



Abstract over processing:

```
fn request_handling(cnn) {
  let (cnn, req) = read(cnn);
  let resource = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```

```
fn server(port) {
  let cnn = accept(port);      I = [a1, a2]
  request_handling(cnn)       f
}
```

$$\begin{array}{lll} I' = [b_1, b_2] = \text{map}(f, I) & = & I' = [b_1, b_2] = \text{smap}(f, I) \\ b_1 = f(a_1) & & (s'_f, b_1) = f(s_g, a_1) \\ b_2 = f(a_2) & & (s''_f, b_2) = f(s'_g, a_2) \end{array}$$

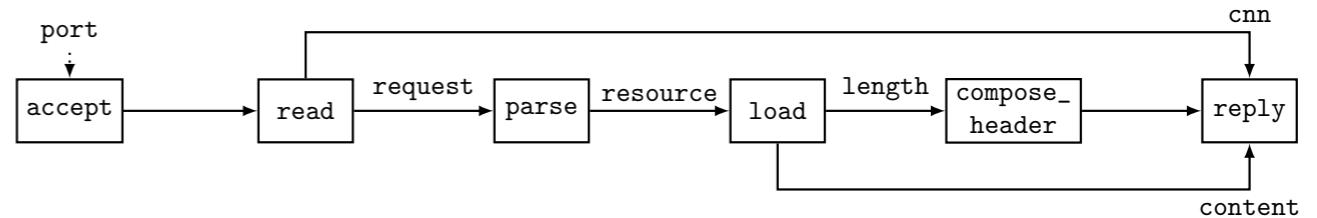
data parallel

$$\begin{array}{ll} \text{fn } f(a)\{\text{let } x = g(a); h(x)\} & (s'_g, x_1) = g(s_g, a_1) \\ \} & (s''_g, x_2) = g(s'_g, a_2) \\ \Rightarrow f = h \circ g & (s'_h, b_1) = h(s_g, x_1) \\ s_f = (s_g, s_h) & (s''_h, b_2) = h(s'_g, x_2) \end{array}$$

strictly sequential

SMap

```
fn server(port) {
    let cnn = accept(port);
    let (cnn, req) = read(cnn);
    let resource = parse(req);
    let (content, length) = load(resource);
    reply(cnn, compose_header(length), content)
}
```



Abstract over processing:

```
fn request_handling(cnn) {
    let (cnn, req) = read(cnn);
    let resource = parse(req);
    let (content, length) = load(resource);
    reply(cnn, compose_header(length), content)
}
```

```
fn server(port) {
    let cnn = accept(port);      I = [a1, a2]
    request_handling(cnn)       f
}
```

$$\begin{array}{lll} I' = [b_1, b_2] = \text{map}(f, I) & = & I' = [b_1, b_2] = \text{smap}(f, I) \\ b_1 = f(a_1) & & (s'_f, b_1) = f(s_g, a_1) \\ b_2 = f(a_2) & & (s''_f, b_2) = f(s'_g, a_2) \end{array}$$

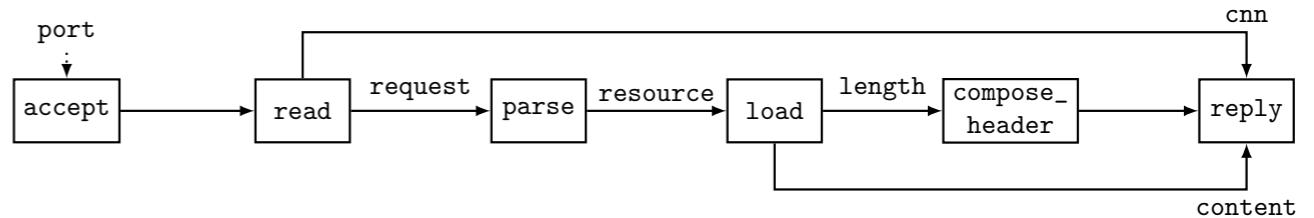
data parallel

$$\begin{array}{l} \text{fn } f(a)\{\text{let } x = g(a); h(x)\} \\ \Rightarrow f = h \circ g \\ s_f = (s_g, s_h) \end{array} \quad \begin{array}{l} (s'_g, x_1) = g(s_g, a_1) \\ (s''_g, x_2) = g(s'_g, a_2) \\ (s'_h, b_1) = h(s_g, x_1) \\ (s''_h, b_2) = h(s'_h, x_2) \end{array}$$

strictly sequential

SMap

```
fn server(port) {
    let cnn = accept(port);
    let (cnn, req) = read(cnn);
    let resource = parse(req);
    let (content, length) = load(resource);
    reply(cnn, compose_header(length), content)
}
```



Abstract over processing:

```
fn request_handling(cnn) {
    let (cnn, req) = read(cnn);
    let resource = parse(req);
    let (content, length) = load(resource);
    reply(cnn, compose_header(length), content)
}

fn server(port) {
    let cnn = accept(port);      I = [a1, a2]
    request_handling(cnn)       f
}
```

$$I' = [b_1, b_2] = \text{map}(f, I) \quad \equiv \quad I' = [b_1, b_2] = \text{smap}(f, I)$$

$$\begin{aligned} b_1 &= f(a_1) & (s'_f, b_1) &= f(s_g, a_1) \\ b_2 &= f(a_2) & (s''_f, b_2) &= f(s'_g, a_2) \end{aligned}$$

data parallel

strictly sequential

$\text{fn } f(a)\{\$
 $\text{let } x = g(a);$
 $\text{h}(x)$
 $\}$

➡

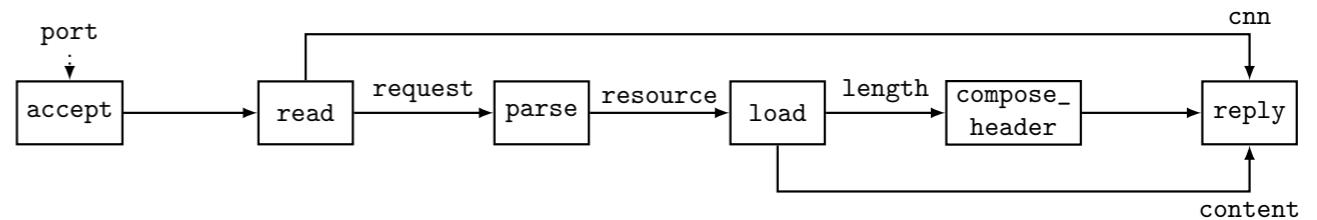
$f = h \circ g$
 $s_f = (s_g, s_h)$

$(s'_g, x_1) = g(s_g, a_1)$	$(s'_g, x_1) = g(s_g, a_1)$
$(s''_g, x_2) = g(s'_g, a_2)$	$(s'_h, b_1) = h(s_h, x_1)$
$(s'_h, b_1) = h(s_h, x_1)$	$(s''_g, x_2) = g(s'_g, a_2)$
$(s''_h, b_2) = h(s'_h, x_2)$	$(s'_h, b_2) = h(s'_h, x_2)$

pipeline parallel

SMap

```
fn server(port) {
    let cnn = accept(port);
    let (cnn, req) = read(cnn);
    let resource = parse(req);
    let (content, length) = load(resource);
    reply(cnn, compose_header(length), content)
}
```



Abstract over processing:

```
fn request_handling(cnn) {
    let (cnn, req) = read(cnn);
    let resource = parse(req);
    let (content, length) = load(resource);
    reply(cnn, compose_header(length), content)
}
```

```
fn server(port) {
    let cnn = accept(port);      I = [a1, a2]
    request_handling(cnn)       f
}
```

$$I' = [b_1, b_2] = \text{map}(f, I) \quad = \quad I' = [b_1, b_2] = \text{smap}(f, I)$$

$$\begin{aligned} b_1 &= f(a_1) & (s'_f, b_1) &= f(s_f, a_1) \\ b_2 &= f(a_2) & (s''_f, b_2) &= f(s'_f, a_2) \end{aligned}$$

data parallel

Abstract over data:

Lists

- Finite
- Infinite (stream)

Generator

- Location independent

```
fn server(port) {
    let cnns:List<Cnn> = infinite_generator() || accept(port));
    let results:List<()> = smap(request_handling, cnns);
    ()
}
```

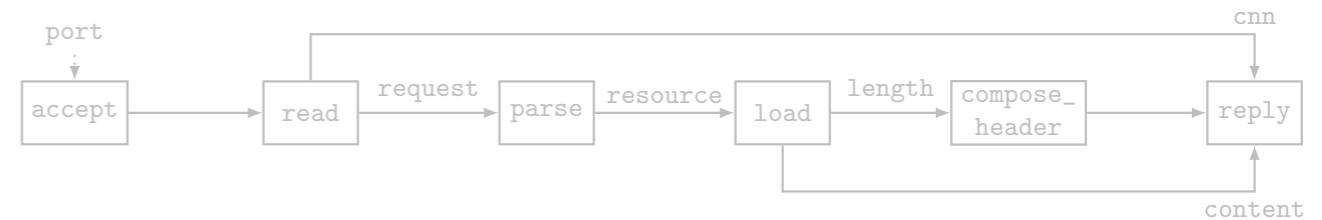
$$\begin{array}{lcl} \text{fn } f(a)\{ & (s'_g, x_1) = g(s_g, a_1) & (s'_g, x_1) = g(s_g, a_1) \\ \quad \text{let } x = g(a); & (s''_g, x_2) = g(s'_g, a_2) & (s'_h, b_1) = h(s_h, x_1) \\ \quad h(x) & (s'_h, b_1) = h(s_h, x_1) & \cancel{(s'_h, b_1) = h(s_h, x_1)} \\ \}\Rightarrow & (s''_h, b_2) = h(s'_h, x_2) & (s''_g, x_2) = g(s'_g, a_2) \\ f = h \circ g & & (s''_h, b_2) = h(s'_h, x_2) \\ s_f = (s_g, s_h) & & (s'_h, b_2) = h(s'_h, x_2) \end{array}$$

strictly sequential

pipeline parallel

SMap

```
fn server(port) {
    let cnn = accept(port);
    let (cnn, req) = read(cnn);
    let resource = parse(req);
    let (content, length) = load(resource);
    reply(cnn, compose_header(length), content)
}
```

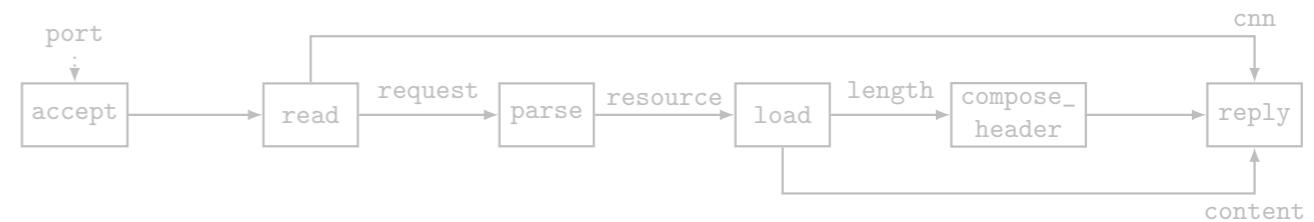


```
fn request_handling(cnn) {
    let (cnn, req) = read(cnn);
    let resource = parse(req);
    let (content, length) = load(resource);
    reply(cnn, compose_header(length), content)
}
```

```
fn server(port) {
    let cnns:List<Cnn> = infinite_generator() || accept(port));
    let results:List<()> = smap(request_handling, cnns);
    ()
}
```

SMap as a Loop

```
fn server(port) {  
    let cnn = accept(port);  
    let (cnn, req) = read(cnn);  
    let resource = parse(req);  
    let (content, length) = load(resource);  
    reply(cnn, compose_header(length), content)  
}
```



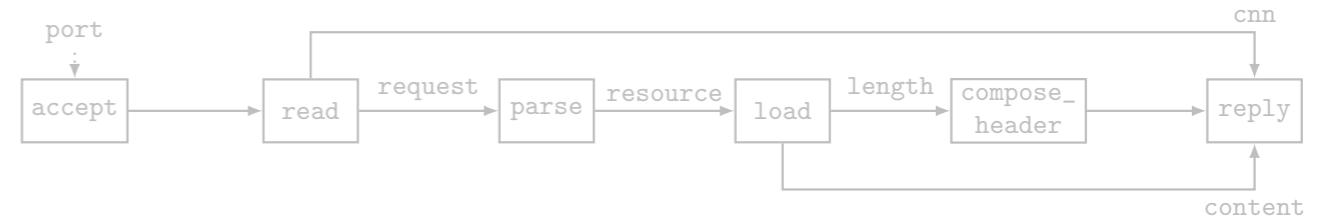
```
fn request_handling(cnn) {  
    let (cnn, req) = read(cnn);  
    let resource = parse(req);  
    let (content, length) = load(resource);  
    reply(cnn, compose_header(length), content)  
}
```

```
fn server(port) {  
    let cnns:List<Cnn> = infinite_generator(|| accept(port));  
    let results:List<()> = for cnn in cnns {  
        request_handling(cnn)  
    };  
    ()  
}
```

```
fn server(port) {  
    let cnns:List<Cnn> = infinite_generator(|| accept(port));  
    let results:List<()> = smap(request_handling, cnns);  
    ()  
}
```

From SMap to Dataflow

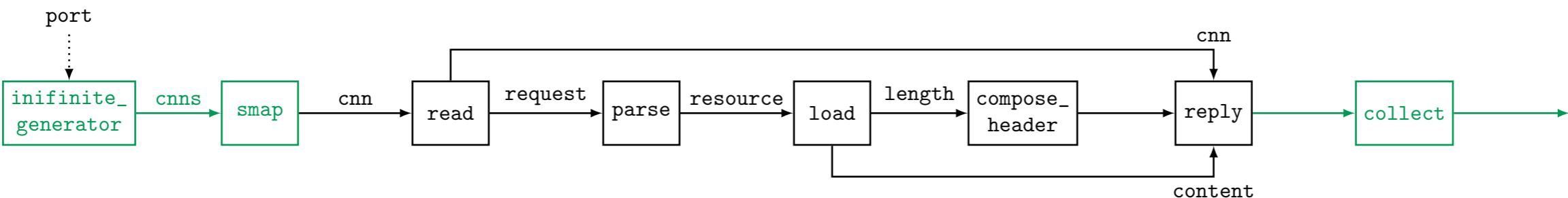
```
fn server(port) {
    let cnn = accept(port);
    let (cnn, req) = read(cnn);
    let resource = parse(req);
    let (content, length) = load(resource);
    reply(cnn, compose_header(length), content)
}
```



```
fn request_handling(cnn) {
    let (cnn, req) = read(cnn);
    let resource = parse(req);
    let (content, length) = load(resource);
    reply(cnn, compose_header(length), content)
}
```

```
fn server(port) {
    let cnns:List<Cnn> = infinite_generator() || accept(port));
    let results:List<()> = for cnn in cnns {
        request_handling(cnn)
    };
    ()
}
```

```
fn server(port) {
    let cnns:List<Cnn> = infinite_generator() || accept(port));
    let results:List<()> = smap(request_handling, cnns);
    ()
}
```



Programming Model/
Language



Compiler



Dataflow
Runtime System

SFP:

- Variables
- Stateful Functions
- Composition
- Algorithms
- Control Flow
- SMap

Programming Model/
Language



Compiler

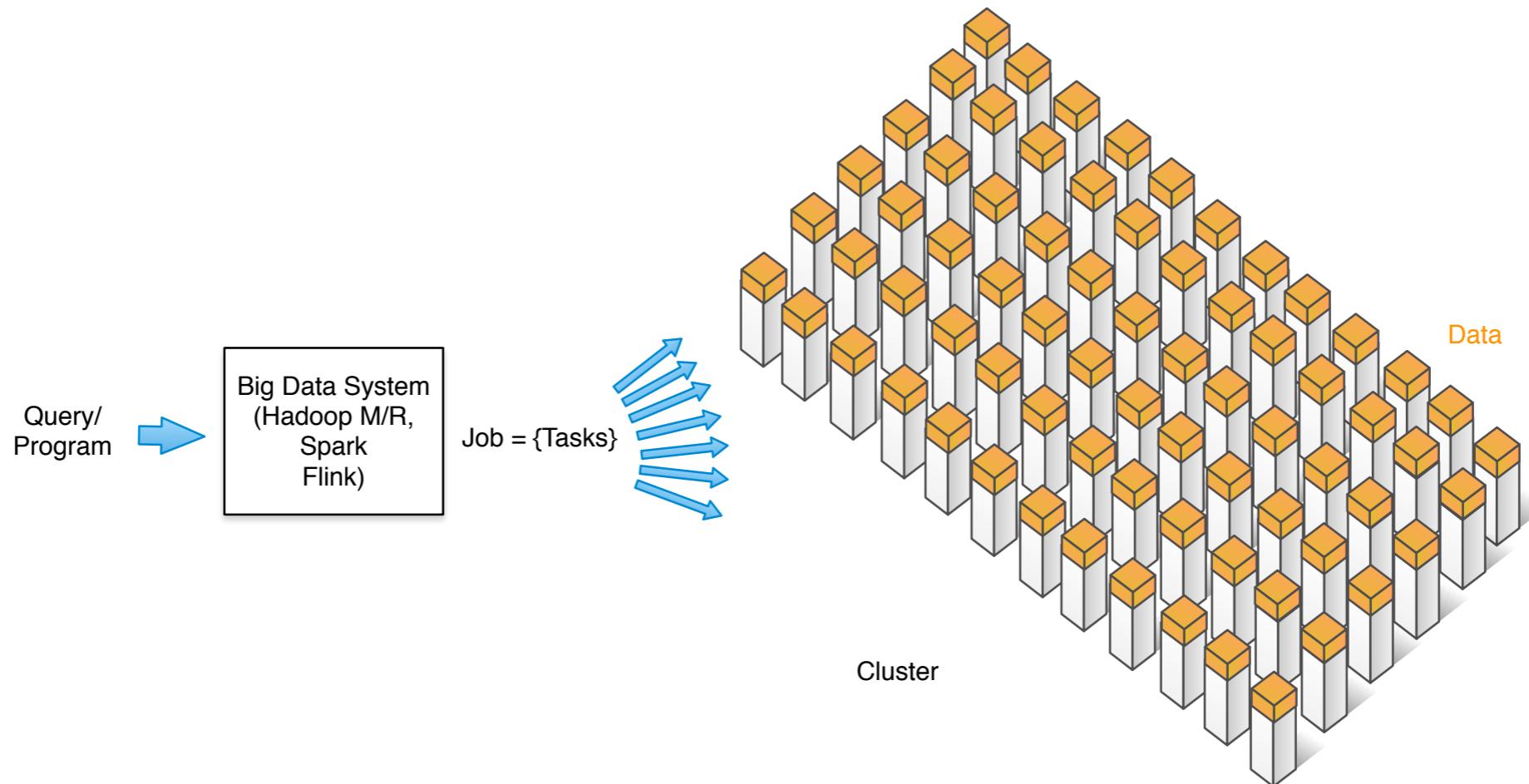


Dataflow
Runtime System

- No concurrency abstractions.
- Practicality:
 - Integration into existing programming models (e.g., OOP)/languages.
 - Gradual switch/reuse of existing code base.

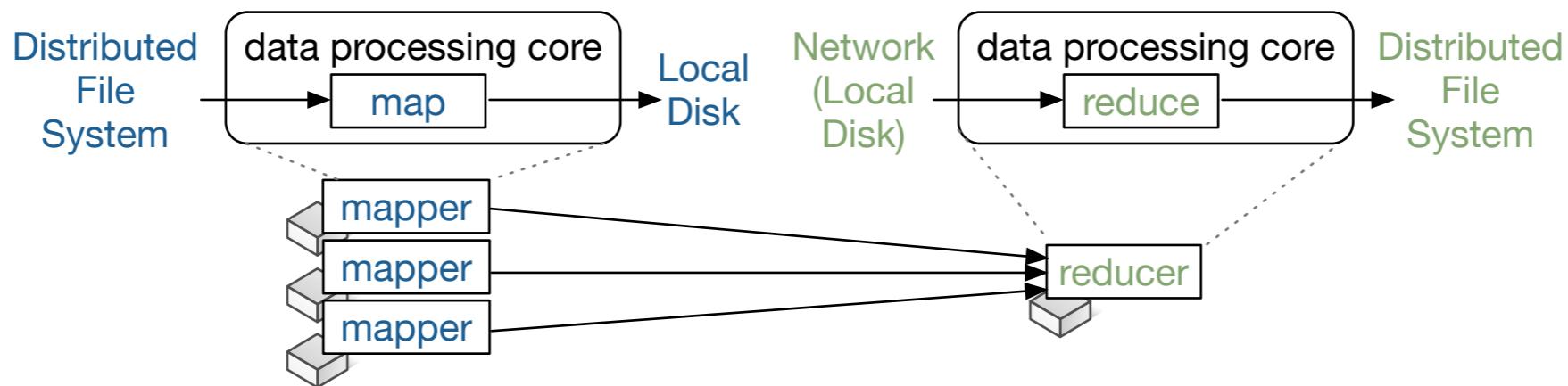
Big Data Systems

1. Big Data Systems (BDSs) scale with the number of cores in the cluster
2. General wisdom: The main bottleneck is I/O (disk and network)



Big Data Systems

1. Big Data Systems (BDSs) scale with the number of cores in the cluster **for independent tasks**.
2. General wisdom: The main bottleneck is I/O (disk and network)



Big Data Systems

1. Big Data Systems (BDSs) scale with the number of cores in the cluster **for independent tasks**.
2. General wisdom: The main bottleneck is I/O (disk and network) **for simple data**.

Jobs are CPU-bound!

WordCount, Sort	Analytics Queries (in Hive)
Simple Data Formats	vs.
Uncompressed	Complex Data Formats (Parquet, Tables, JSON)
	Compressed

Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making sense of performance in data analytics frameworks.(NSDI'15).

Big Data Systems

1. Big Data Systems (BDSs) scale with the number of cores in the cluster **for independent tasks**.
2. General wisdom: The main bottleneck is I/O (disk and network) **for simple data**.

Jobs are CPU-bound!

WordCount, Sort	Analytics Queries (in Hive)
Simple Data Formats	vs.
Uncompressed	Complex Data Formats (Parquet, Tables, JSON)
	Compressed

Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making sense of performance in data analytics frameworks.(NSDI'15).

BDS cores/data processing pipelines do not scale well on multicores!

- Local optimizations do not solve this problem.
 - BDSs do not benefit from new network HW.
- Rewrite data processing cores!

Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Ioannis Koltsidas, and Nikolas Ioannou. 2016. On the [ir]relevance of network performance for data processing. (HotCloud'16).

Data Processing Cores

The diagram illustrates the processing flow for three data processing cores: Hadoop, Spark, and Flink. It shows three main stages: **context**, **computation**, and **context**. In the first stage, **read I/O** leads to **deserialize**, which feeds into the **computation** stage. The computation stage contains **map, reduce, query graph**. In the final stage, **compress** leads to **serialize**, which then leads to **write I/O**.

Stage	Hadoop	Spark	Flink
context	Context ctxt	TaskContext ctxt	TaskContext ctxt
computation	map, reduce, query graph	mapTask(partitionId, partition)	runTask(partitionId, ctxt)
context	decompress write I/O	partitionId, ctxt writer.write(rdd.iterator(partition, ctxt).asInstanceOf[Iterator[_ <: Product2[Any, Any]]]) writer.stop(success = true).get	OT returned; if((returned == null) !format.nextRecord(reuse)) { output.collect(returned); }

(a) Hadoop

```

1  public class Mapper<KEYIN, VALUEIN,
2          KEYOUT, VALUEOUT> {
3      /* The default implementation
4          is the identity function. */
5      protected
6      void map(KEYIN key, VALUEIN value,
7              Context ctxt)
8      {
9          ctxt.write(KEYOUT
10             (VALU
11         }
12     public
13     void run(Context ctxt) {
14         while (ctxt.nextKeyValue())
15             map(ctxt.getCurrentKey(),
16                 ctxt.getCurrentValue(),
17                 ctxt);
18     }

```

(b) Spark

```

1  private[spark] class
2  ShuffleMapTask(partitionId: Int,
3                  partition: Partition)
4  extends Task[MapStatus] {
5
6  override
7  def runTask(ctxt: TaskContext)
8
9
10
11
12
13
14
15
16
17
18

```

(c) Flink

```

1  public class DataSourceTask<OT>
2  extends AbstractInvokable {
3
4  private
5  InputFormat<OT, InputSplit> format;
6
7  private Collector<OT> output;
8
9
10
11
12
13
14
15
16
17
18

```

Rewriting Hadoop Map/Reduce

```
fn output_side(writer, key, value) {
    output(writer, key, value)
}

fn compute_and_output(mapper, writer, split) {
    let (line, content) = split;
    let kv_pairs = hmr_map(mapper, line, content);
    for kv_pair in kv_pairs {
        let (k, v) = kv_pair;
        output_side(writer, k, v)
    }
}

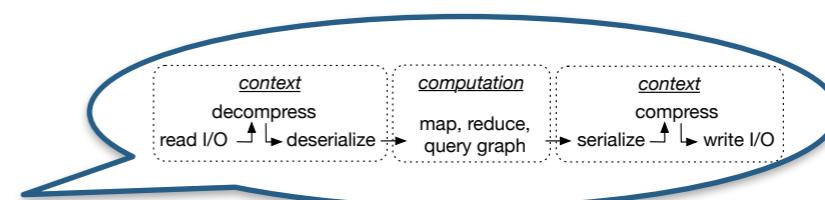
fn coarse(reader: org.apache.hadoop.mapreduce.Mapper$Context,
          mapper: org.apache.hadoop.mapreduce.Mapper,
          writer: org.apache.hadoop.mapreduce.Mapper$Context) {
    let splits = finite_generator(|| load_split(reader),
                                || has_more_data(reader));
    for split in splits {
        compute_and_output(mapper, writer, split)
    }
}
```

Rewriting Hadoop Map/Reduce

```
fn output_side(writer, key, value) {
    output(writer, key, value)
}

fn compute_and_output(mapper, writer, split) {
    let (line, content) = split;
    let kv_pairs = hmr_map(mapper, line, content);
    for kv_pair in kv_pairs {
        let (k, v) = kv_pair;
        output_side(writer, k, v)
    }
}

fn coarse(reader: org.apache.hadoop.mapreduce.Mapper$Context,
          mapper: org.apache.hadoop.mapreduce.Mapper,
          writer: org.apache.hadoop.mapreduce.Mapper$Context) {
    let splits = finite_generator(|| load_split(reader),
                                || has_more_data(reader));
    for split in splits {
        compute_and_output(mapper, writer, split)
    }
}
```

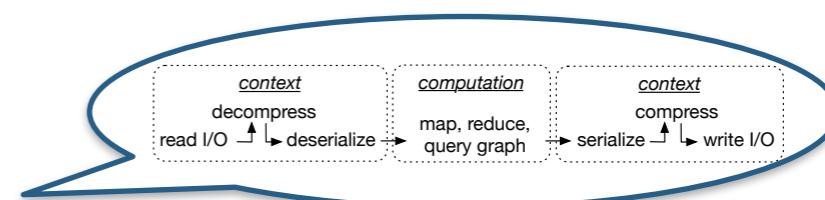


Rewriting Hadoop Map/Reduce

```
fn output_side(writer, key, value) {
    output(writer, key, value)
}

fn compute_and_output(mapper, writer, split) {
    let (line, content) = split;
    let kv_pairs = hmr_map(mapper, line, content);
    for kv_pair in kv_pairs {
        let (k, v) = kv_pair;
        output_side(writer, k, v)
    }
}

fn coarse(reader: org.apache.hadoop.mapreduce.Mapper$Context,
          mapper: org.apache.hadoop.mapreduce.Mapper,
          writer: org.apache.hadoop.mapreduce.Mapper$Context) {
    let splits = finite_generator(|| load_split(reader),
                                || has_more_data(reader));
    for split in splits {
        compute_and_output(mapper, writer, split)
    }
}
```

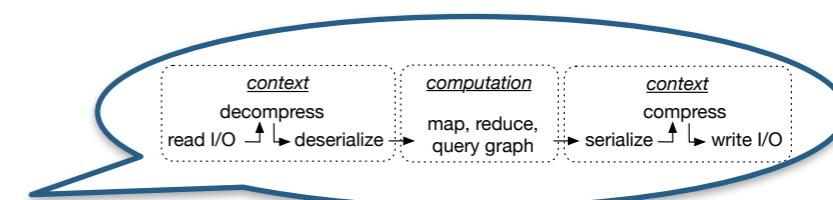


Rewriting Hadoop Map/Reduce

```
fn output_side(writer, key, value) {
    output(writer, key, value)
}
    writer.write(key, value)

fn compute_and_output(mapper, writer, split) {
    let (line, content) = split;
    let kv_pairs = hmr_map(mapper, line, content);
    for kv_pair in kv_pairs {
        let (k, v) = kv_pair;
        output_side(writer, k, v)
    }
}

fn coarse(reader: org.apache.hadoop.mapreduce.Mapper$Context,
          mapper: org.apache.hadoop.mapreduce.Mapper,
          writer: org.apache.hadoop.mapreduce.Mapper$Context) {
    let splits = finite_generator(|| load_split(reader),
                                || has_more_data(reader));
    for split in splits {
        compute_and_output(mapper, writer, split)
    }
}
```



Rewriting Hadoop Map/Reduce

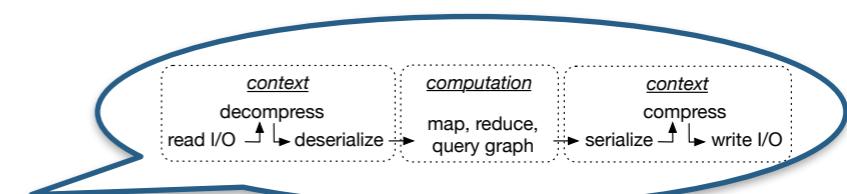
```
fn output_side(writer, key, value) {
    output(writer, key, value)
}
    writer.write(key, value)

fn compute_and_output(mapper, writer, split) {
    let (line, content) = split;
    let kv_pairs = hmr_map(mapper, line, content);
    for kv_pair in kv_pairs {
        let (k, v) = kv_pair;
        output_side(writer, k, v)
    }
}

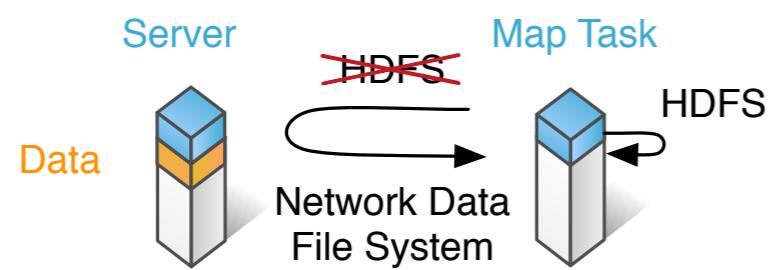
fn coarse(reader: org.apache.hadoop.mapreduce.Mapper$Context,
          mapper: org.apache.hadoop.mapreduce.Mapper,
          writer: org.apache.hadoop.mapreduce.Mapper$Context) {
    let splits = finite_generator(|| load_split(reader),
                                || has_more_data(reader));
    for split in splits {
        compute_and_output(mapper, writer, split)
    }
}
```

4 variants:

- Coarse (C)
- Coarse Input Fine Output (CIFO)
- Fine Input Coarse Output (FICO)
- Fine (F)



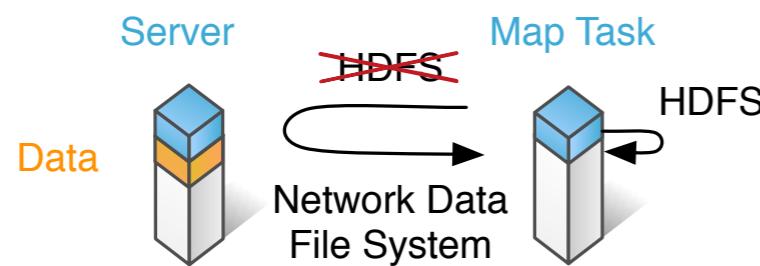
Evaluation



- SequenceFile
- JSON
- Snappy, LZO
- TPC-H Parts table

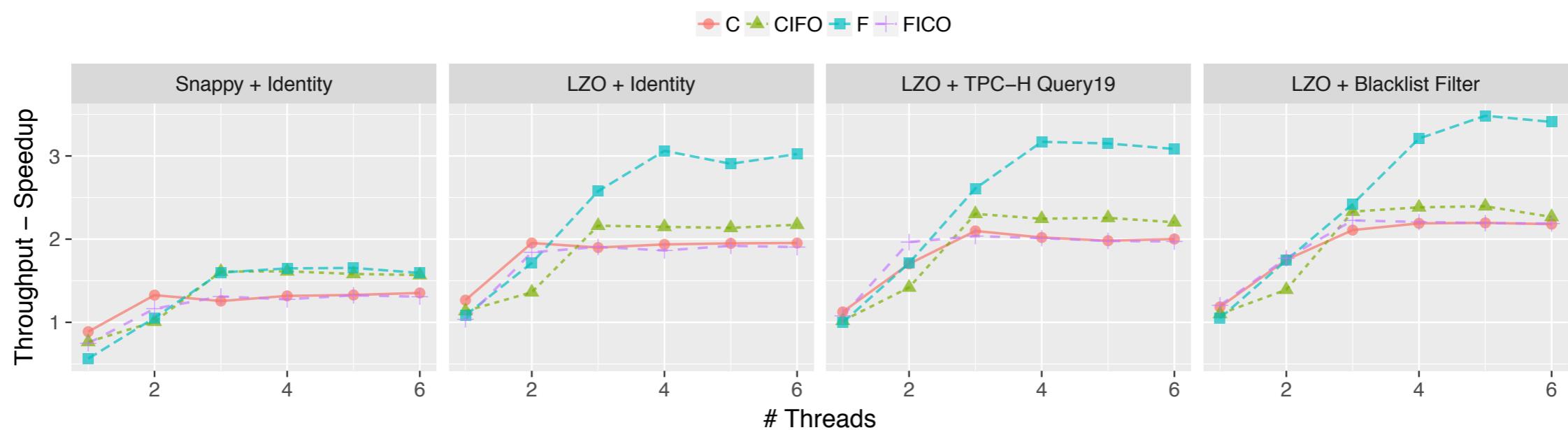
Almost the same setup as for
a reduce task!

Throughput Analysis



- SequenceFile
- JSON
- Snappy, LZO
- TPC-H Parts table

Almost the same setup as for
a reduce task!



Speedup of up to 3.5x for compute-intensive configurations!

Programming Model/
Language



Compiler



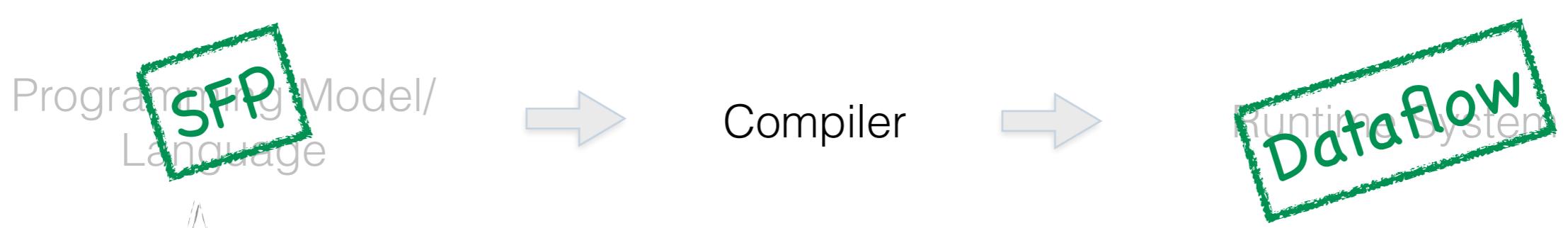
Runtime System
Dataflow

SFP:

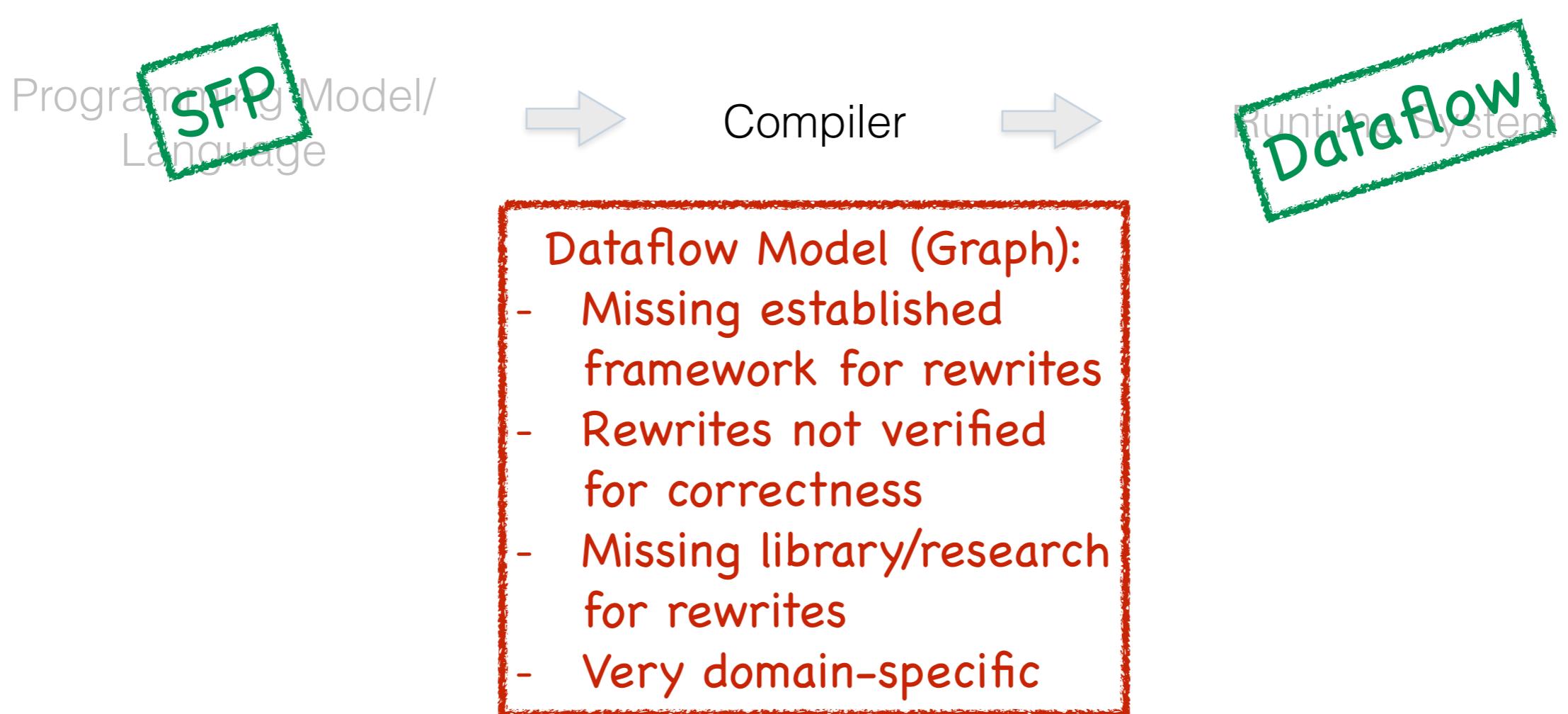
- Variables
- Stateful Functions
- Composition
- Algorithms
- Control Flow
- SMap

Sebastian Ertel, Christof Fetzer, and Pascal Felber. 2015. Ohua: Implicit Dataflow Programming for Concurrent Systems. In Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ '15). ACM.

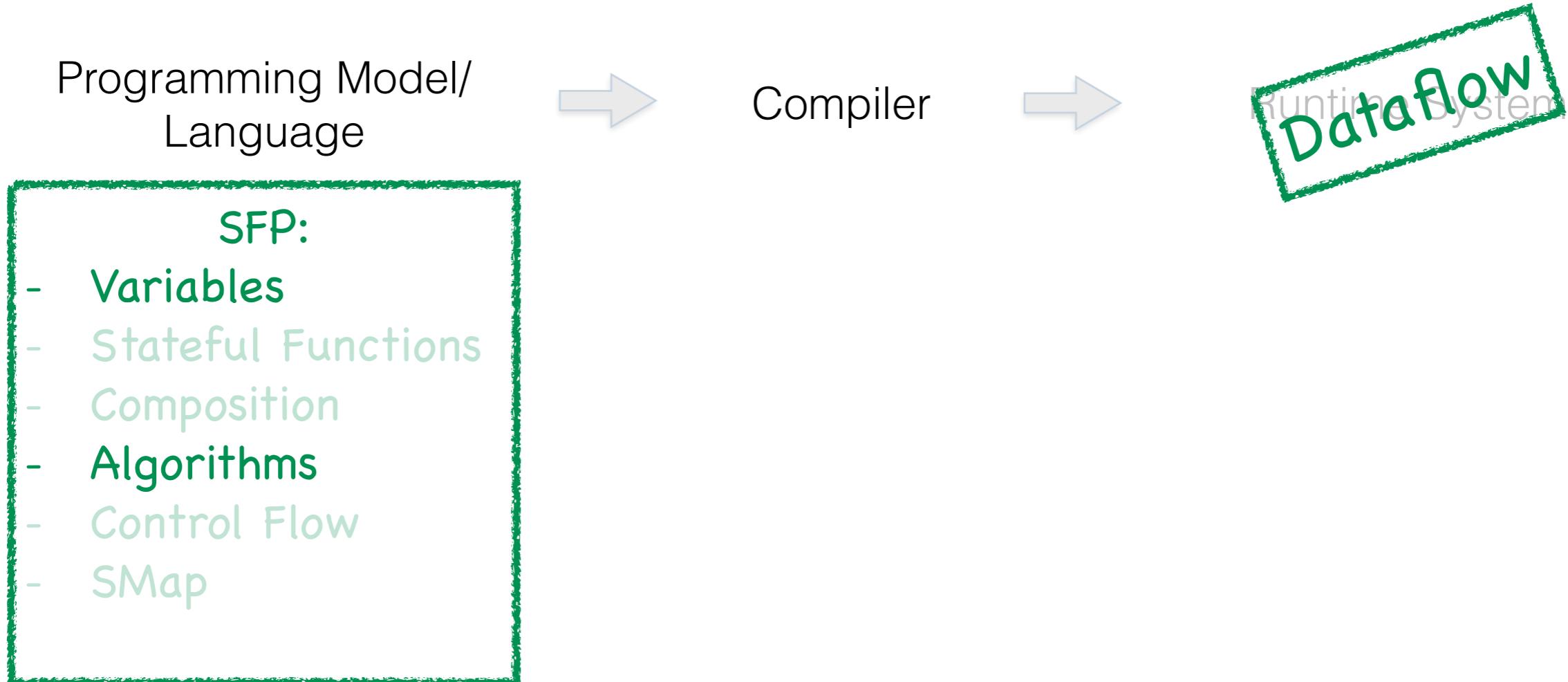
Sebastian Ertel, Justus Adam, and Jeronimo Castrillon. 2018. Supporting Fine-grained Dataflow Parallelism in Big Data Systems. In Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'18). ACM.



- No concurrency abstractions.
- Practicality:
 - Integration into existing programming models (e.g., OOP)/ languages.
 - Gradual switch/reuse of existing code base.

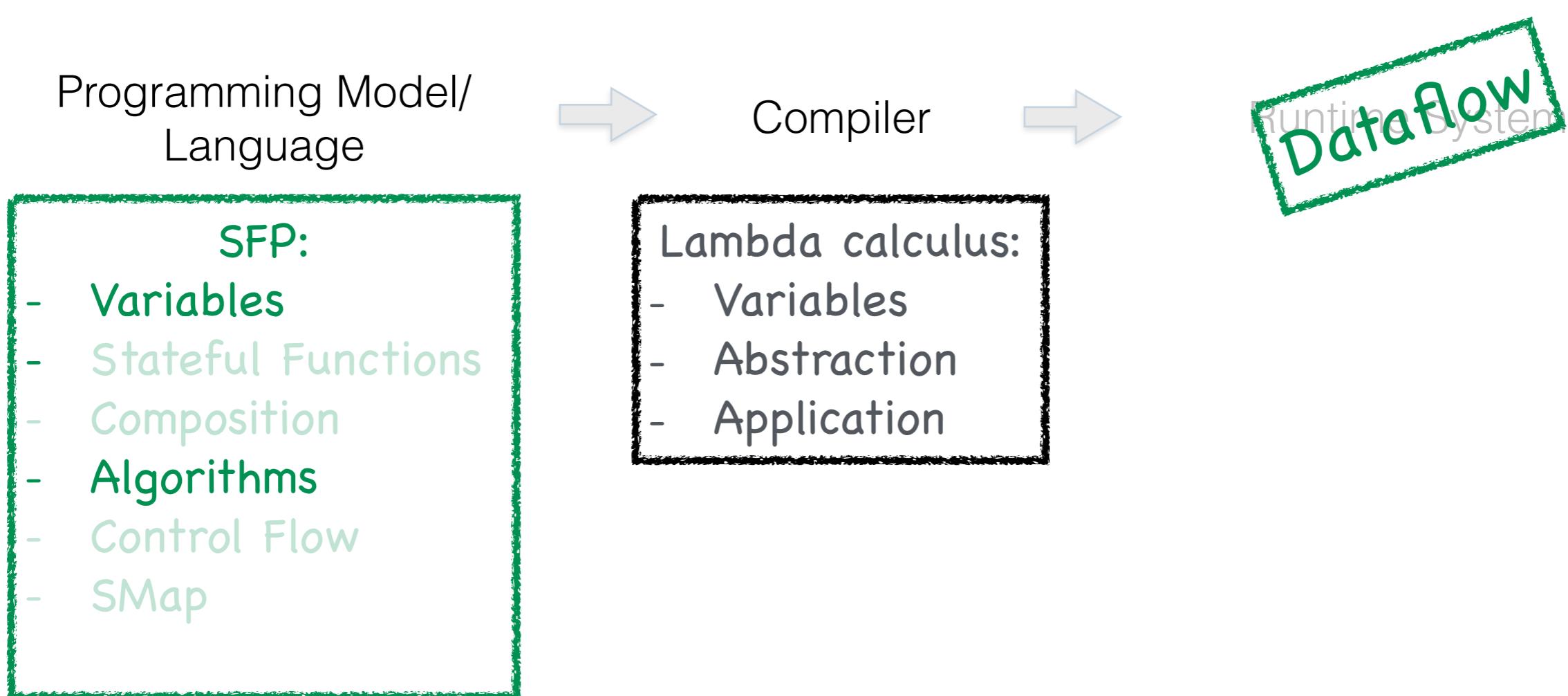


PL/Compiler Co-Design



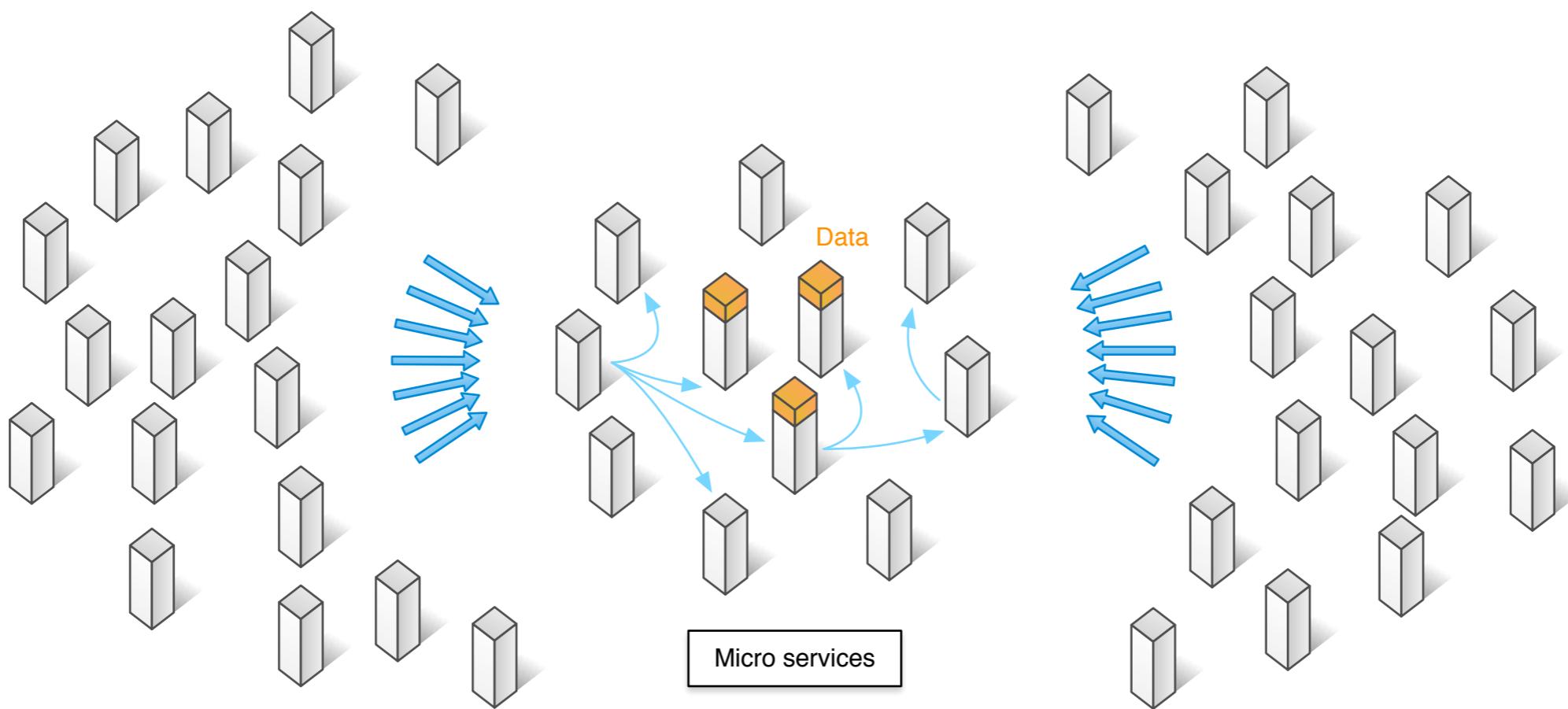
J. Backus. Can programming be liberated from the von neumann style?:A functional style and its algebra of programs. Commun.ACM,1978.
Barendregt, Hendrik Pieter. "Functional programming and lambda calculus." *Formal models and semantics*. Elsevier, 1990.

PL/Compiler Co-Design



J. Backus. Can programming be liberated from the von neumann style?:A functional style and its algebra of programs. Commun.ACM,1978.
Barendregt, Hendrik Pieter. "Functional programming and lambda calculus." *Formal models and semantics*. Elsevier, 1990.

Microservice Architectures

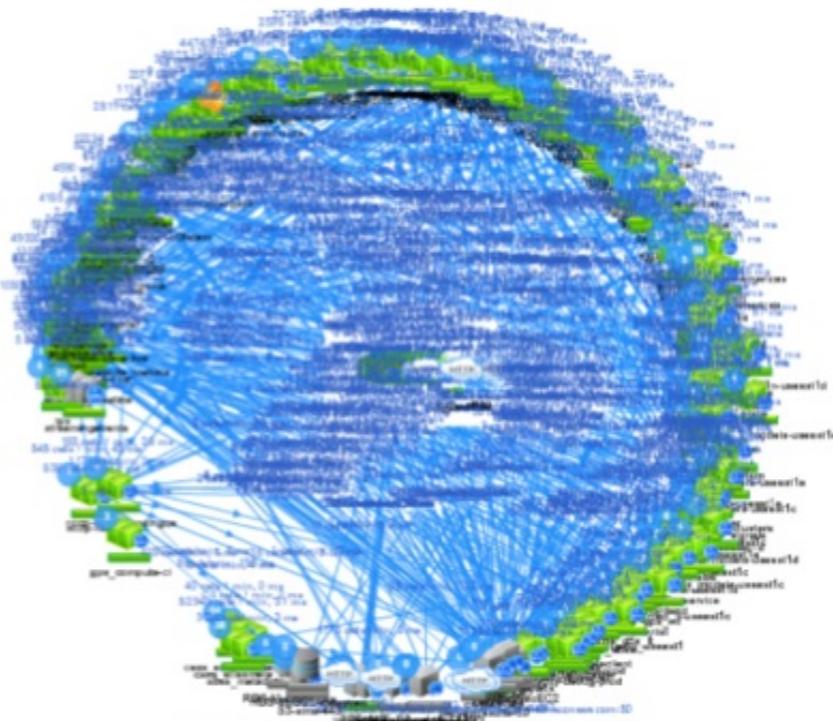


Redefining the Death Star

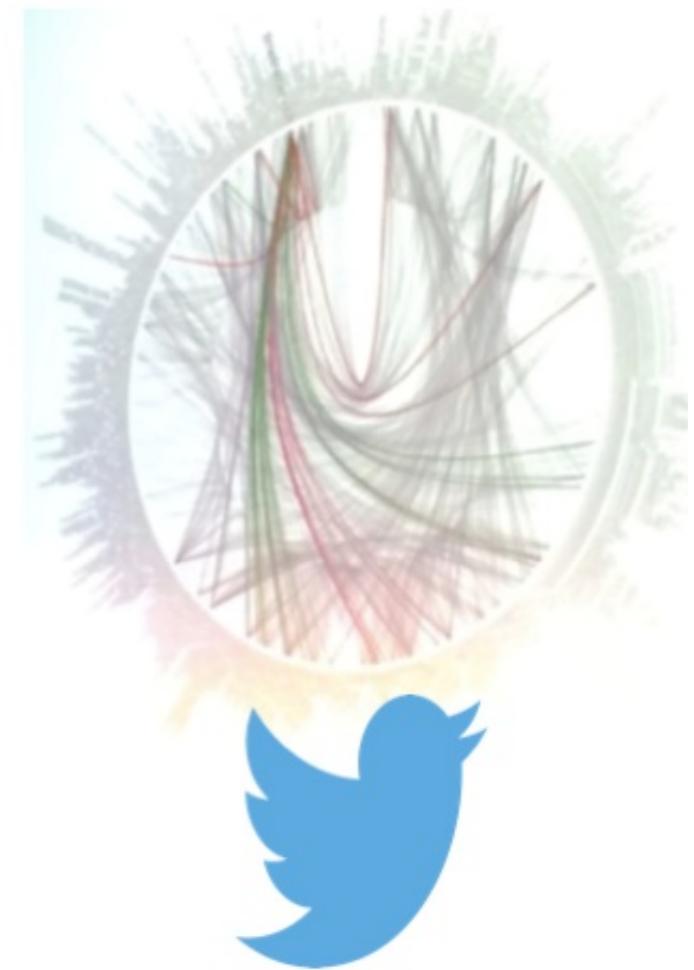
450 microservices



500+ microservices



500+ microservices



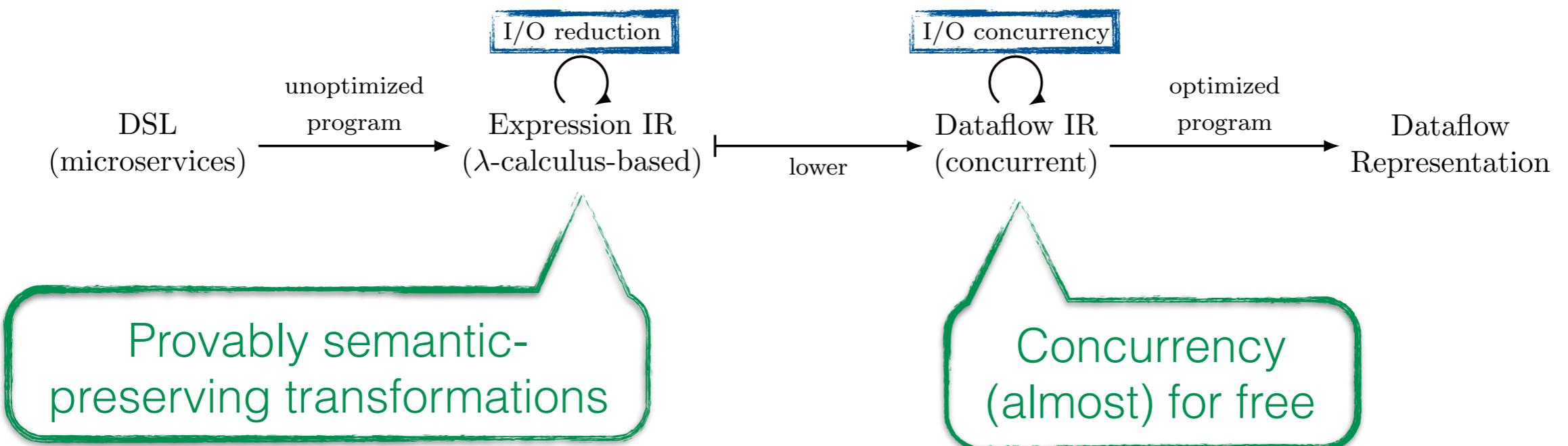
Source:

Netflix: <http://www.slideshare.net/BruceWong3/the-case-for-chaos>

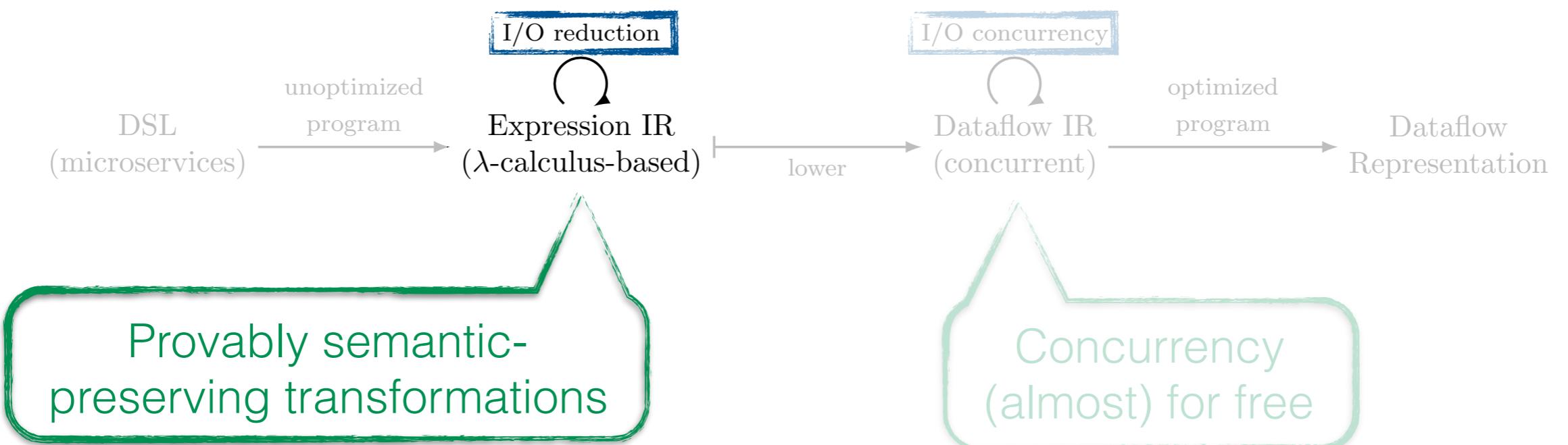
Twitter: <https://twitter.com/adrianco/status/441883572618948608>

Hail-o: <https://sudo.hailoapp.com/services/2015/03/09/journey-into-a-microservice-world-part-3/>

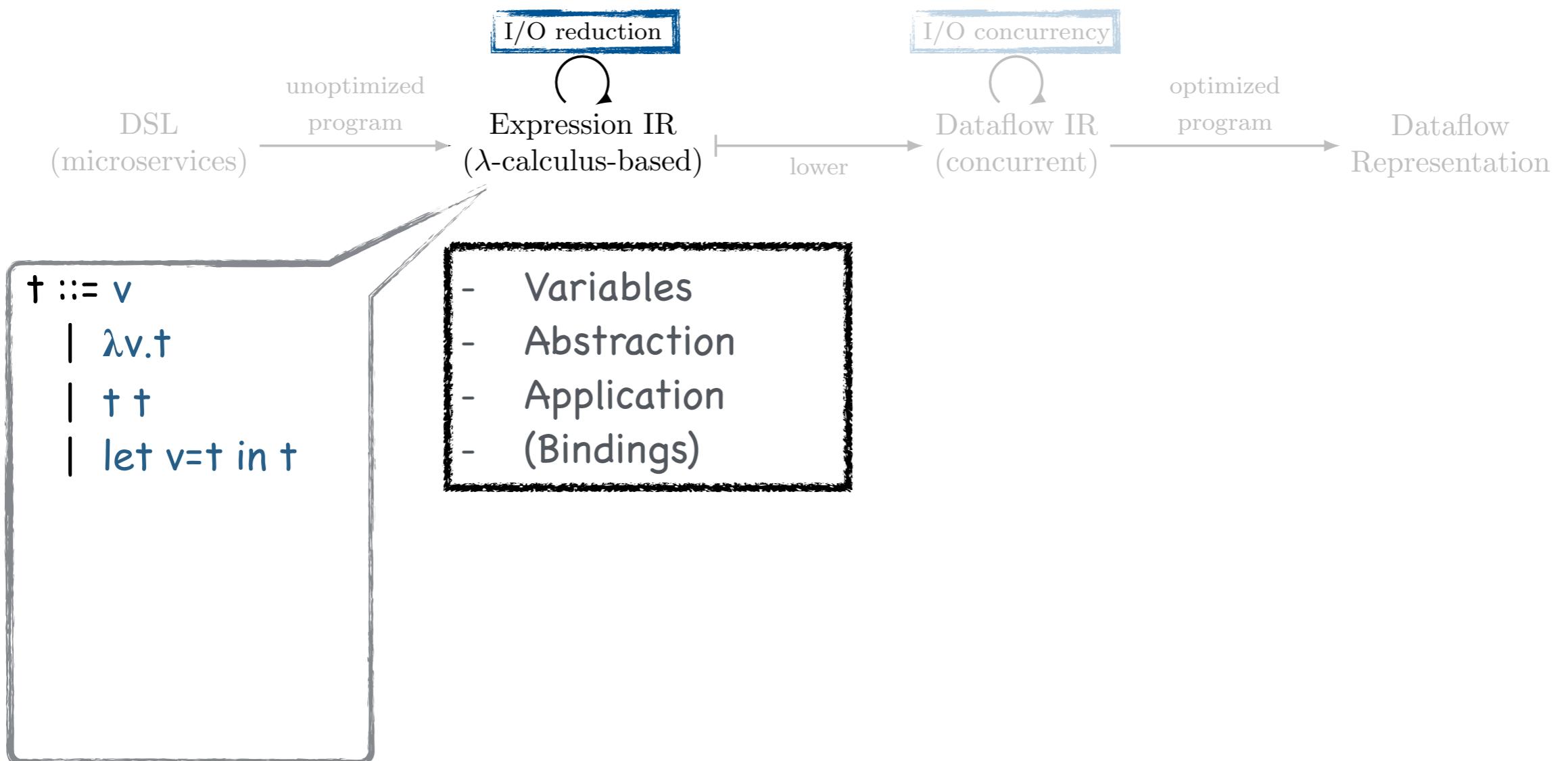
Compiler



Compiler



Foundation: Call-by-need Lambda Calculus

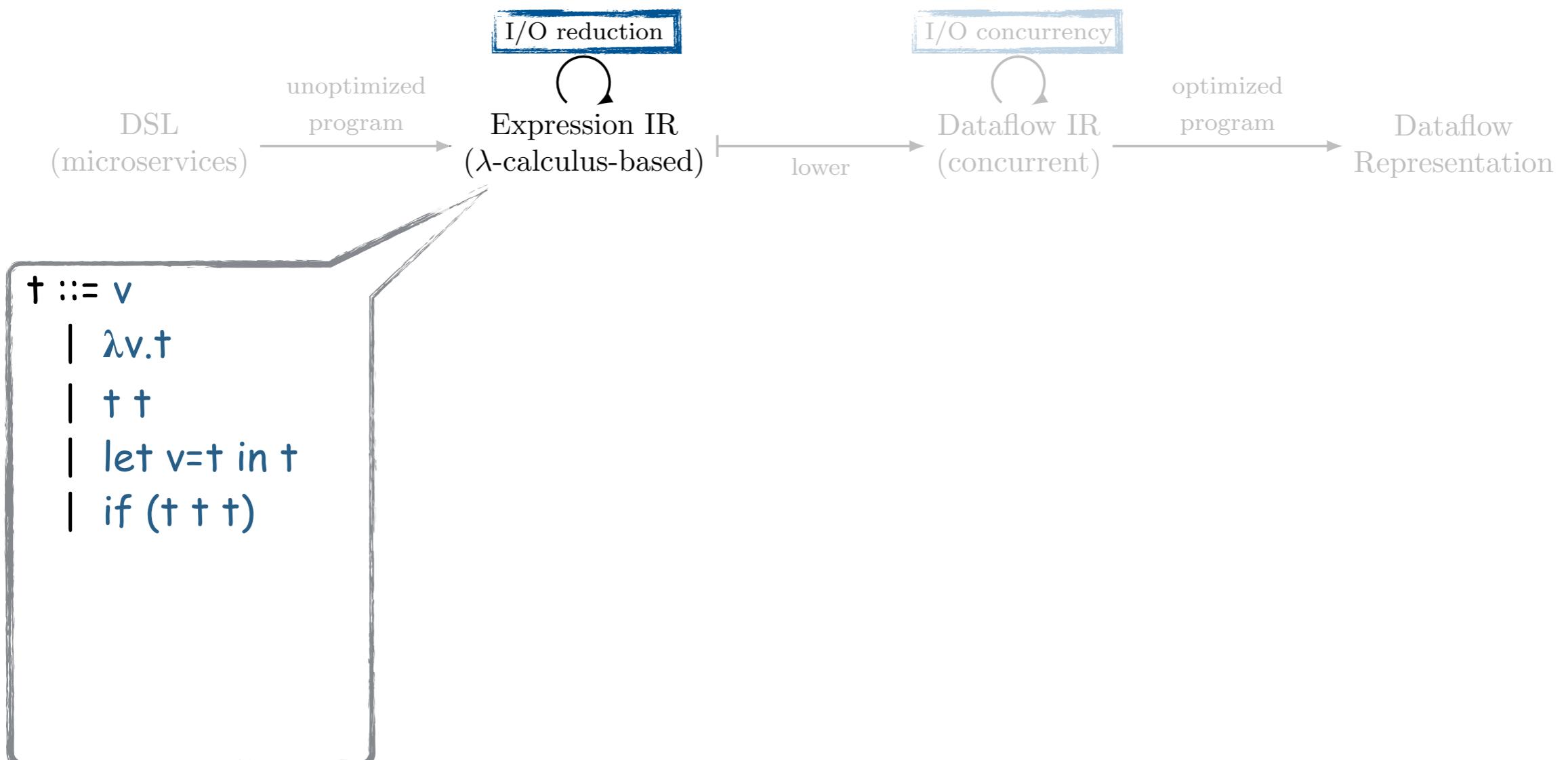


Ariola ZM, Felleisen M. The call-by-need lambda calculus. Journal of functional programming. 1997.

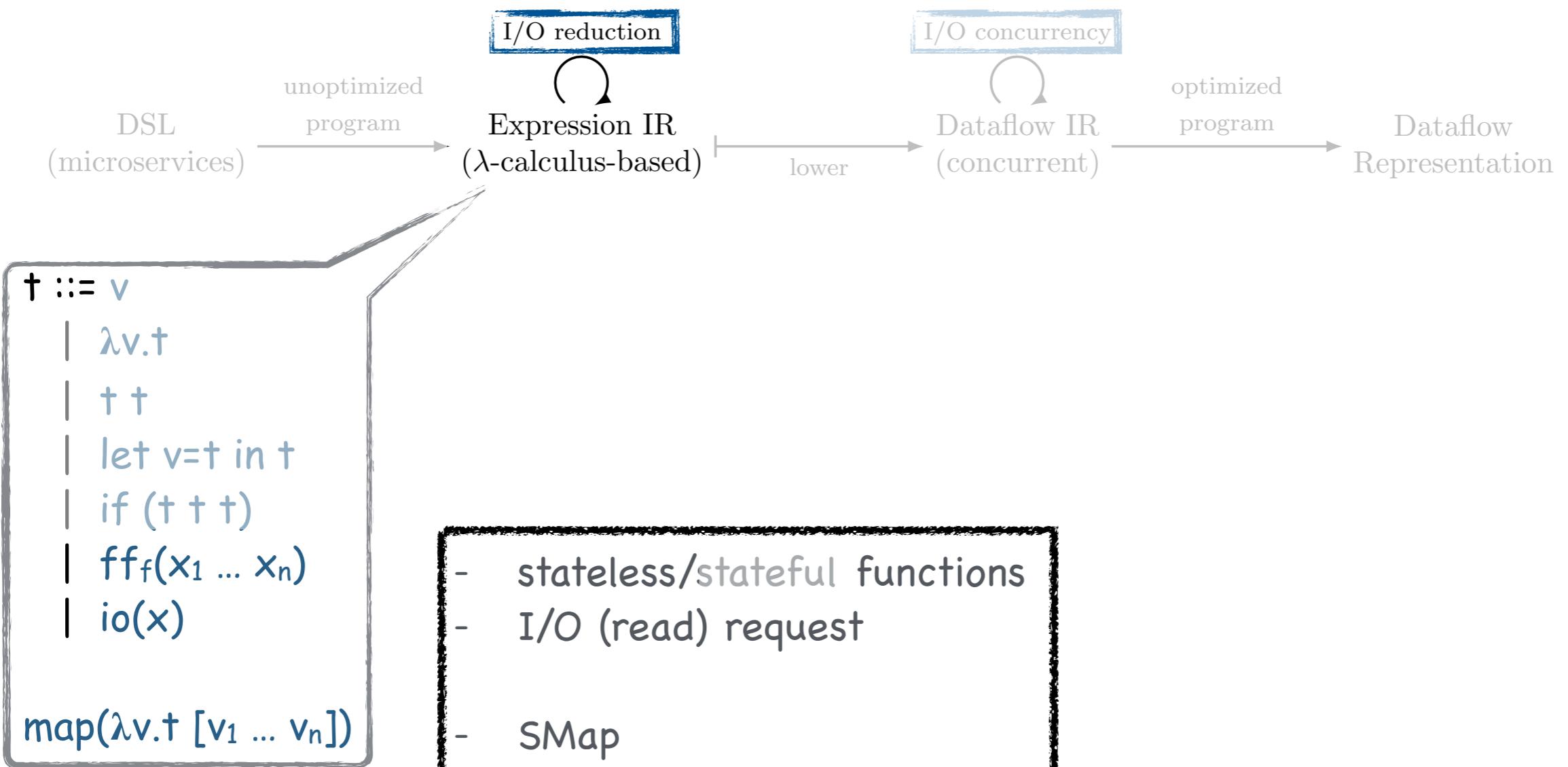
Maraist J, Odersky M, Wadler P. The call-by-need lambda calculus. Journal of functional programming. 1998.

Chang S, Felleisen M. The call-by-need lambda calculus, revisited. In European Symposium on Programming 2012. Springer.

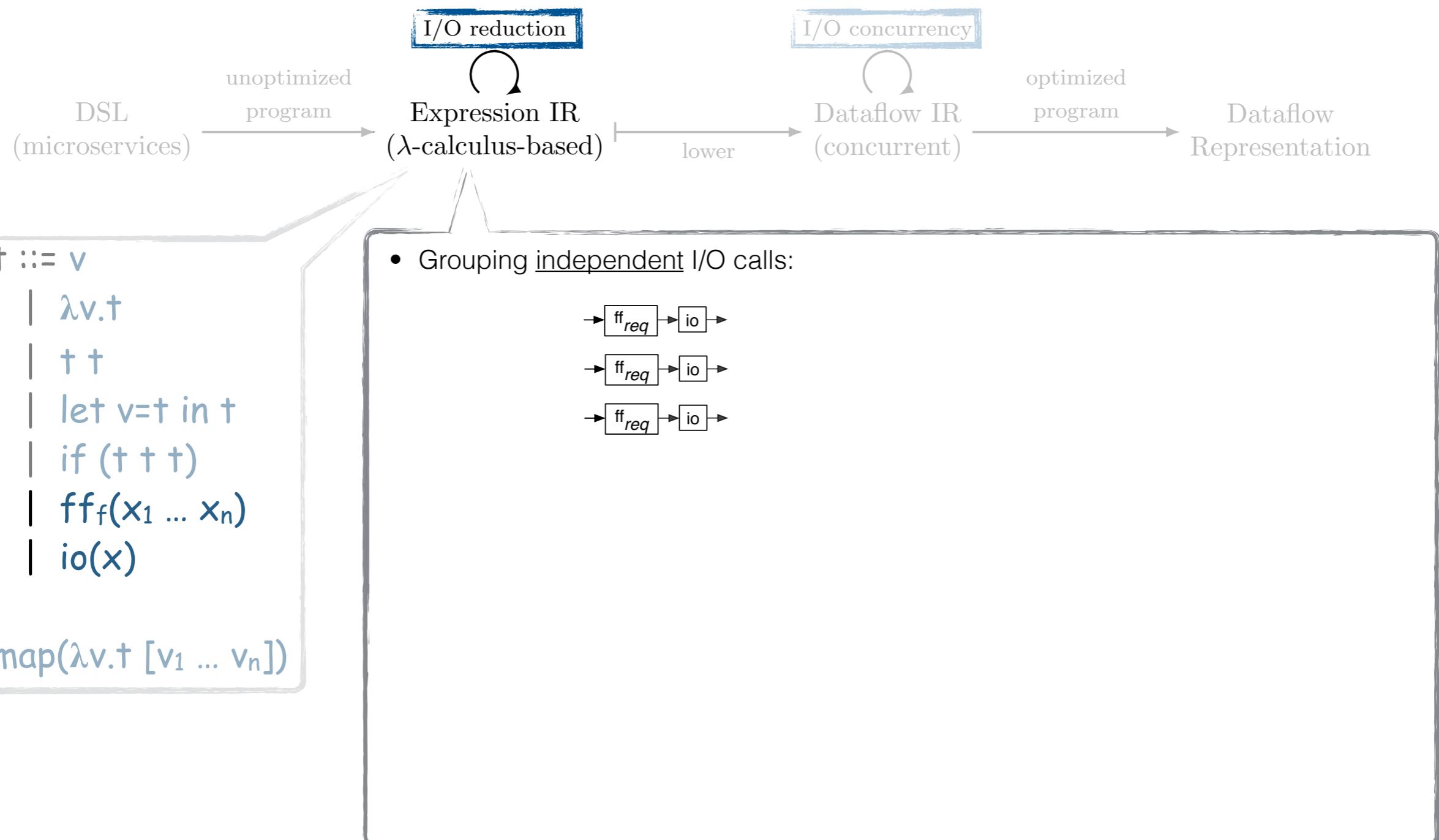
Foundation: Call-by-need Lambda Calculus



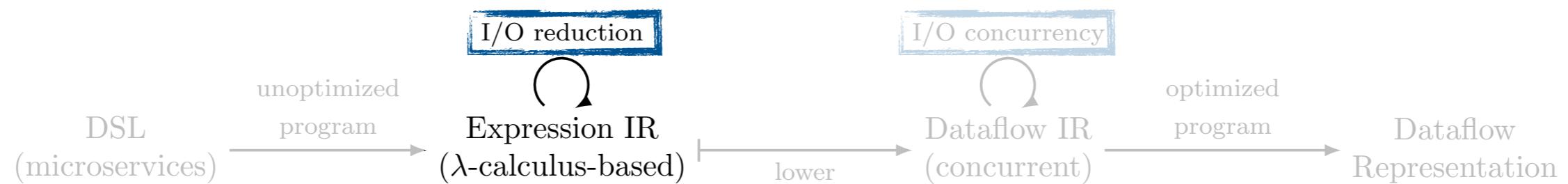
Domain-specific Aspects



I/O Transformations



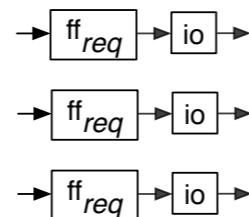
I/O Transformations



$\dagger ::= v$
 | $\lambda v. \dagger$
 | $\dagger \dagger$
 | $\text{let } v = \dagger \text{ in } \dagger$
 | $\text{if } (\dagger \dagger \dagger)$
 | $\text{ff}_f(x_1 \dots x_n)$
 | $\text{io}(x)$

 $\text{map}(\lambda v. \dagger [v_1 \dots v_n])$

- Grouping independent I/O calls:



Remember SMap

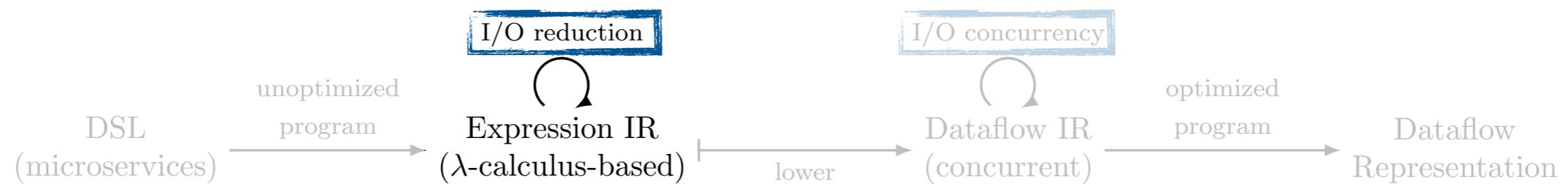
$$\begin{aligned} (s'_g, x_1) &= g(s_g, a_1) \\ (s''_g, x_2) &= g(s'_g, a_2) \\ (s'_h, b_1) &= h(s_h, x_1) \\ (s''_h, b_2) &= h(s'_h, x_2) \end{aligned}$$



$$\begin{aligned} (s'_g, x_1) &= g(s_g, a_1) \\ (s'_h, b_1) &= h(s_h, x_1) \\ (s''_g, x_2) &= g(s'_g, a_2) \\ (s''_h, b_2) &= h(s'_h, x_2) \end{aligned}$$

pipeline parallel

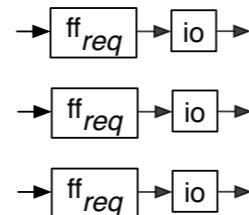
I/O Transformations



- † ::= v
- | $\lambda v. t$
- | $t \ t$
- | $\text{let } v=t \text{ in } t$
- | $\text{if } (t \ t \ t)$
- | $\text{ff}_f(x_1 \dots x_n)$
- | $\text{io}(x)$

$\text{map}(\lambda v. t [v_1 \dots v_n])$

- Grouping independent I/O calls:

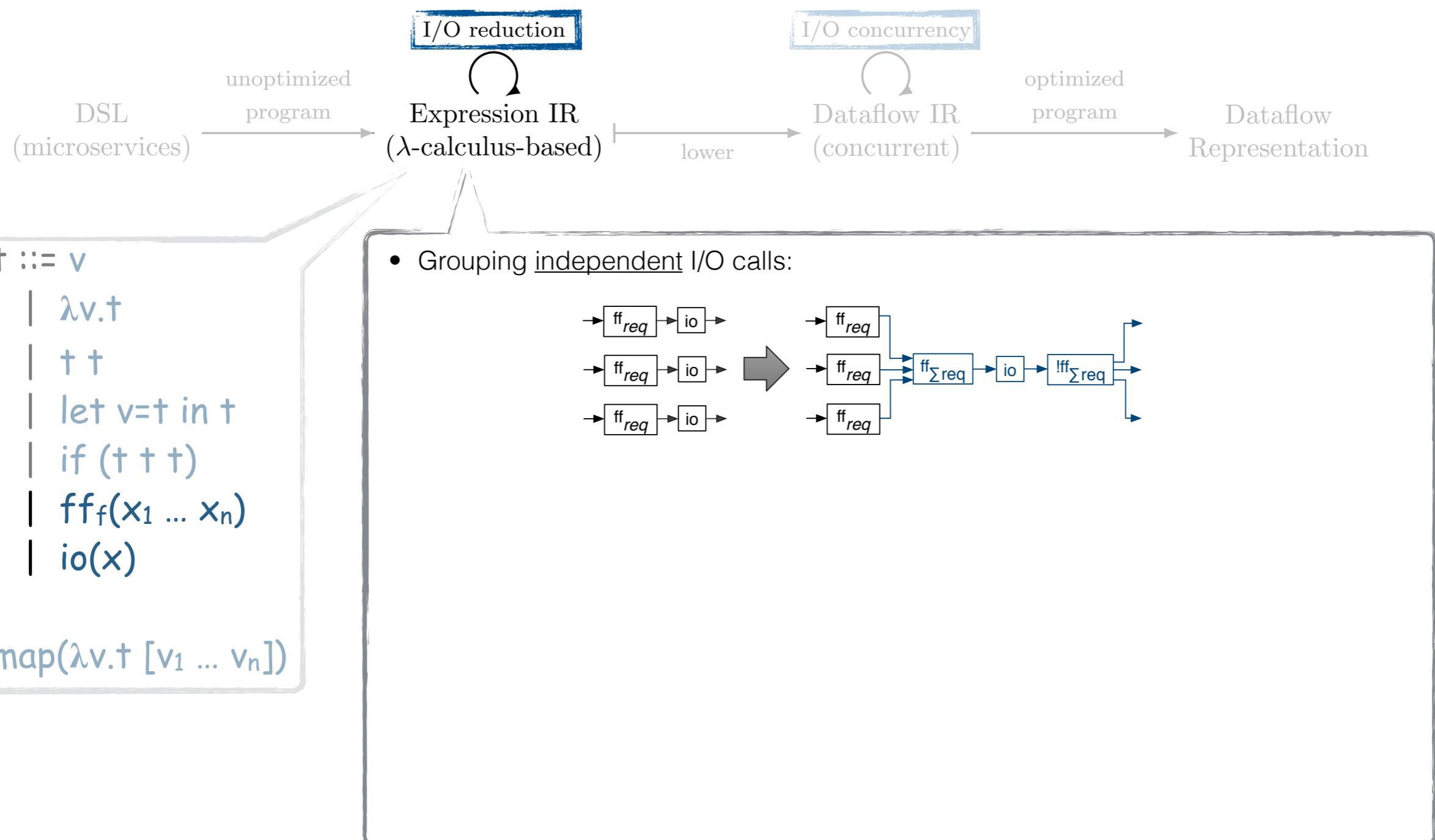


Let-floating

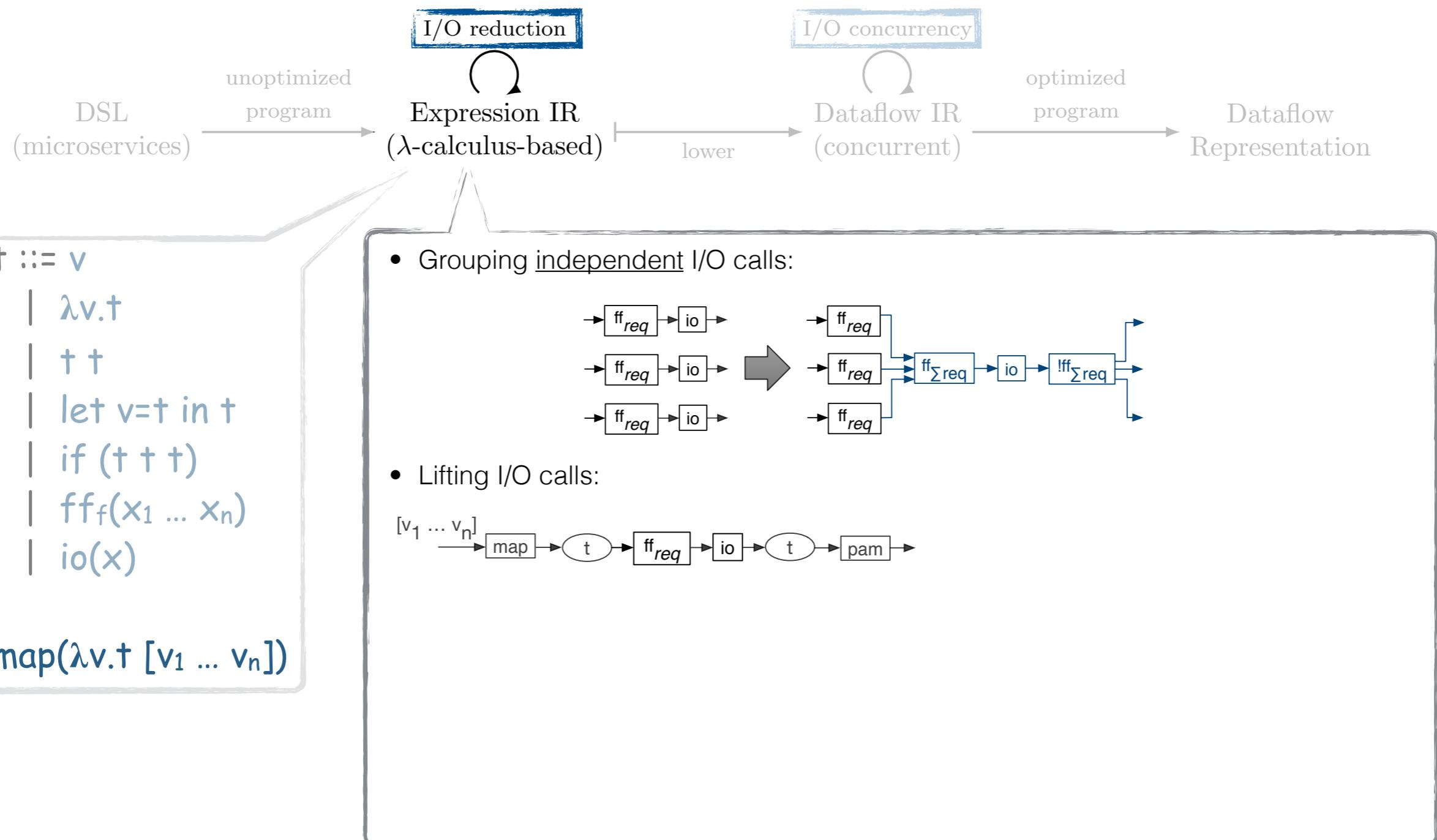
$$\begin{aligned}
 \text{smap}(f, [a_1, a_2]) = & \text{let } (s'_g, x_1) = g(s_g, a_1) \text{ in} \\
 & \text{let } (s''_g, x_2) = g(s'_g, a_2) \text{ in} \\
 & \text{let } (s'_h, b_1) = h(s_h, x_1) \text{ in} \\
 & \text{let } (s''_h, b_2) = h(s'_h, x_2) \text{ in} \\
 & [b_1, b_2]
 \end{aligned}
 \quad = \quad
 \begin{aligned}
 & \text{let } (s'_g, x_1) = g(s_g, a_1) \text{ in} \\
 & \cancel{\text{let } (s'_h, b_1) = h(s_h, x_1) \text{ in}} \\
 & \cancel{\text{let } (s''_g, x_2) = g(s'_g, a_2) \text{ in}} \\
 & \cancel{\text{let } (s''_h, b_2) = h(s'_h, x_2) \text{ in}} \\
 & [b_1, b_2]
 \end{aligned}$$

pipeline parallel

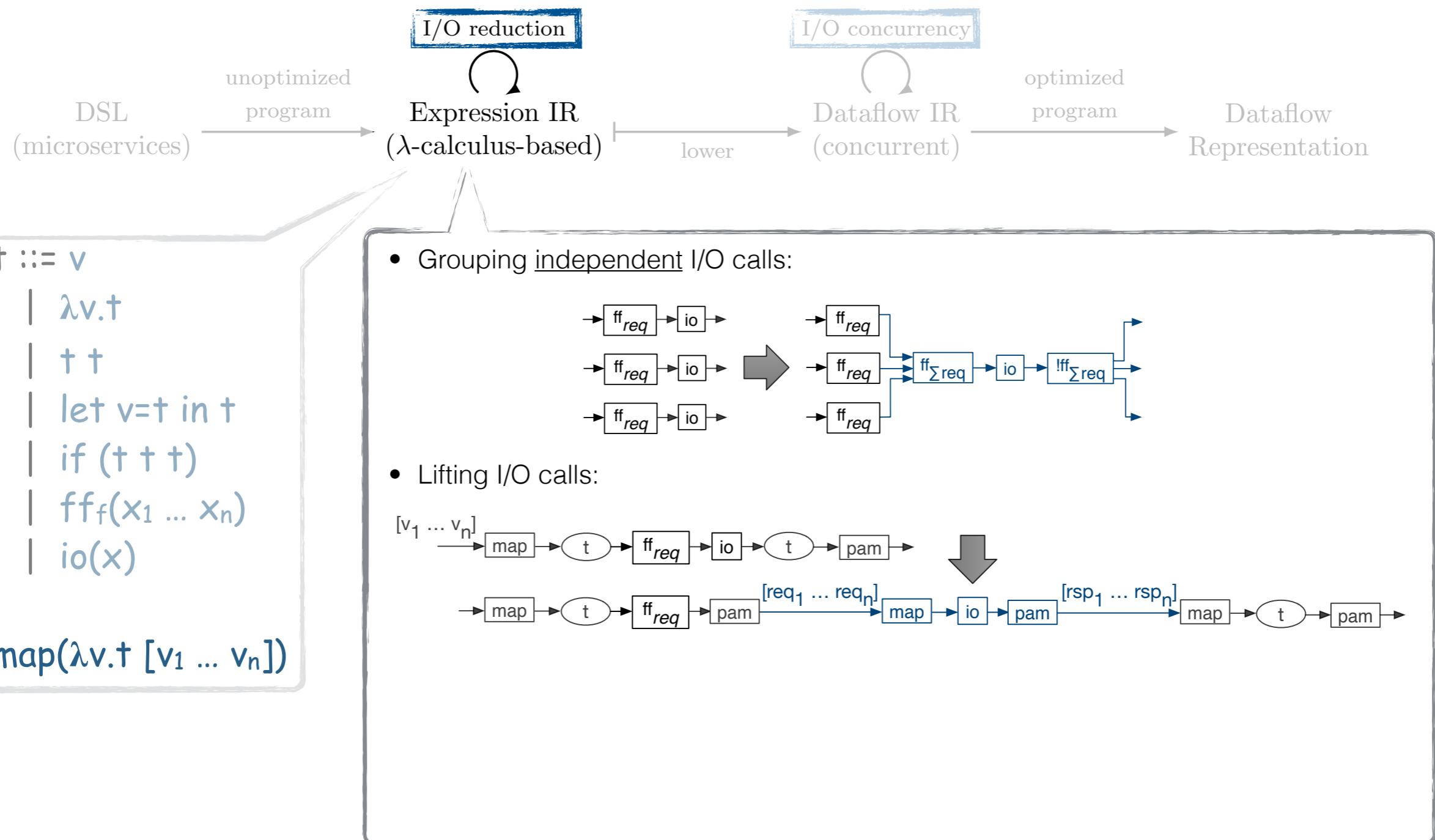
I/O Transformations



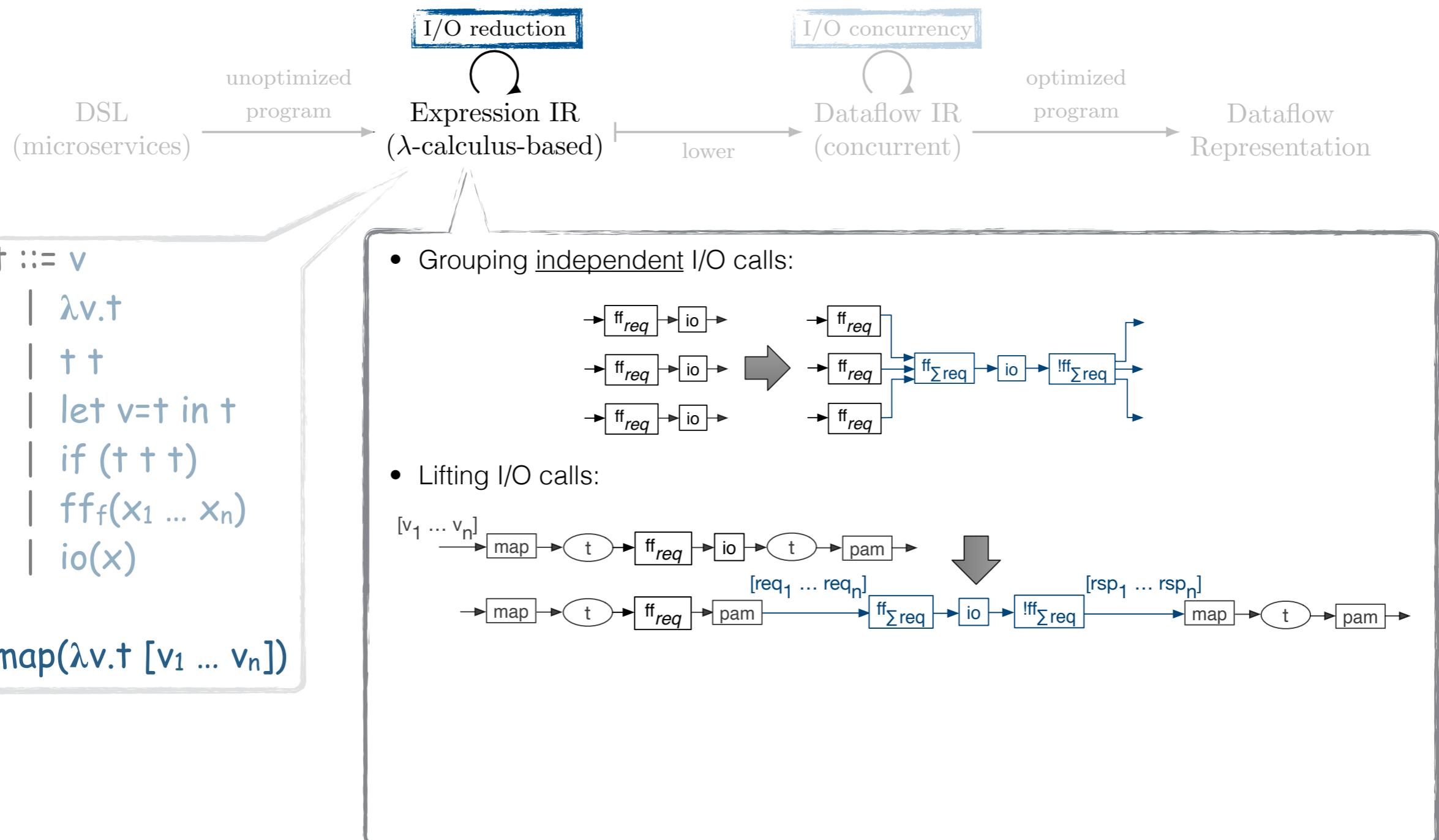
I/O Transformations



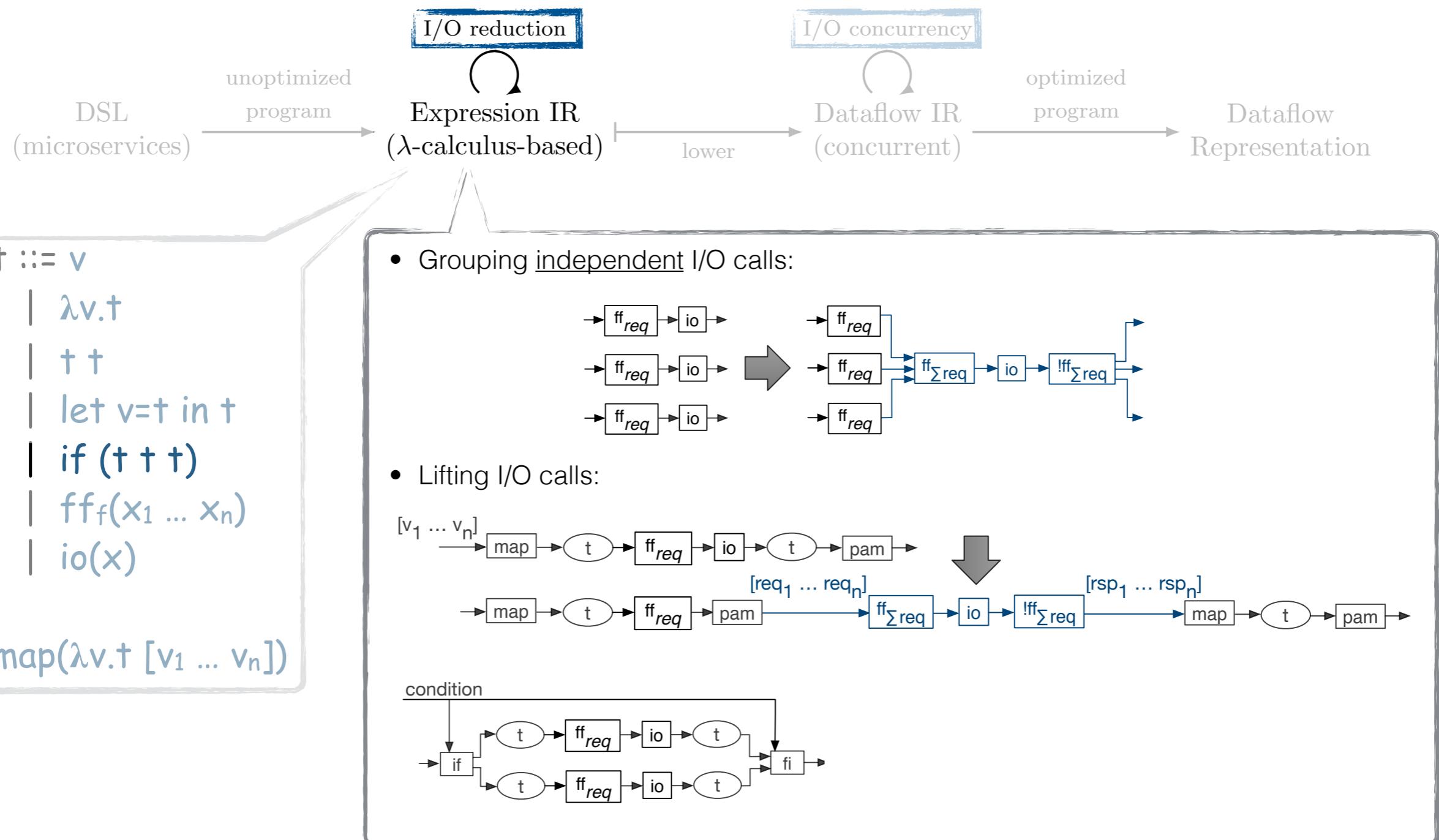
I/O Transformations



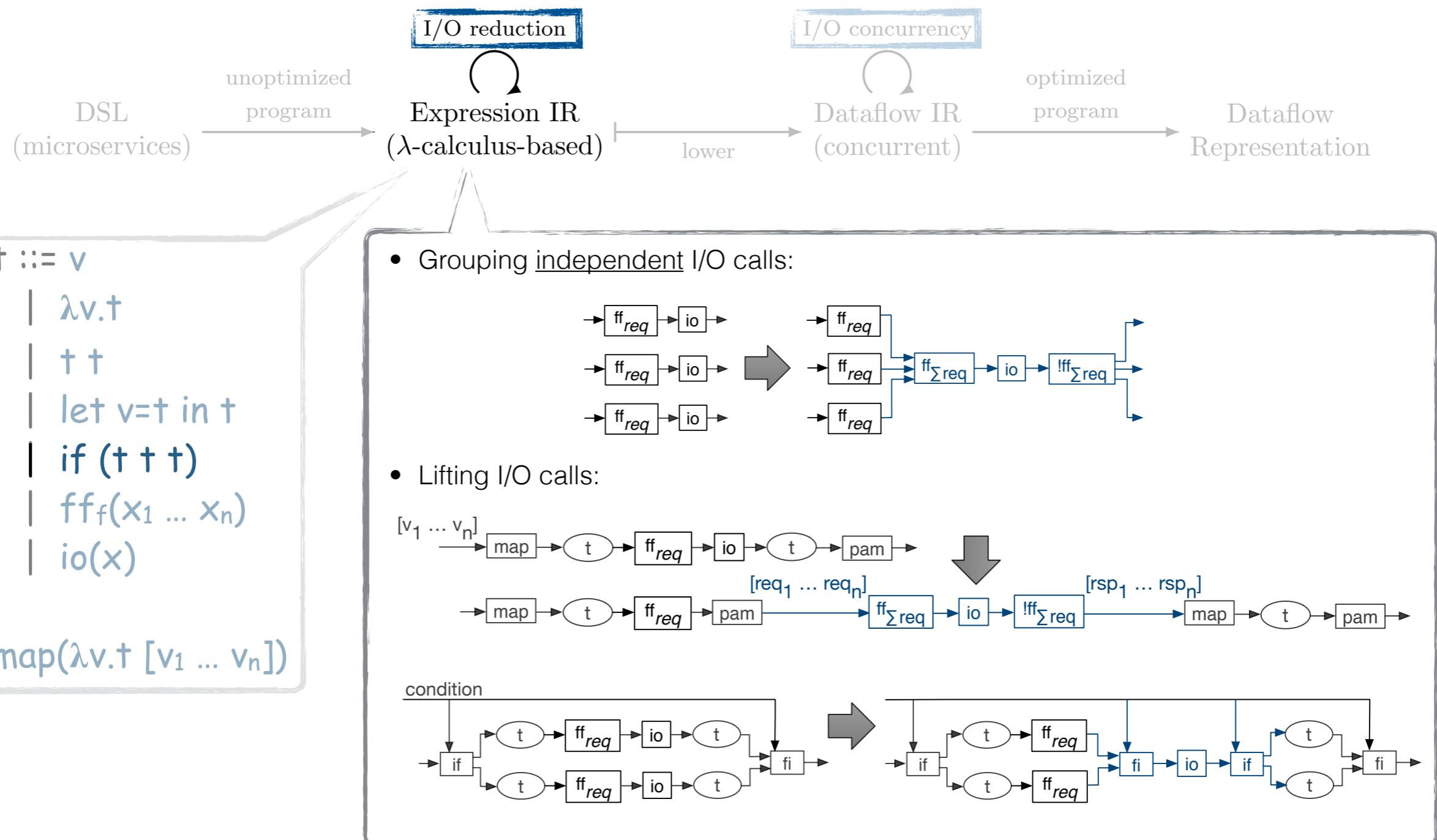
I/O Transformations



I/O Transformations

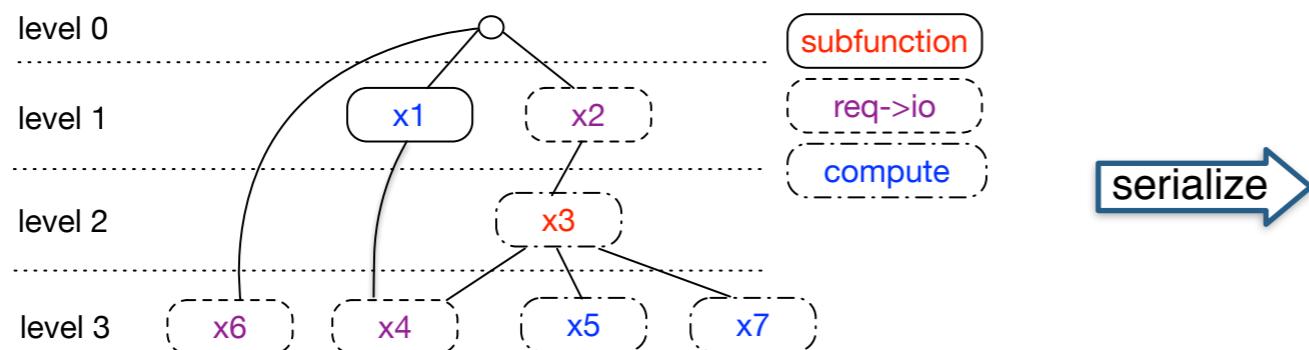


I/O Transformations



Evaluation

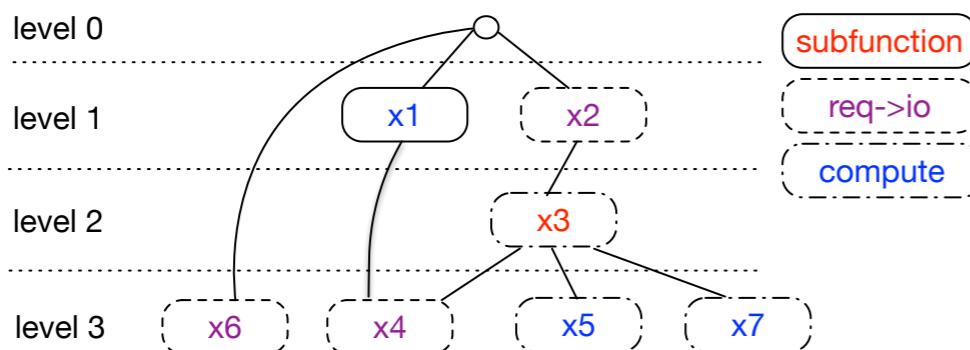
Level-Graphs:



Programs:

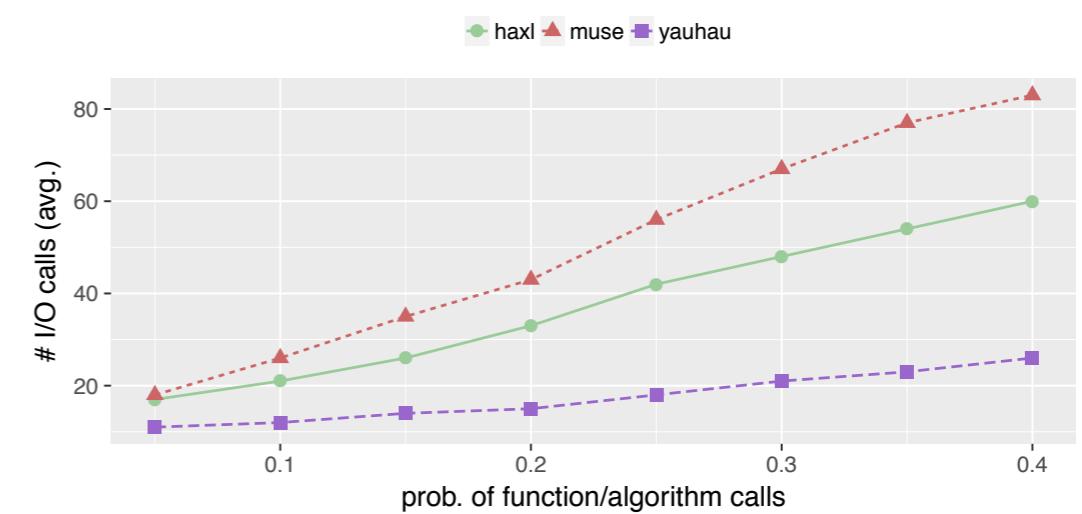
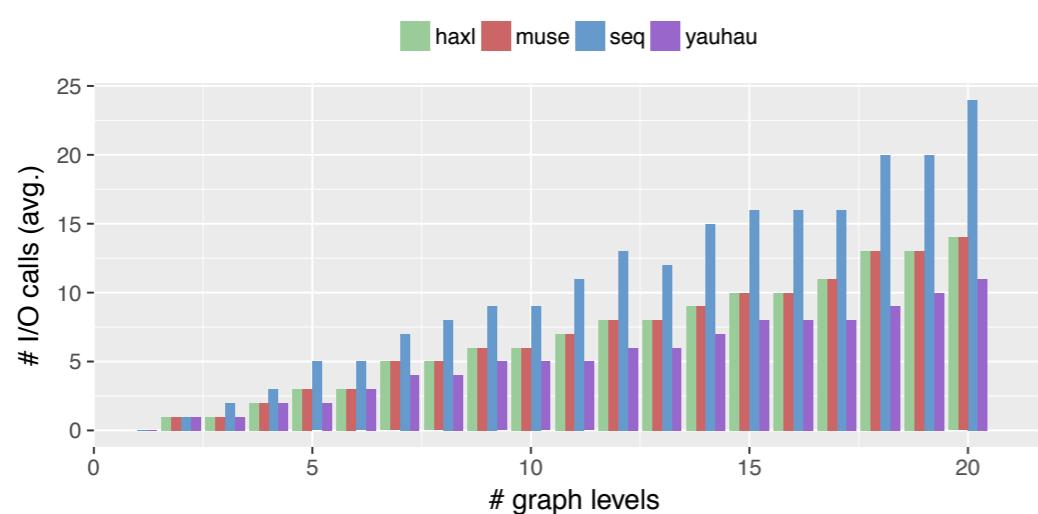
Evaluation

Level-Graphs:



Programs:

- Haxl
- Muse
- Seq
- Yauhau



Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is no fork: an abstraction for efficient, concurrent, and concise data access. ICFP '14.

Alexey Kachayev. 2015. Reinventing Haxl: Efficient, Concurrent and Concise Data Access. Presentation at EuroClojure 2015.

Programming Model/
Language



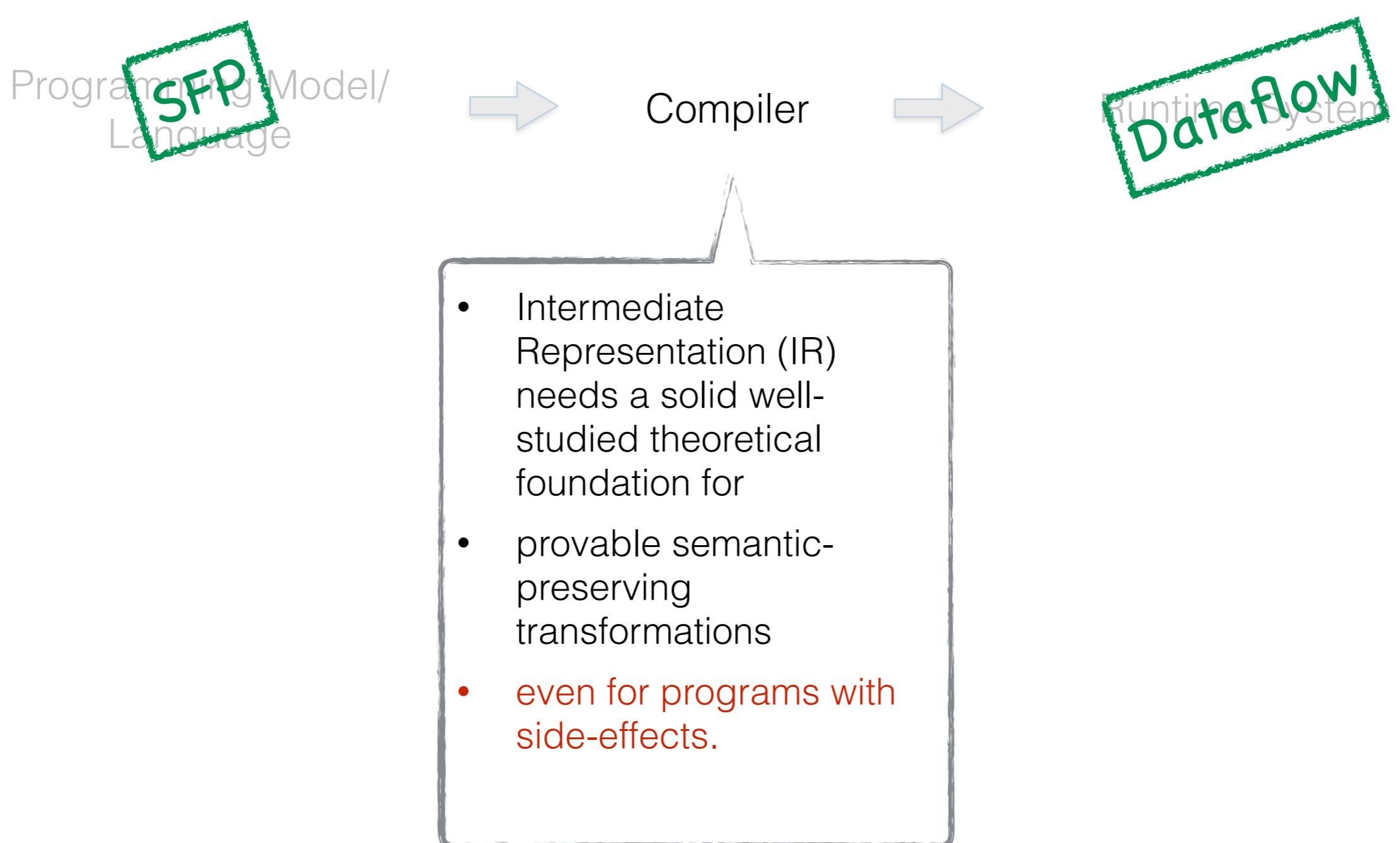
Compiler



Runtime System
Dataflow

Lambda calculus:
- Variables
- Abstraction
- Application

Sebastian Ertel, Andrés Goens, Justus Adam, and Jeronimo Castrillon. 2018. Compiling for concise code and efficient I/O. In Proceedings of the 27th International Conference on Compiler Construction (CC 2018). ACM.



Programming Model/
Language



Compiler



Runtime System
Dataflow

SFP:

- Variables
- Stateful Functions
- Composition
- Algorithms
- Conditionals
- SMap

Lambda calculus:

- Variables
- Abstraction
- Application
- Combinators

Stateful Functions



```
class FileLoad extends Filter {  
  
    private Channel<String> in, out;  
  
    public void work() {  
        // explicit channel control  
        String resource = in.take();  
        String contents = load(resource);  
        out.put(contents);  
    }  
  
    private Map<String, String> cache = new HashMap<>();  
  
    private String load(String resource){  
        String content = null;  
        if(!cache.containsKey(resource)){  
            content = new String(  
                Files.readAllBytes(Paths.get(resource))  
            );  
            cache.put(resource,content);  
            // cache eviction emitted for brevity  
        } else {  
            content = cache.get(resource);  
        }  
        return content;  
    }  
}
```

State encapsulation



```
class FileLoad {  
    Map<String, String> cache = new HashMap<>();  
  
    String load(String resource) {  
        String contents = null;  
        // load file data from disk or cache (omitted)  
        return contents;  
    }  
}  
  
struct FileLoad {  
    cache : HashMap,  
};  
  
impl FileLoad {  
    fn load(&self, resource:String) -> String {  
        let contents : String = {  
            // load file data from disk or cache (omitted)  
        };  
        contents  
    }  
}  
  
char* load(char* resource) {  
    static GHashTable* cache = g_hash_table_new();  
    char* contents = NULL;  
    // load file data from disk or cache (omitted)  
    return contents;  
}
```



Foundations of Stateful Functions



Foundations of Stateful Functions



State Threads:

```
load :: String -> State (HashMap String String)
          String
load :: String -> ST (MHashMap#(String, String))
```

Foundations of Stateful Functions



State Threads:

```
load :: String -> State (HashMap String String)
          String
load :: String -> ST (MHashMap s String String)
          String
```

$$f_{st} = (a, s) \rightarrow (b, s)$$

Composition:

$$\begin{aligned} f &= a \rightarrow b \\ g &= b \rightarrow c \\ g \circ f &= a \rightarrow c \end{aligned}$$

$$f_{st} = (a, s_f) \rightarrow (b, s_f)$$

$$g_{st} = (a, s_g) \rightarrow (c, s_g)$$

$$g_{st} \circ f_{st} = ?$$

Foundations of Stateful Functions



State Threads:

```
load :: String -> State (HashMap String String)
          String
load :: String -> ST (MHashMap#(String, String))
```

$$f_{st} = (a, s) \rightarrow (b, s)$$

Composition:

$$f = a \rightarrow b$$

$$g = b \rightarrow c$$

$$g \circ f = a \rightarrow c$$

$$f_{st} = (a, s_f) \rightarrow (b, s_f)$$

$$g_{st} = (a, s_g) \rightarrow (c, s_g)$$

$$g_{st} \circ f_{st} = ?$$

Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. 2019. STCLang: state thread composition as a foundation for monadic dataflow parallelism. In Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019). ACM.

Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. 2019. Category-Theoretic Foundations of “STCLang: state thread composition as a foundation for monadic dataflow parallelism”. arXiv preprint arXiv:1906.12098

Foundations of Stateful Functions



State Threads:

```
load :: String -> State (HashMap String String)
          String
load :: String -> ST (MHashMap#(String, String))
          String
```

$$f_{st} = (a, s) \rightarrow (b, s)$$

SMap:

- Formal analysis on determinism and deadlocks.
- Not only **maps** but also **folds**!
- Not only pipeline parallelism but also task-level and data parallelism!

Composition:

$$f = a \rightarrow b$$

$$g = b \rightarrow c$$

$$g \circ f = a \rightarrow c$$

$$f_{st} = (a, s_f) \rightarrow (b, s_f)$$

$$g_{st} = (a, s_g) \rightarrow (c, s_g)$$

$$g_{st} \circ f_{st} = ?$$

Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. 2019. STCLang: state thread composition as a foundation for monadic dataflow parallelism. In Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019). ACM.

Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. 2019. Category-Theoretic Foundations of “STCLang: state thread composition as a foundation for monadic dataflow parallelism”. arXiv preprint arXiv:1906.12098

Foundations of Stateful Functions



State Threads:

```
load :: String -> State (HashMap String String)
          String
load :: String -> ST (MHashMap#(String, String))
          String
```

$$f_{st} = (a, s) \rightarrow (b, s)$$

SMap:

- Formal analysis on determinism and deadlocks.
- Not only **maps** but also **folds**!
- Not only pipeline parallelism but also task-level and data parallelism!

Composition:

$$f = a \rightarrow b$$

$$g = b \rightarrow c$$

$$g \circ f = a \rightarrow c$$

$$f_{st} = (a, s_f) \rightarrow (b, s_f)$$

$$g_{st} = (a, s_g) \rightarrow (c, s_g)$$

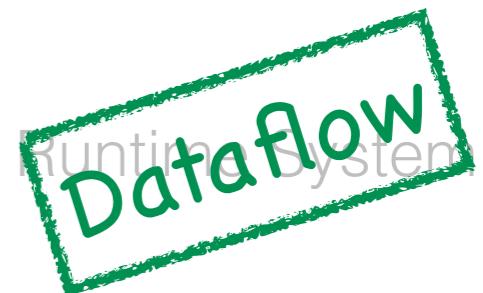
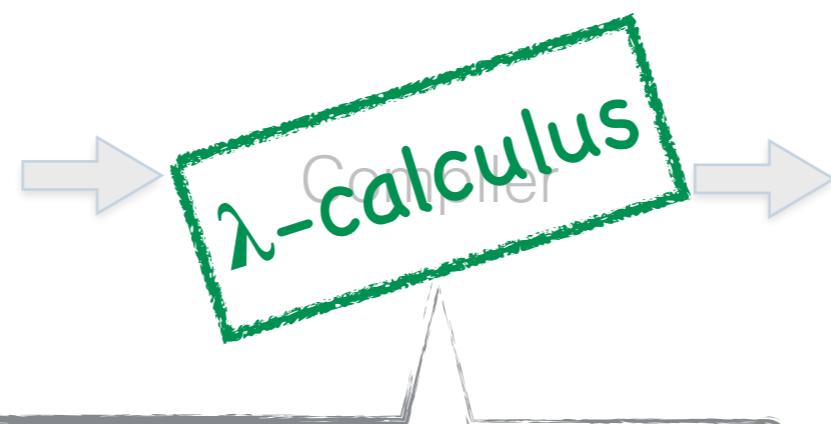
$$g_{st} \circ f_{st} = ?$$

```
fn map_reduce() {
  let stateG = init_G();
  for data in list {
    let x = f(data);
    stateG.g(x)
  };
  stateG
}
```

Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. 2019. STCLang: state thread composition as a foundation for monadic dataflow parallelism. In Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019). ACM.

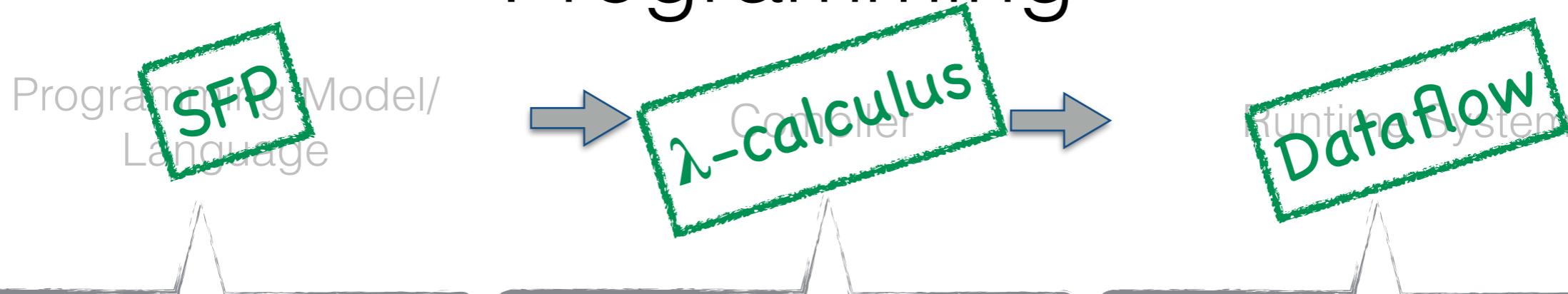
Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. 2019. Category-Theoretic Foundations of “STCLang: state thread composition as a foundation for monadic dataflow parallelism”. arXiv preprint arXiv:1906.12098

Programming Model/
Language



- Intermediate Representation (IR) needs a solid well-studied theoretical foundation for
- provable semantic-preserving transformations
- even for programs with side-effects.

Ingredients for Implicit Parallel Programming

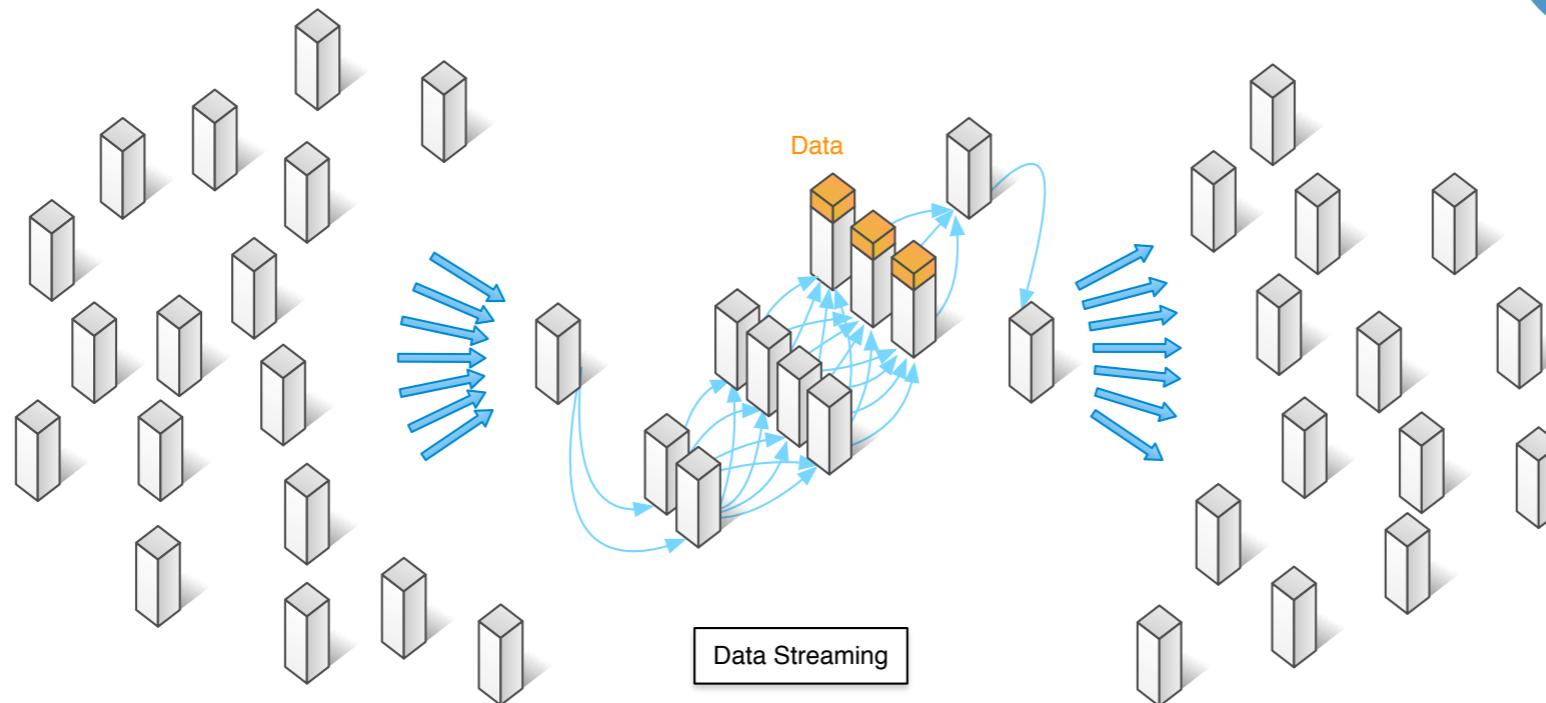


- No concurrency abstractions.
- Practicality:
 - Integration into existing programming models (e.g., OOP)/languages.
 - Gradual switch/reuse of existing code base.
- Intermediate Representation (IR) needs a solid well-studied theoretical foundation for
 - provable semantic-preserving transformations
 - even for programs with side-effects.
- Flexible/dynamic runtime representation.
- Efficient parallel execution of independent computations.
- Deterministic execution.
- Concurrent I/O.

Changing the Game

- Functional Reactive Programming (FRP) based on stateful functions.
- FRP instead of streams for big data systems.

<https://github.com/ohua-dev/stc-lang>



- Current Main Focus: Rust integration ➡ Safety
<https://github.com/ohua-dev/ohua-rust-runtime>
- STM alternative.
- User-defined functions and NoSQL (in Noria).



Thanks for your attention!



<https://ohua-dev.github.io/>

Backup Slides

Programming Model/
Language

SFP

λ -calculus

Dataflow

Sebastian Ertel and Pascal Felber. 2014. A framework for the dynamic evolution of highly-available dataflow programs. In Proceedings of the 15th International Middleware Conference (Middleware '14). ACM.

Sebastian Ertel, Christof Fetzer, and Pascal Felber. 2015. Ohua: Implicit Dataflow Programming for Concurrent Systems. In Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ '15). ACM.

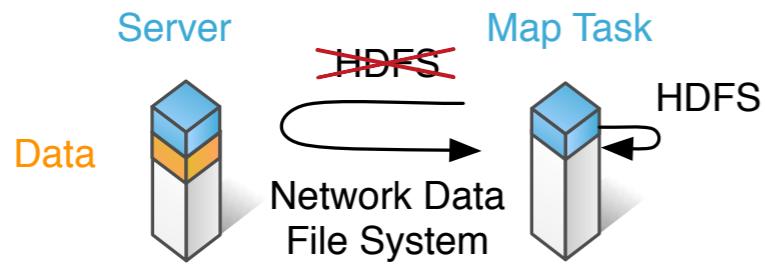
Sebastian Ertel, Justus Adam, and Jeronimo Castrillon. 2018. Supporting Fine-grained Dataflow Parallelism in Big Data Systems. In Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'18). ACM.

Sebastian Ertel, Andrés Goens, Justus Adam, and Jeronimo Castrillon. 2018. Compiling for concise code and efficient I/O. In Proceedings of the 27th International Conference on Compiler Construction (CC 2018). ACM.

Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. 2019. STCLang: state thread composition as a foundation for monadic dataflow parallelism. In Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019). ACM.

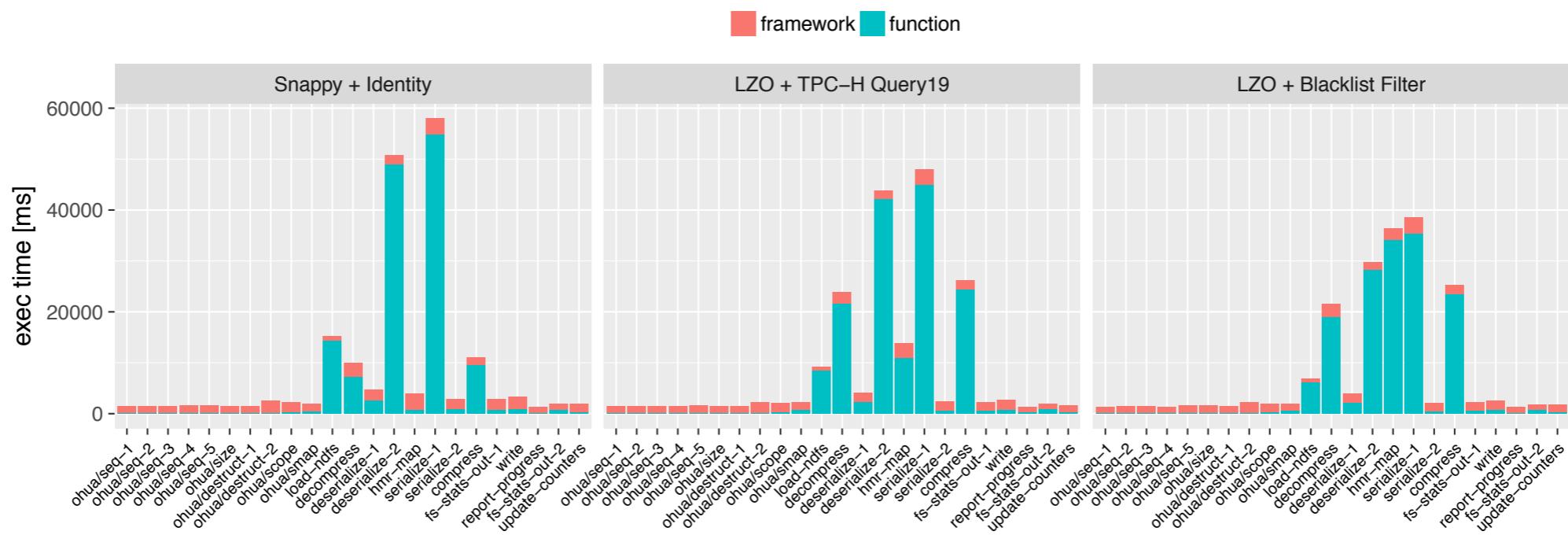
Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. 2019. Category-Theoretic Foundations of “STCLang: state thread composition as a foundation for monadic dataflow parallelism”. arXiv preprint arXiv:1906.12098

Execution Breakdown



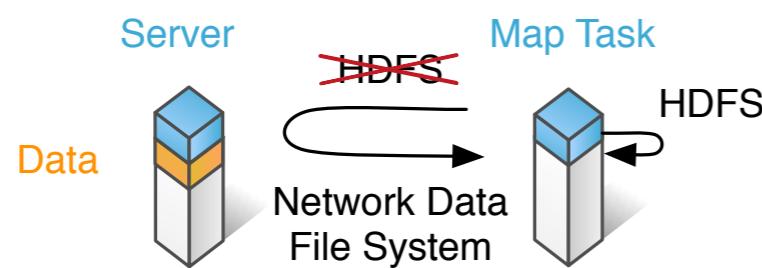
- SequenceFile
 - JSON
 - Snappy, LZO
 - TPC-H Parts table

Almost the same setup as for
a reduce task!



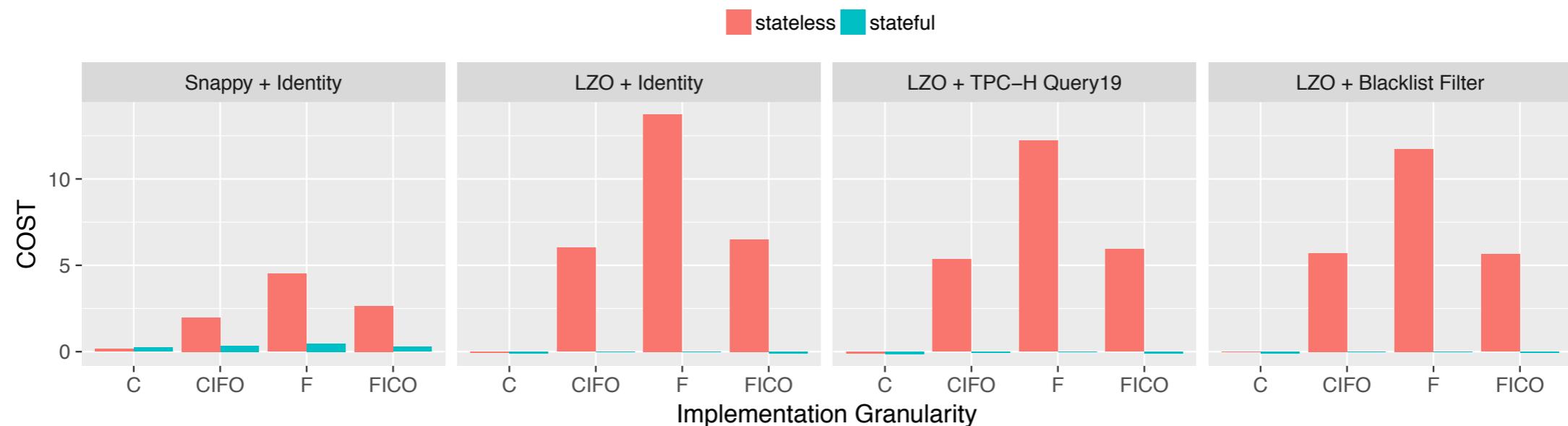
Pipeline parallelism depends on balanced individual stages!

COST Analysis



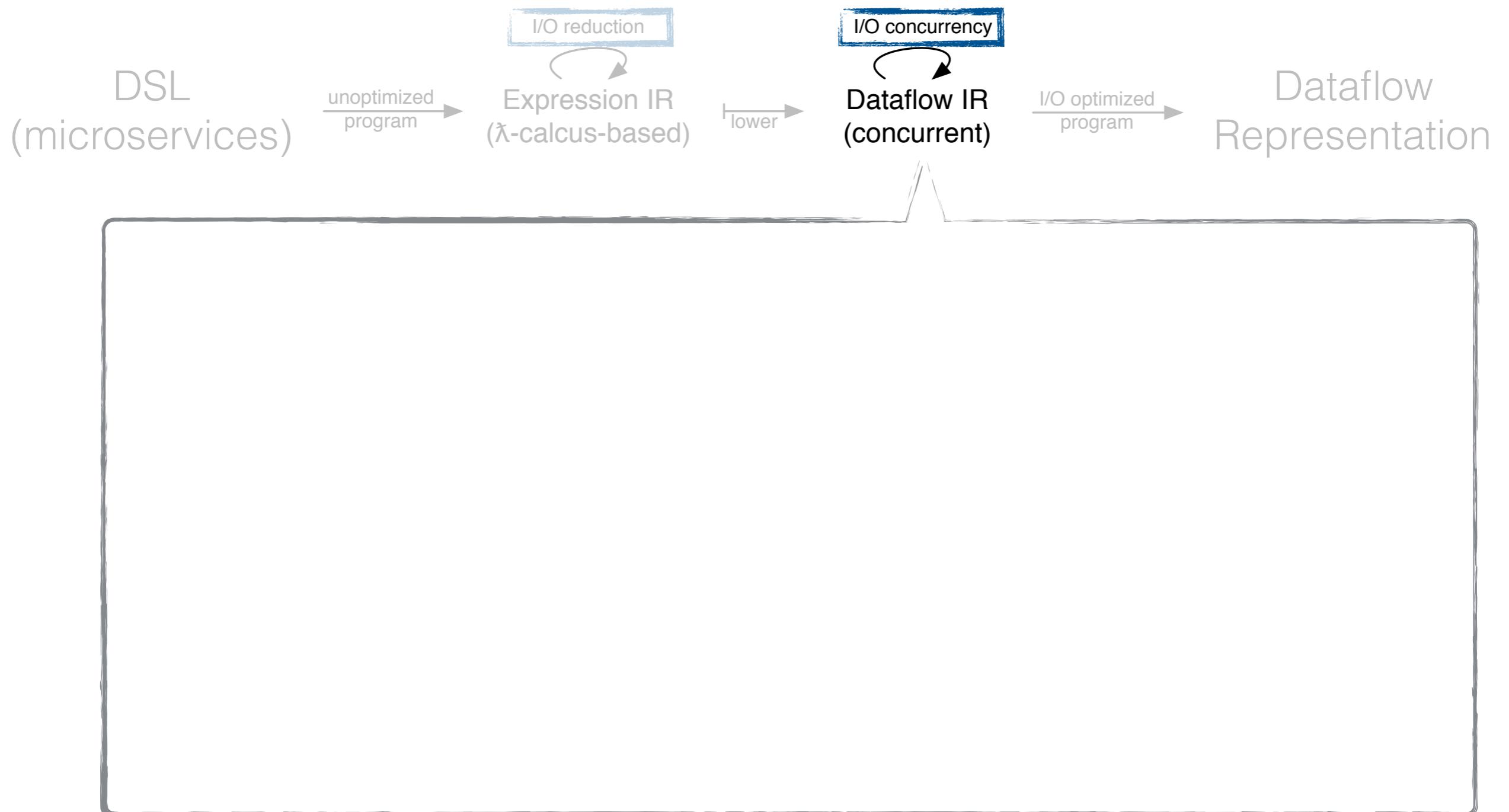
- SequenceFile
- JSON
- Snappy, LZO
- TPC-H Parts table

Almost the same setup as for
a reduce task!

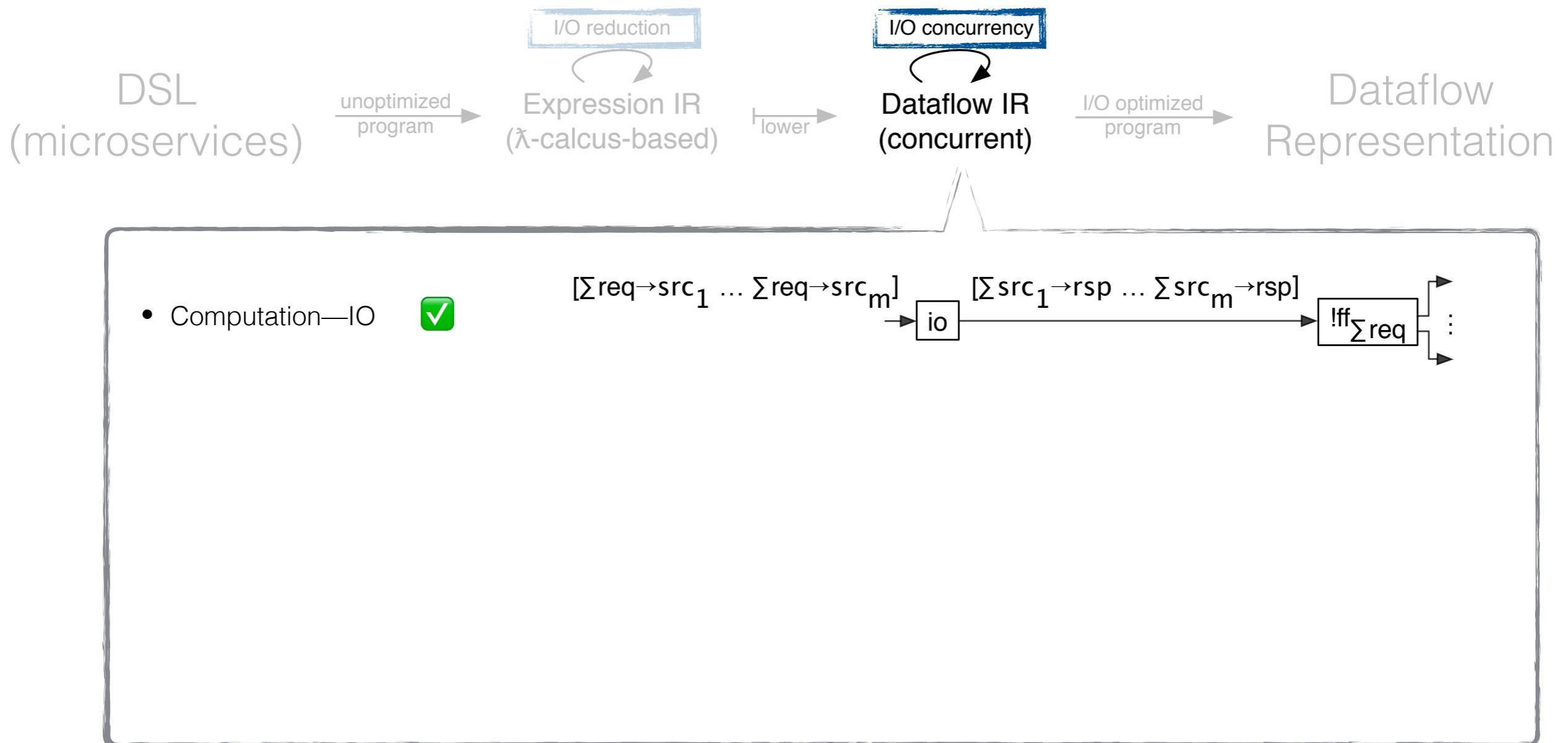


Processing is stateful!

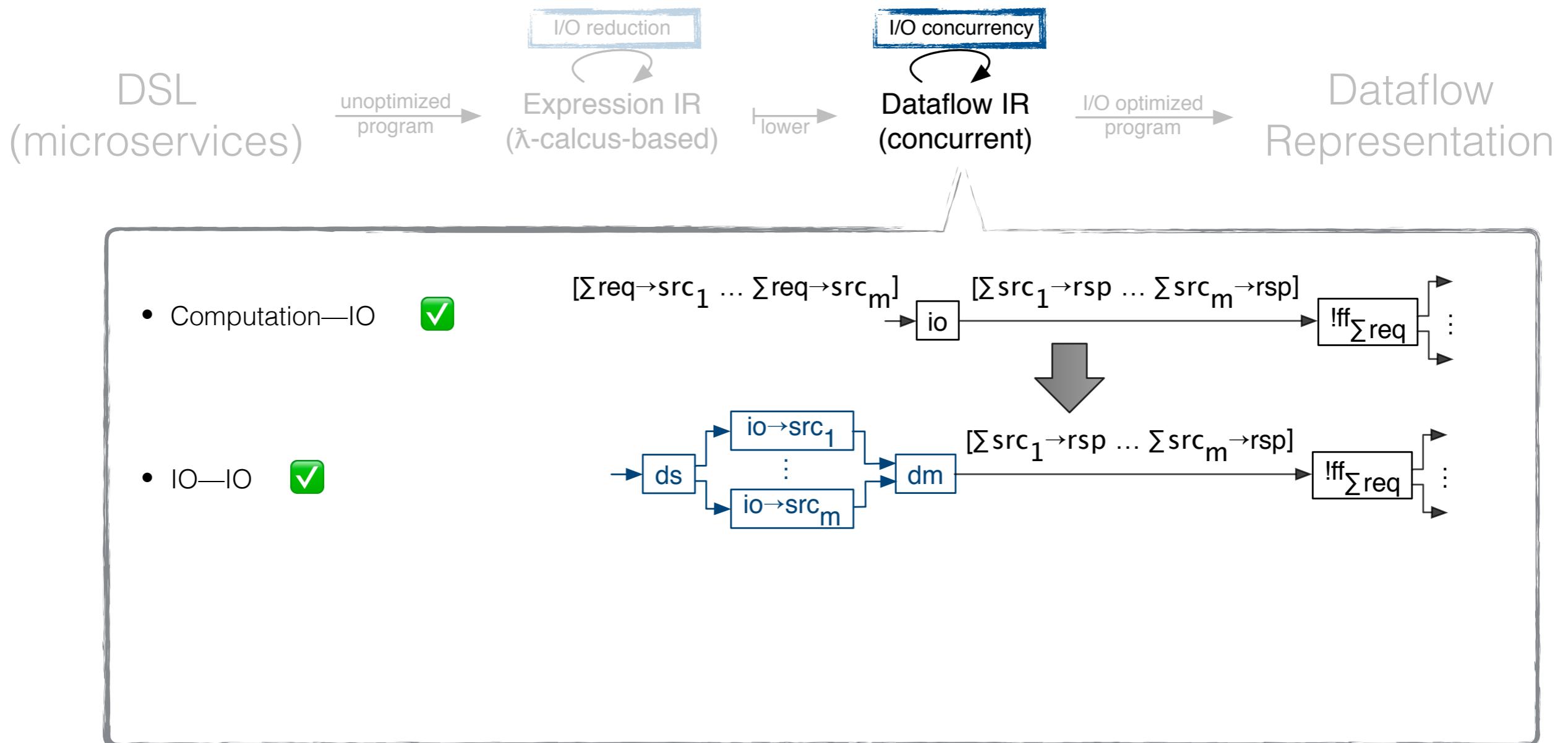
Dataflow Concurrency



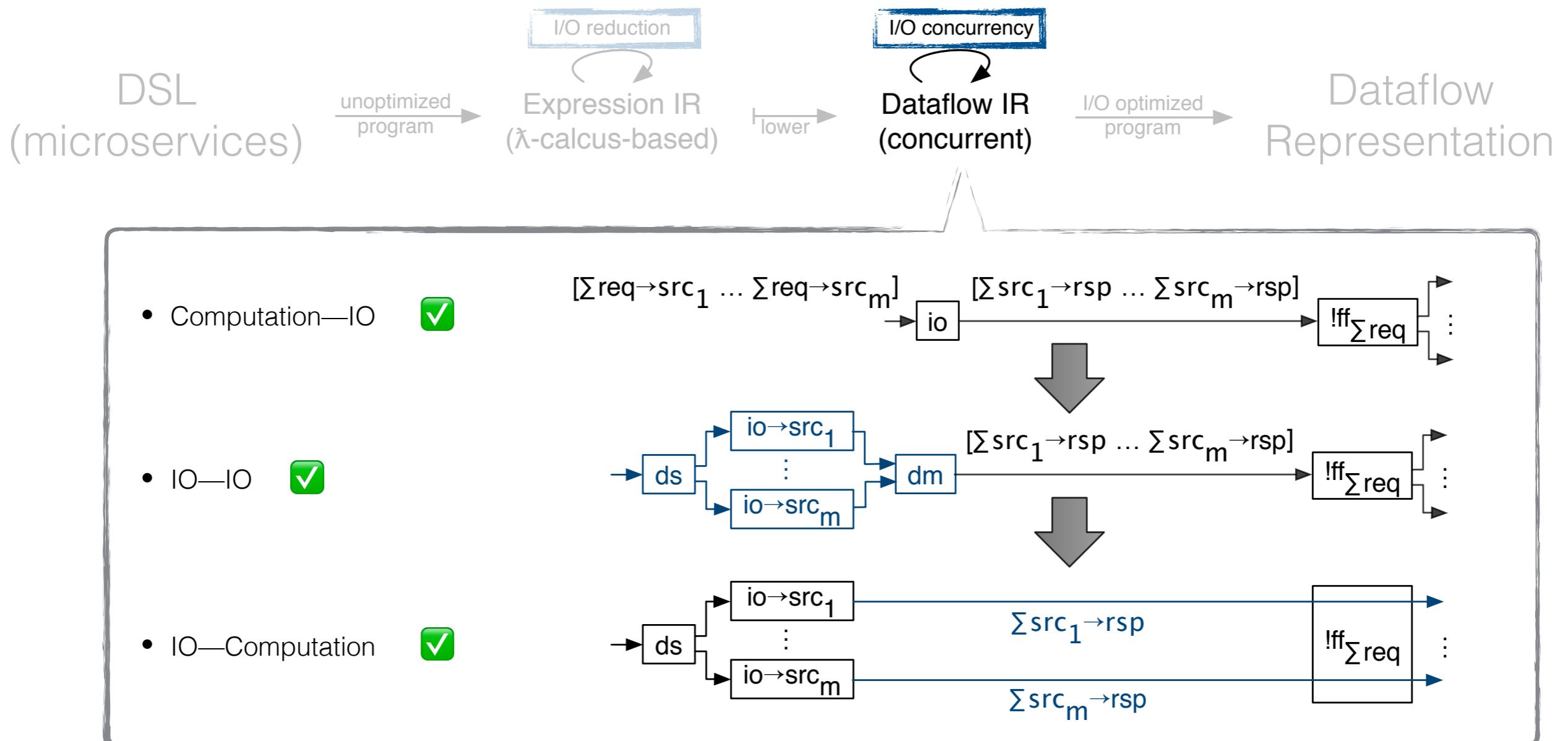
Dataflow Concurrency



Dataflow Concurrency

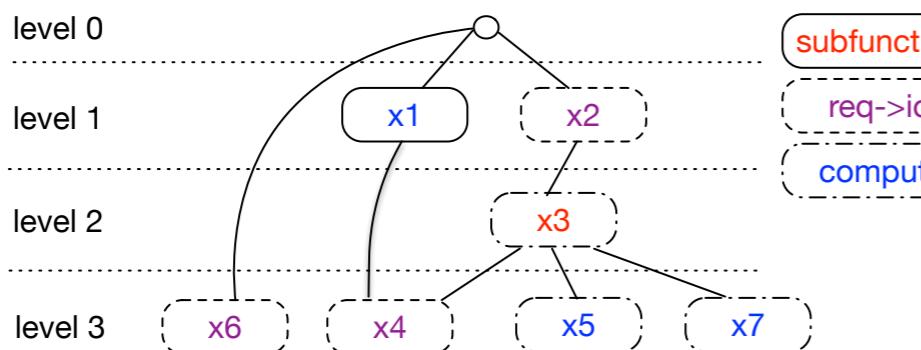


Dataflow Concurrency



Evaluation

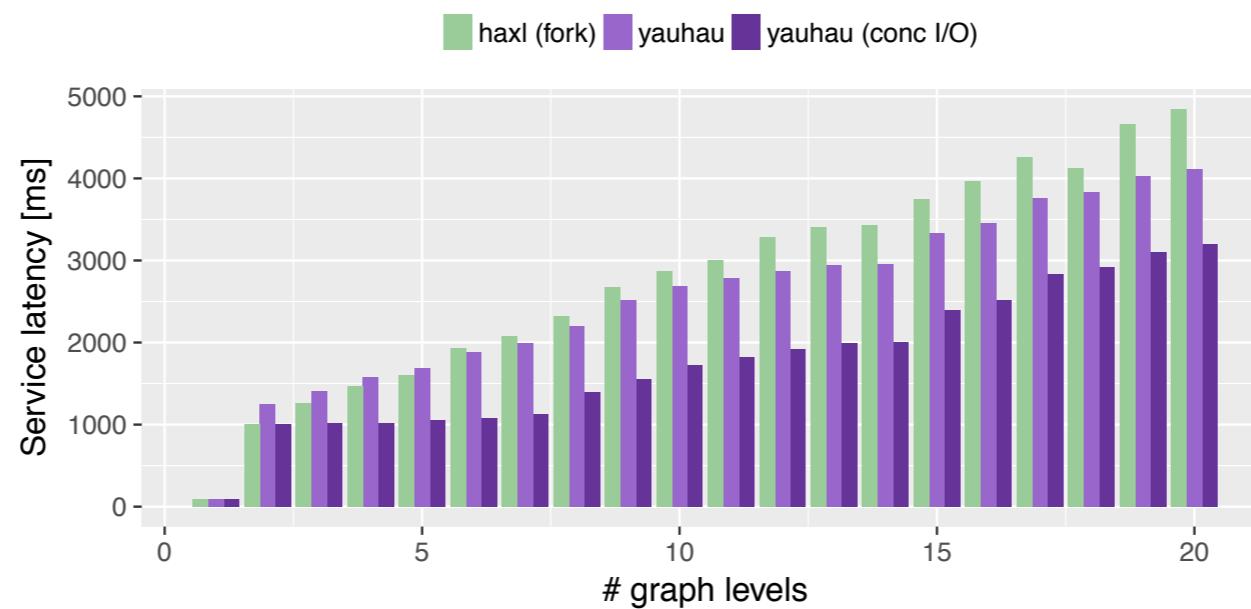
Level-Graphs:



Programs:

- Haxl
- Muse
- Seq
- Ÿauhau

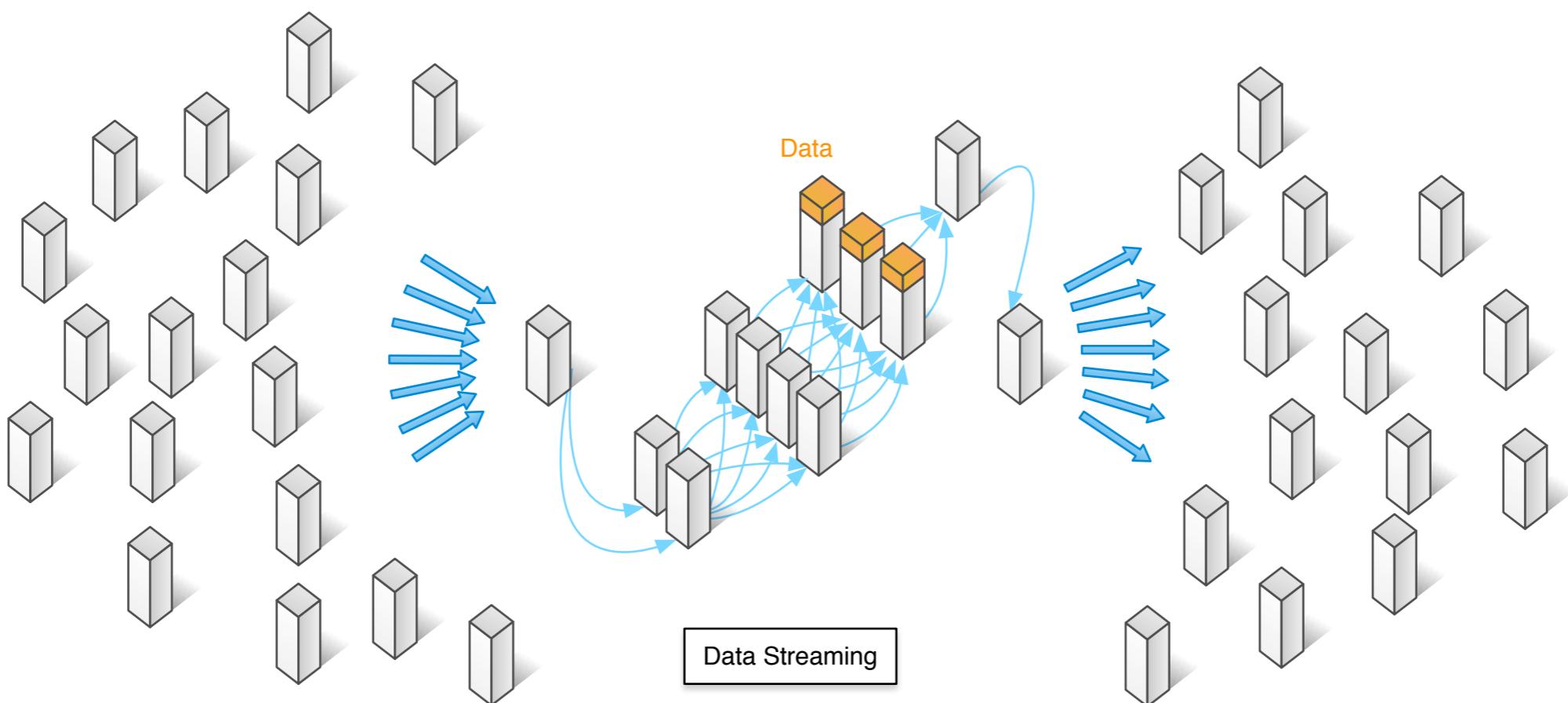
serialize



Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is no fork: an abstraction for efficient, concurrent, and concise data access. ICFP '14.

Alexey Kachayev. 2015. Reinventing Haxl: Efficient, Concurrent and Concise Data Access. Presentation at EuroClojure 2015.

Data Streaming



Programming Model/
Language



Compiler



Runtime System

Dataflow:

- Parallelism
- DSE

Hierarchical Scheduling

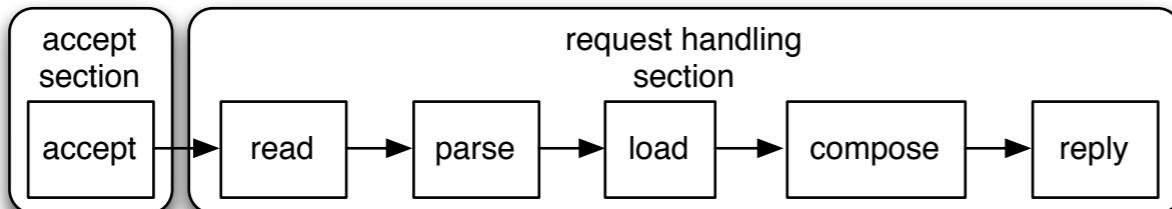
- TODO

Flexible Operator Mappings (Sections)

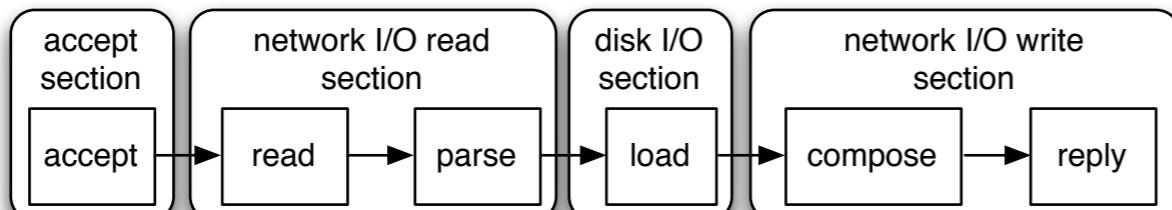
- TODO

Evaluating Different Server Design

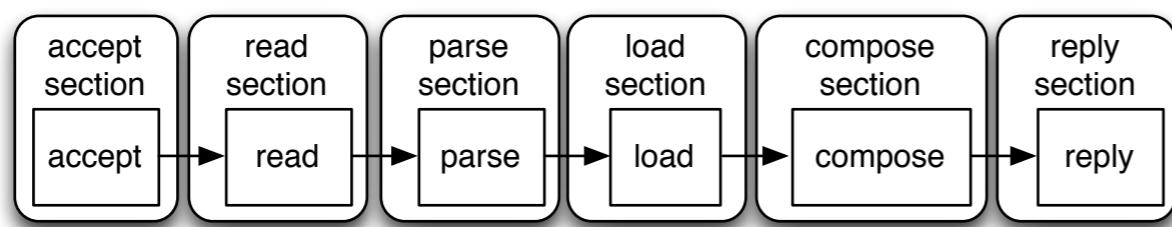
- Classic



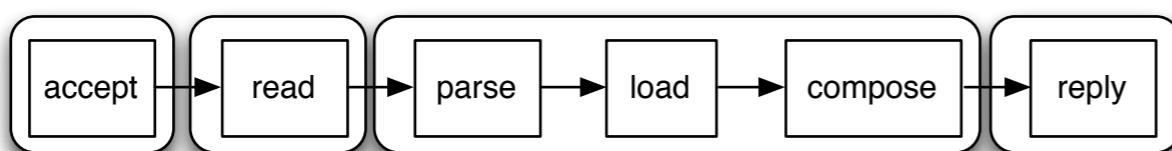
- NIO-style



- Naive

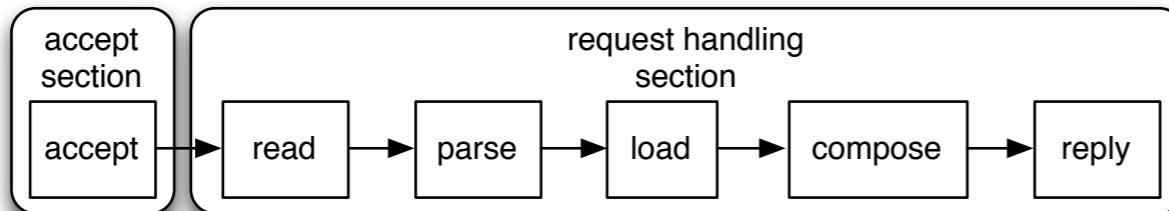


- Isolated network I/O

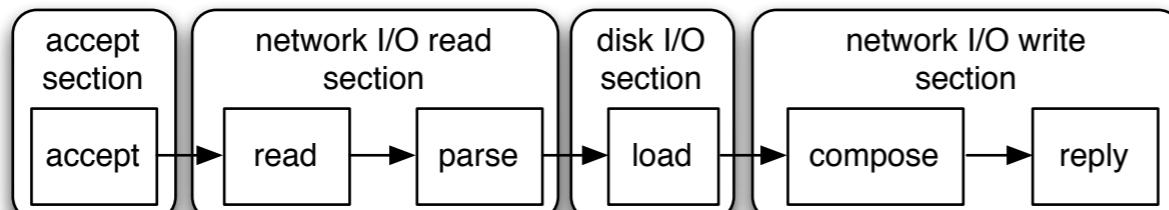


Evaluating Different Server Design

- Classic

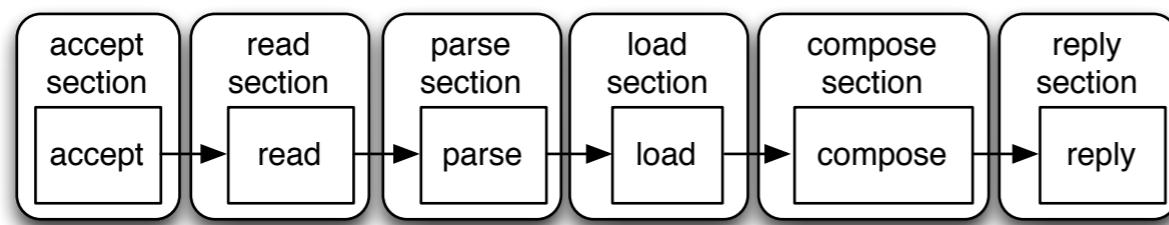


- NIO-style

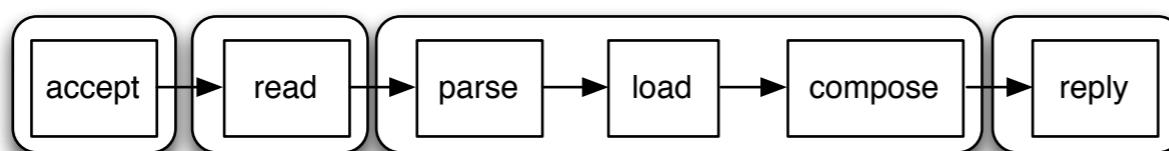


Jetty has two separate implementations of these!

- Naive

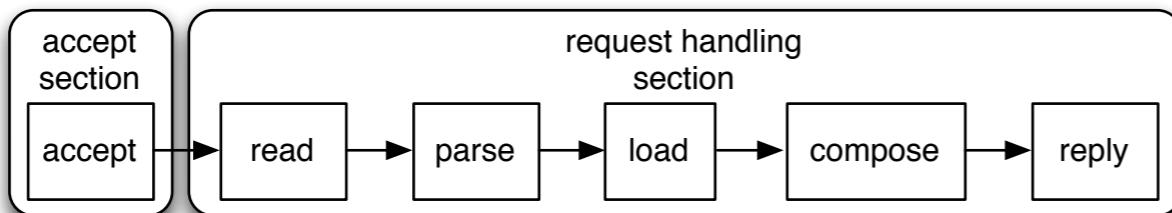


- Isolated network I/O

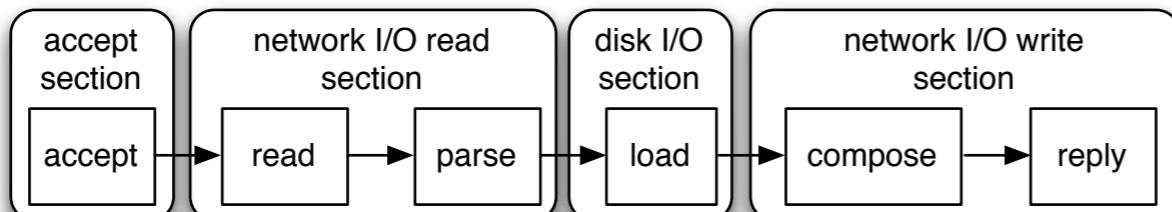


Evaluating Different Server Design

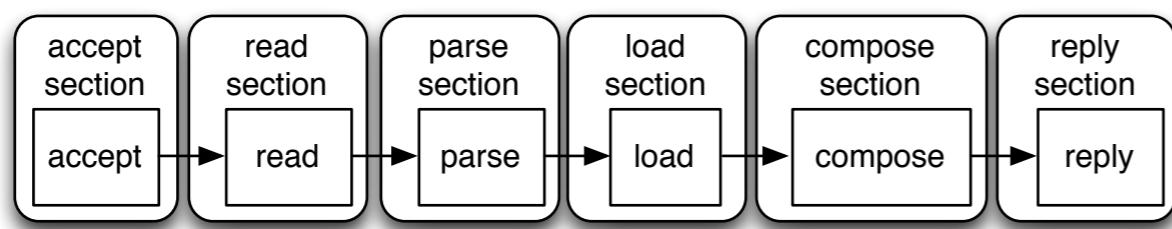
- Classic



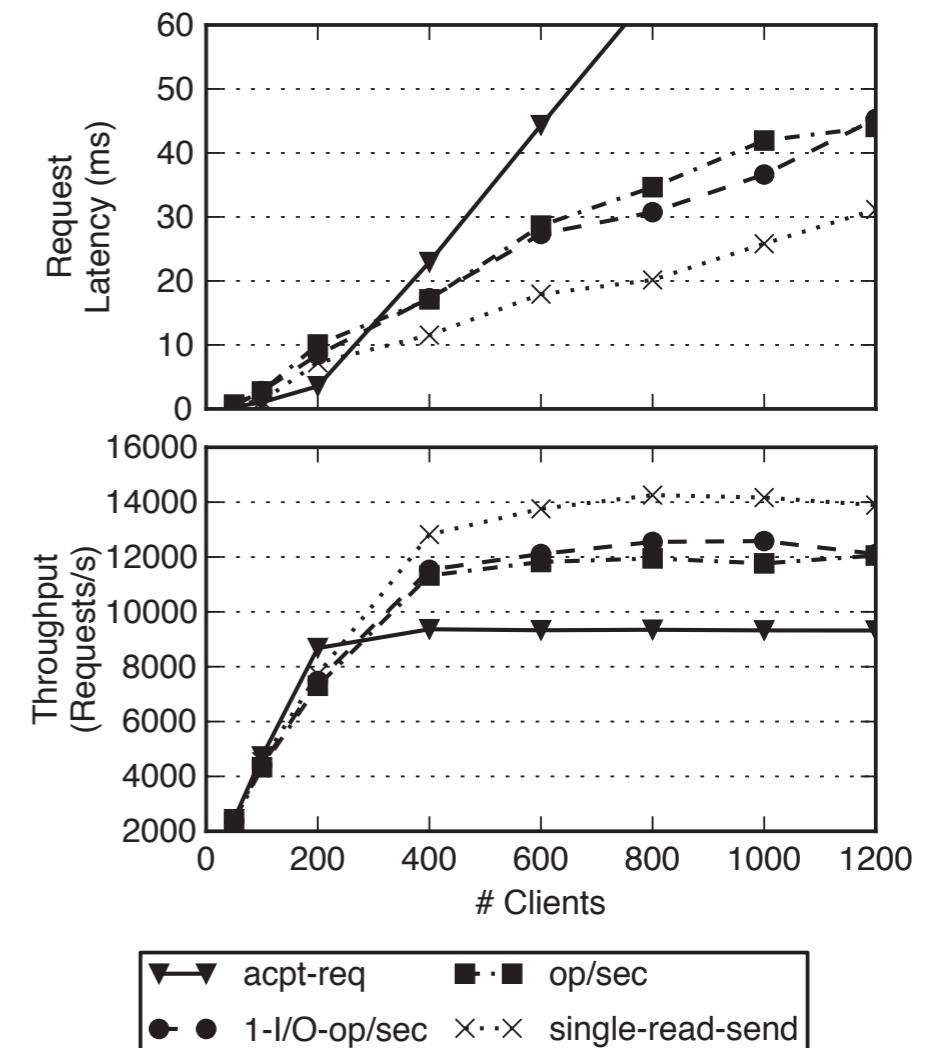
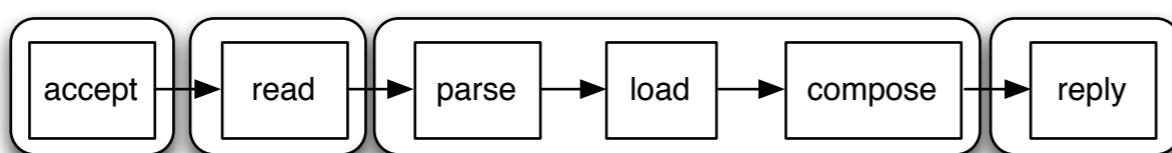
- NIO-style



- Naive



- Isolated network I/O



Programming Model/
Language



Compiler

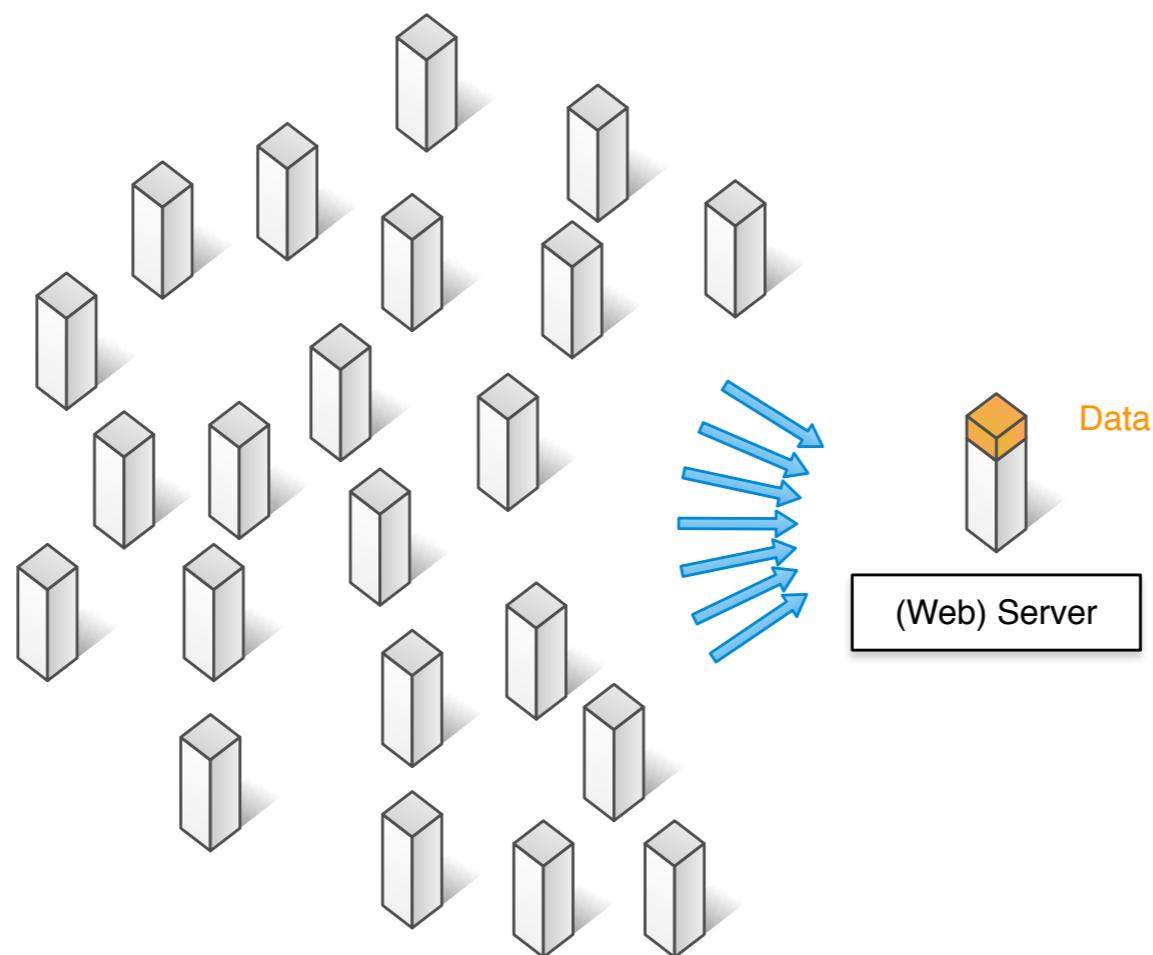


Runtime System

Dataflow:

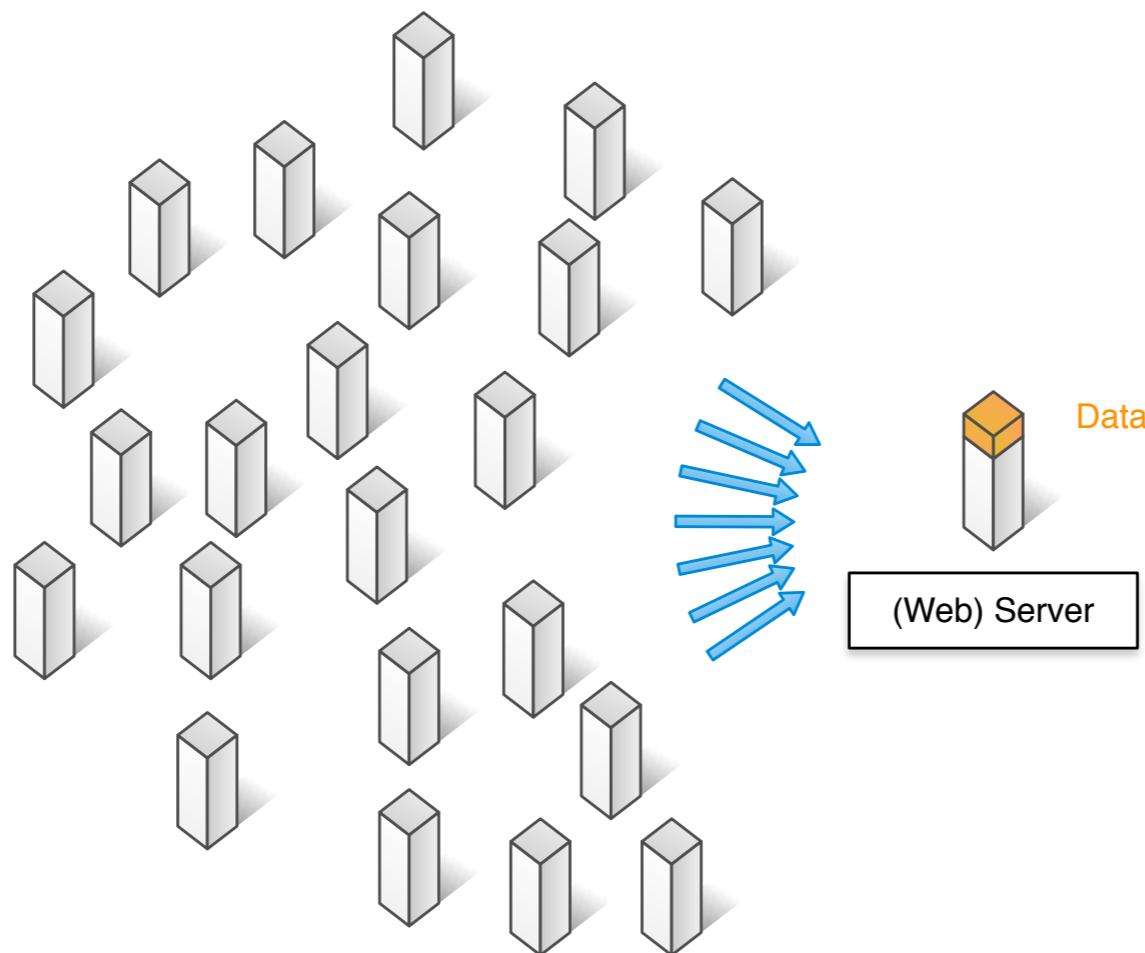
- Parallelism
- DSE
- Mapping/Scheduling

Server Architecture



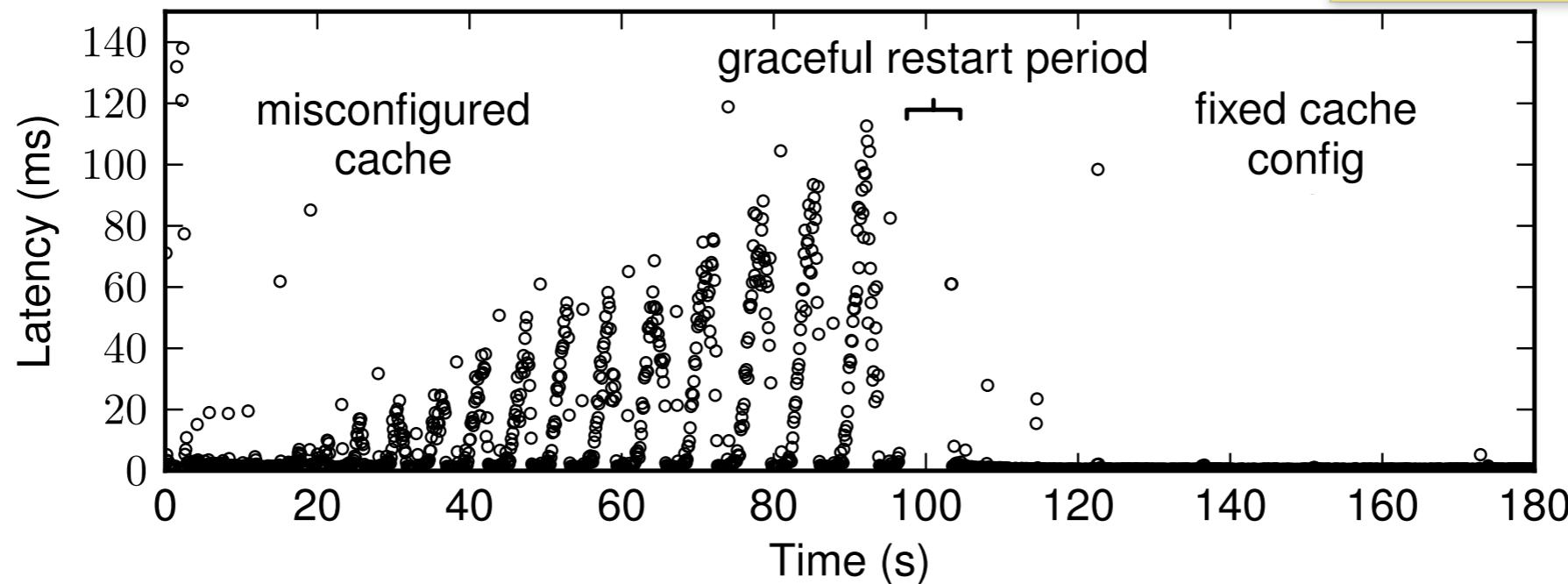
Server Architecture

Requirement: 24/7 availability ↗ Problem: **Updates**



Jetty's Graceful Restart

Make heading more precise:
Jetty's restart incurs high downtimes



Downtimes: 8–10 s \Rightarrow Request latency: 1–2

Lost request count!

Hot Swapping

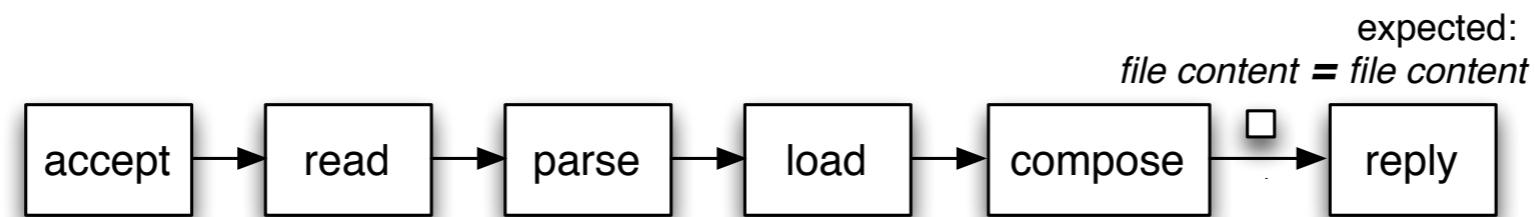
Idea:
Update program during execution



Problem:
Preserve program correctness

Main challenges:

- Consistency ➔ State quiescence
- Mutual references ➔ Non-blocking update coordination



Feng, N. et.al., "Dynamic evolution of network management software by software hot-swapping," IFIP/IEEE IM 2001.

Iulian Neamtiu and Michael Hicks. 2009. Safe and timely updates to multi-threaded programs. In PLDI '09.

Hayden, C.M.; Hicks, M.; et.al. A study of dynamic software update quiescence for multithreaded programs, HotSWUp'12

Hot Swapping

Idea:
Update program during execution

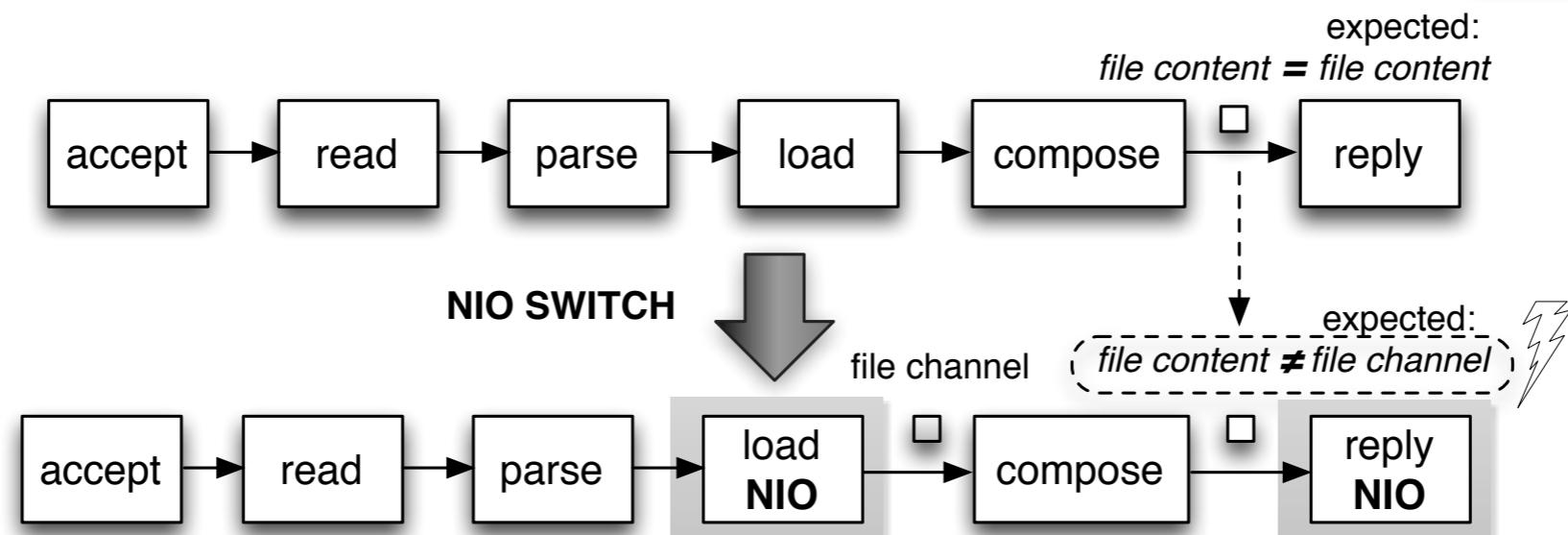


Problem:
Preserve program correctness

Main challenges:

- Consistency ➔ State quiescence
- Mutual references ➔ Non-blocking update coordination

Unclear that light marker contains data with “file content”



Feng, N. et.al., "Dynamic evolution of network management software by software hot-swapping," IFIP/IEEE IM 2001.

Iulian Neamtiu and Michael Hicks. 2009. Safe and timely updates to multi-threaded programs. In PLDI '09.

Hayden, C.M.; Hicks, M.; et.al. A study of dynamic software update quiescence for multithreaded programs, HotSWUp'12

Hot Swapping

Idea:

Update program during execution



Problem:

Preserve program correctness

Main challenges:

- Consistency ➔ State quiescence
- Mutual references ➔ Non-blocking update coordination

Hot Swapping

Idea:

Update program during execution



Problem:

Preserve program correctness

Main challenges:

• Consistency ➔ State quiescence

• Mutual references ➔ Non-blocking update coordination

Hot Swapping

Idea:

Update program during execution



Problem:

Preserve program correctness

Main challenges:

- Consistency ➡ State quiescence := who? and when?
- Mutual references ➡ Non-blocking update coordination

Hot Swapping

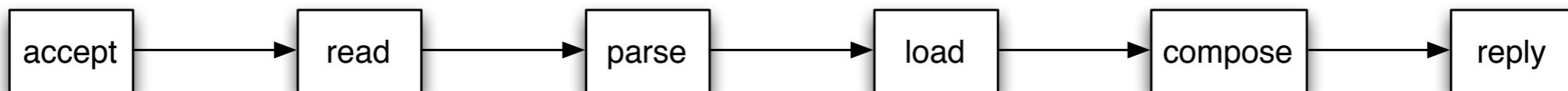
Idea:
Update program during execution



Problem:
Preserve program correctness

Main challenges:

- Consistency State quiescence := who? and when?



- Mutual references Non-blocking update coordination

Hot Swapping

Idea:
Update program during execution

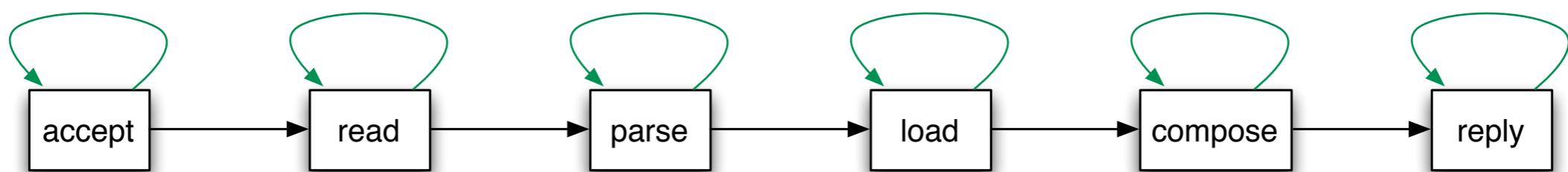


Problem:
Preserve program correctness

Main challenges:

- Consistency ➔ State quiescence := **who?** and **when?**

Operator-local state



- Mutual references ➔ Non-blocking update coordination

Hot Swapping

Idea:
Update program during execution



Problem:
Preserve program correctness

Main challenges:

- Consistency



State quiescence := **who?** and **when?**



- Mutual references



Non-blocking update coordination

Hot Swapping

Idea:
Update program during execution

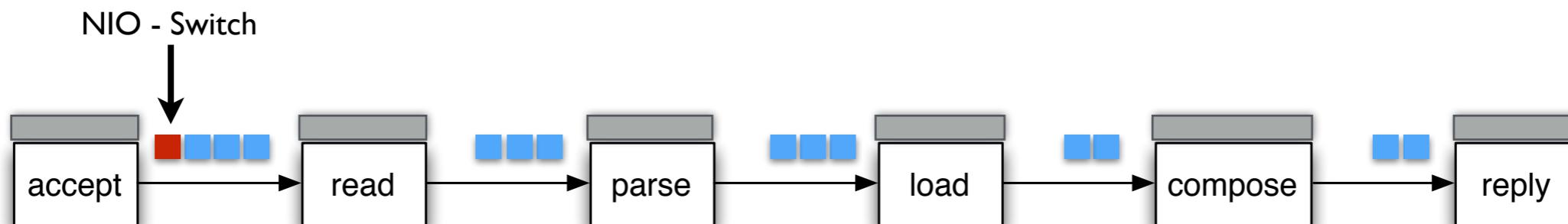


Problem:
Preserve program correctness

Main challenges:

- Consistency

⇒ State quiescence := **who?** and **when?**



- Mutual references

⇒ Non-blocking update coordination

Hot Swapping

Idea:
Update program during execution

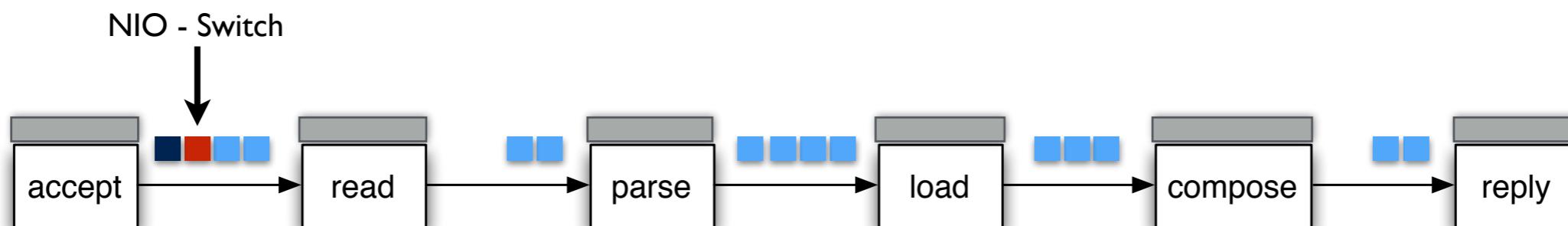


Problem:
Preserve program correctness

Main challenges:

- Consistency

⇒ State quiescence := **who?** and **when?**



- Mutual references

⇒ Non-blocking update coordination

Hot Swapping

Idea:
Update program during execution

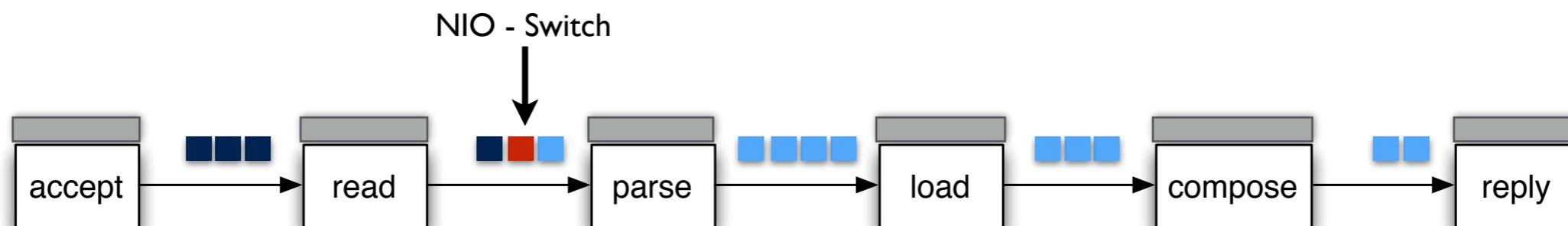


Problem:
Preserve program correctness

Main challenges:

- Consistency

⇒ State quiescence := **who?** and **when?**



- Mutual references

⇒ Non-blocking update coordination

Hot Swapping

Idea:
Update program during execution



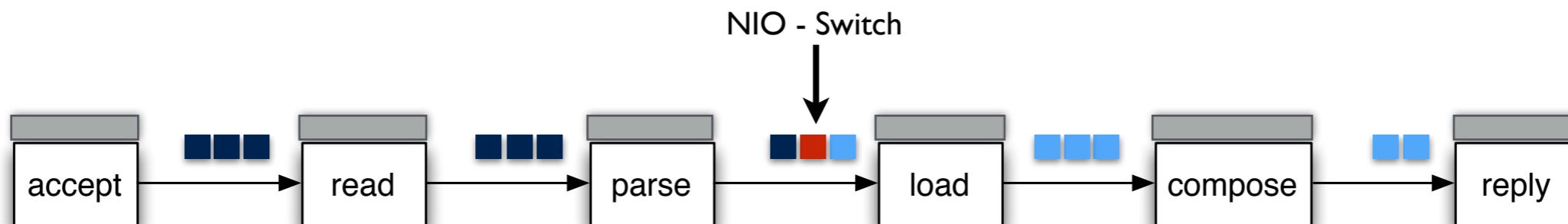
Problem:
Preserve program correctness

Main challenges:

- Consistency



State quiescence := **who?** and **when?**



- Mutual references



Non-blocking update coordination

Hot Swapping

Idea:
Update program during execution

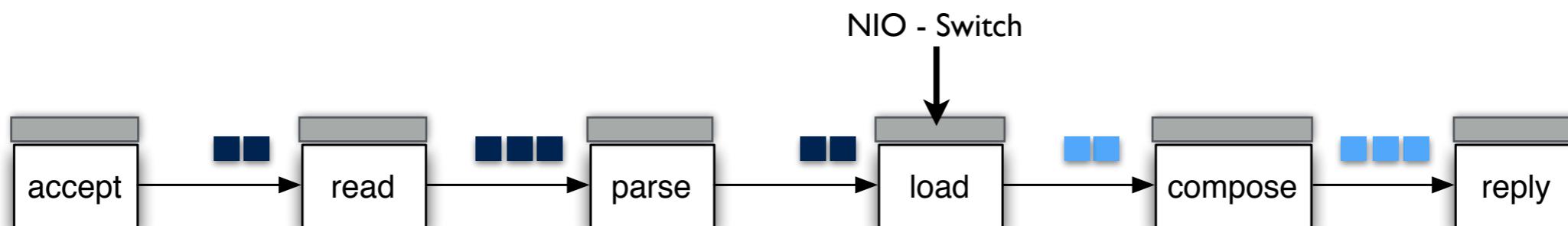


Problem:
Preserve program correctness

Main challenges:

- Consistency

⇒ State quiescence := **who?** and **when?**



- Mutual references

⇒ Non-blocking update coordination

Hot Swapping

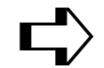
Idea:
Update program during execution



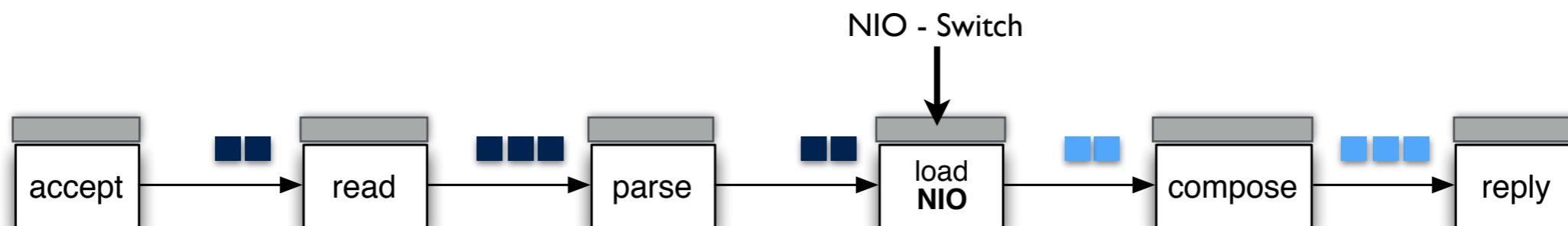
Problem:
Preserve program correctness

Main challenges:

- Consistency



State quiescence := **who?** and **when?**



- Mutual references



Non-blocking update coordination

Hot Swapping

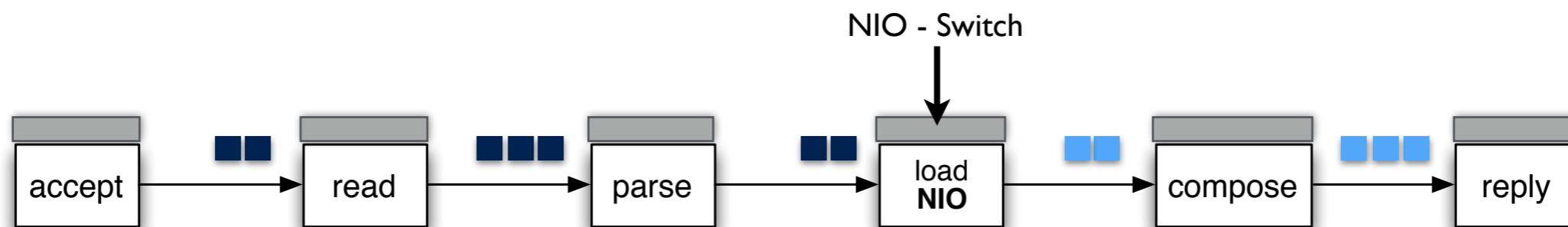
Idea:
Update program during execution



Problem:
Preserve program correctness

Main challenges:

- Consistency ➔ State quiescence := who? and when?
- Mutual references ➔ Non-blocking update coordination



Hot Swapping

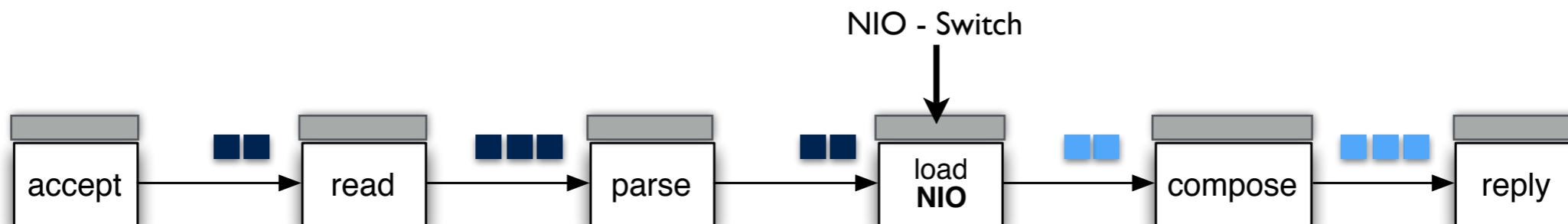
Idea:
Update program during execution



Problem:
Preserve program correctness

Main challenges:

- Consistency ➔ State quiescence := who? and when?
- Mutual references ➔ Non-blocking update coordination



Dataflow



Distributed
System

Hot Swapping

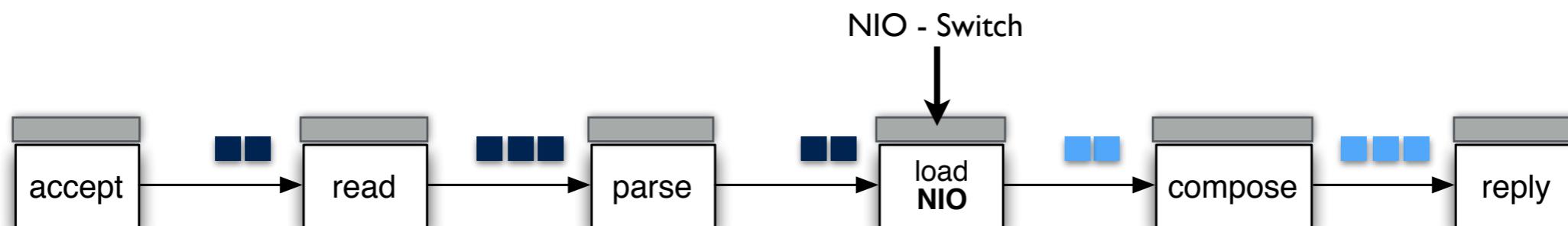
Idea:
Update program during execution



Problem:
Preserve program correctness

Main challenges:

- Consistency → State quiescence := who? and when?
- Mutual references → Non-blocking update coordination



Dataflow



Distributed System



Introduction of a solid notion of time!

K. Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. 3, 1 (February 1985), 63-75.

Hot Swapping

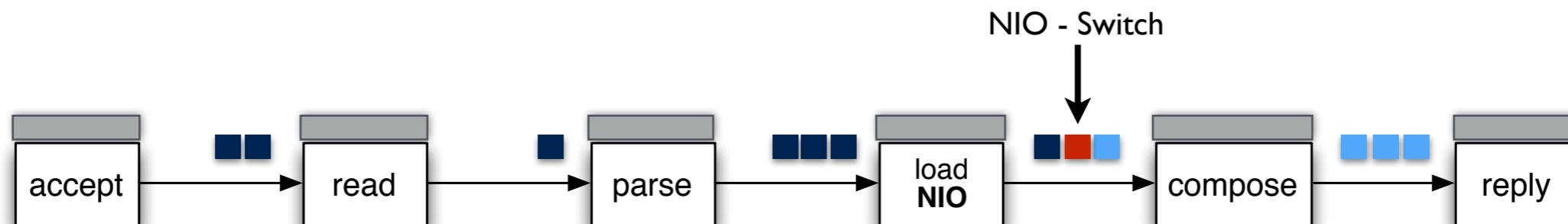
Idea:
Update program during execution



Problem:
Preserve program correctness

Main challenges:

- Consistency → State quiescence := who? and when?
- Mutual references → Non-blocking update coordination



Dataflow



Distributed System



Introduction of a solid notion of time!

K. Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. 3, 1 (February 1985), 63-75.

Hot Swapping

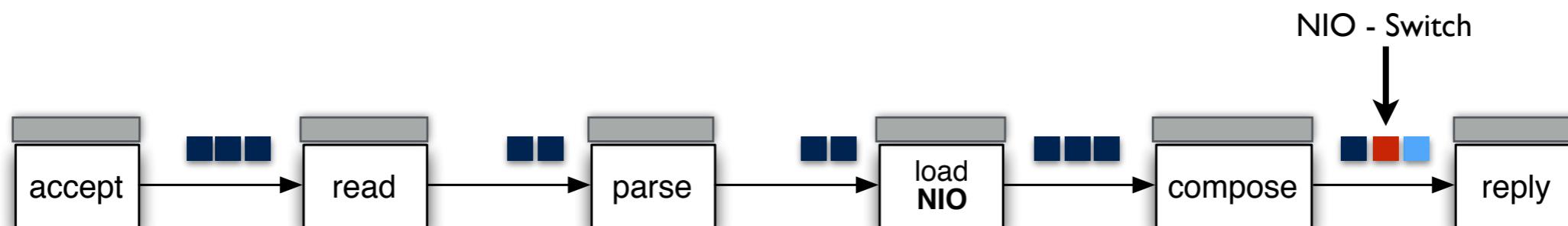
Idea:
Update program during execution



Problem:
Preserve program correctness

Main challenges:

- Consistency → State quiescence := who? and when?
- Mutual references → Non-blocking update coordination



Dataflow



Distributed System



Introduction of a solid notion of time!

K. Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. 3, 1 (February 1985), 63-75.

Program equivalence!
Correctness? Consistency?

Hot Swapping

Idea:
Update program during execution



Problem:
Preserve program correctness

Main challenges:

- Consistency → State quiescence := who? and when?
- Mutual references → Non-blocking update coordination



Dataflow



Distributed
System



Introduction of a solid
notion of time!

K. Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. 3, 1 (February 1985), 63-75.

From Live Updates to Dynamic Software Evolution



Procedure: Program replica + state transfer

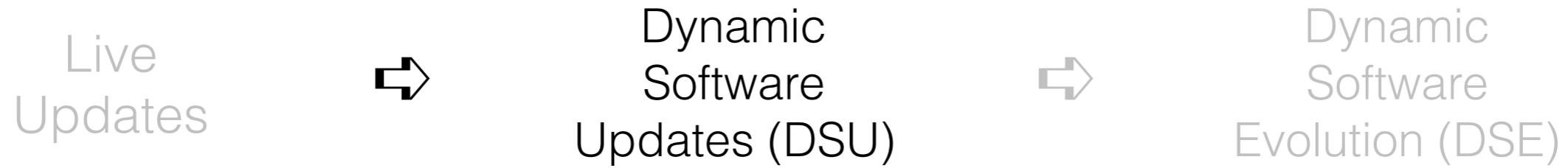
Impact: Typically small and local bug fixes or security patches

Occurrence: Infrequent

Christopher M. Hayden, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. State transfer for clear and efficient runtime updates. IEEE ICDEW '11.

Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. Kitsune: efficient, general-purpose dynamic software updating for C. OOPSLA '12.

From Live Updates to Dynamic Software Evolution



Procedure: In-place code + state update

Impact: Typically small and local bug fixes or security patches

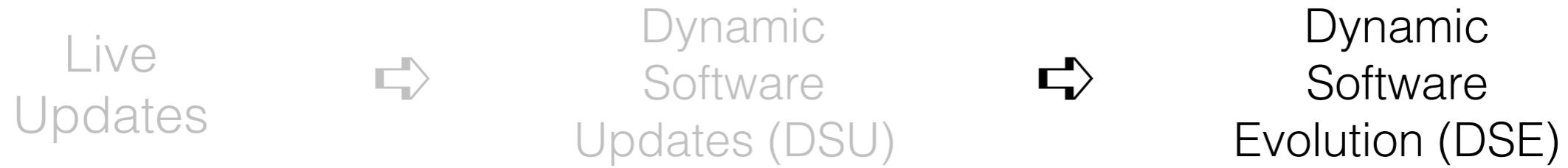
Occurrence: Infrequent

Michael Hicks and Scott Nettles. 2005. Dynamic software updating. ACM Trans. Program. Lang. Syst. (TOPLAS)

Iulian Neamtiu and Michael Hicks. 2009. Safe and timely updates to multi-threaded programs. In PLDI '09.

Cristiano Giuffrida and Andrew S. Tanenbaum. Cooperative update: a new model for dependable live update. In HotSWUp '09

From Live Updates to Dynamic Software Evolution



Procedure: In-place code + state update

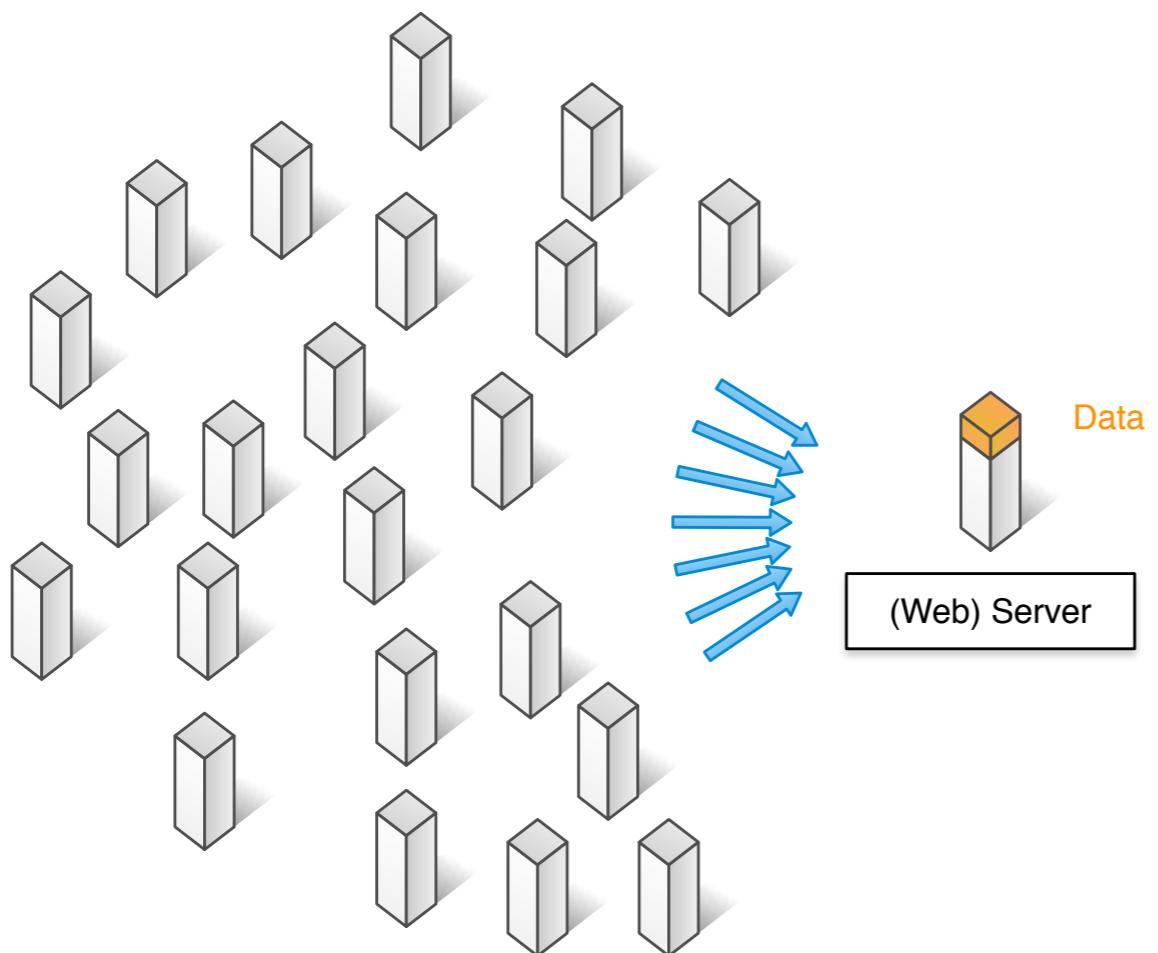
Impact: Local changes (operators) and program updates (graph).

Occurrence: Part of the development process

⇒ Dynamic evolution of any (middleware) data flow program.

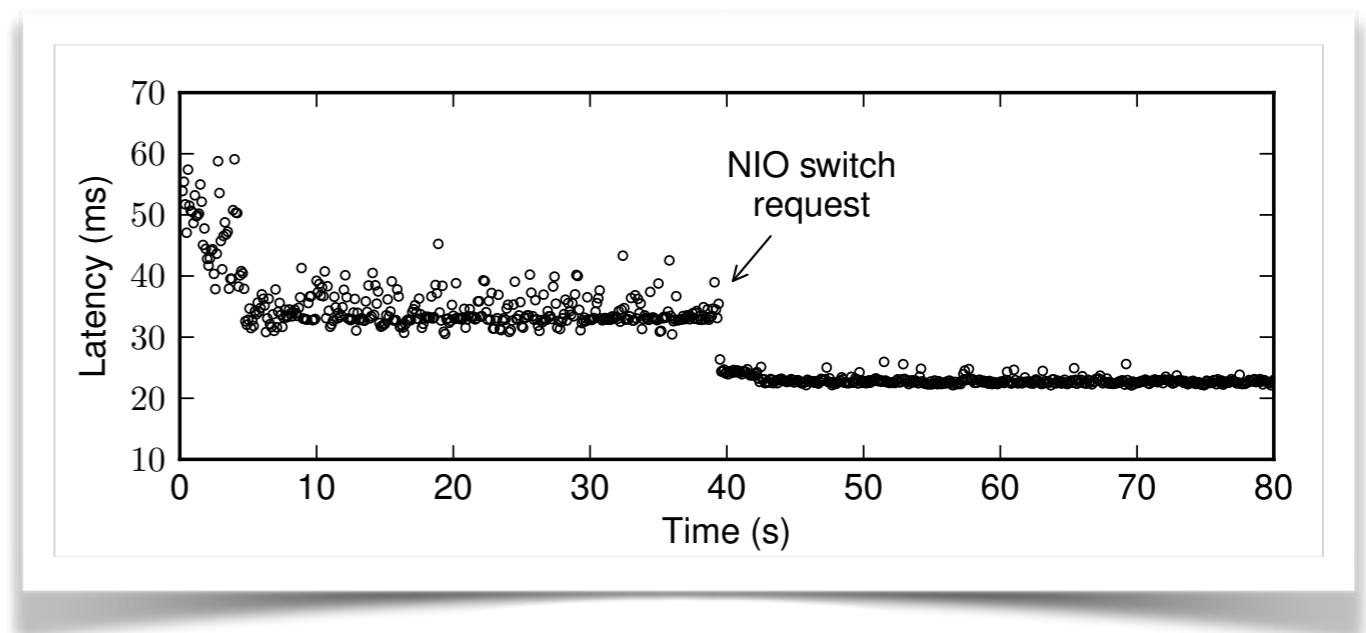
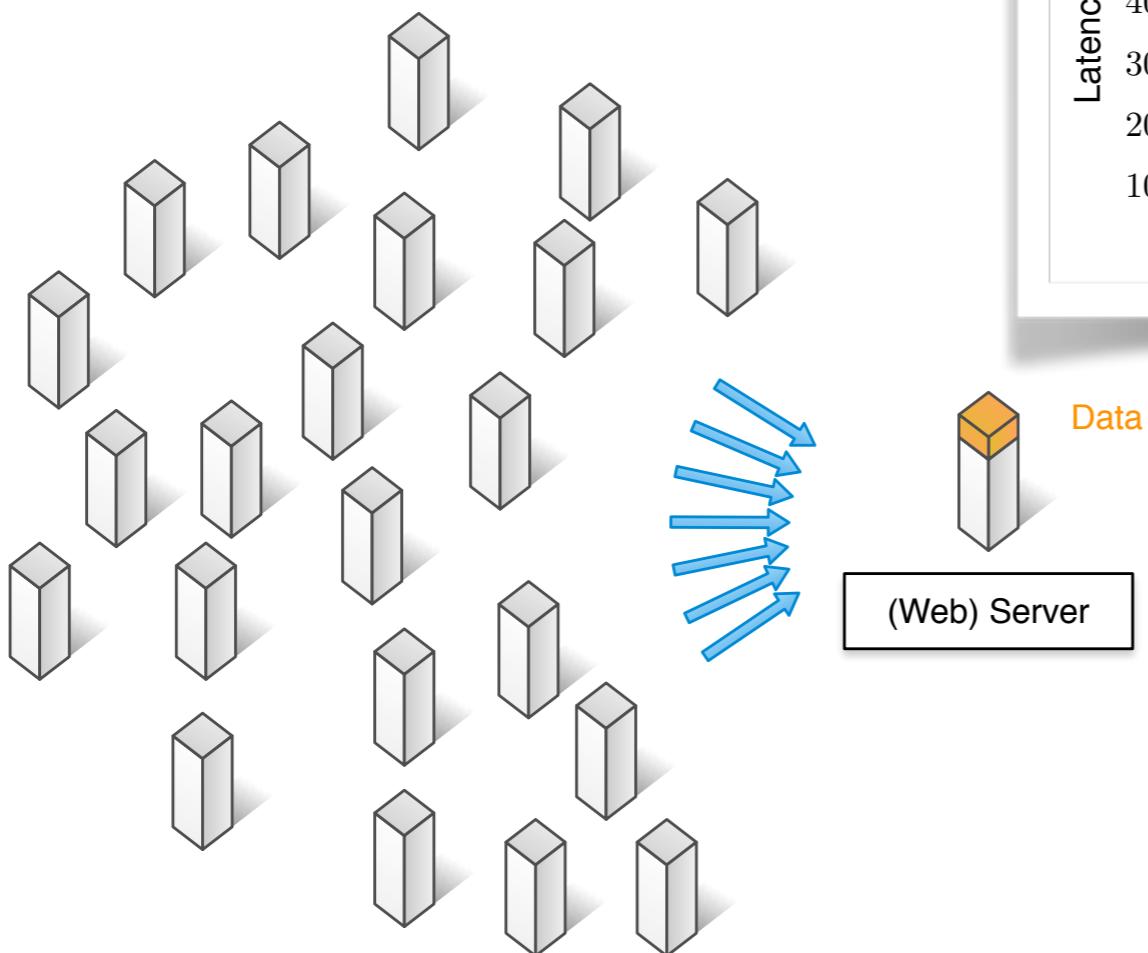
Evaluation

- 30 concurrent clients
- Request delay: 1ms



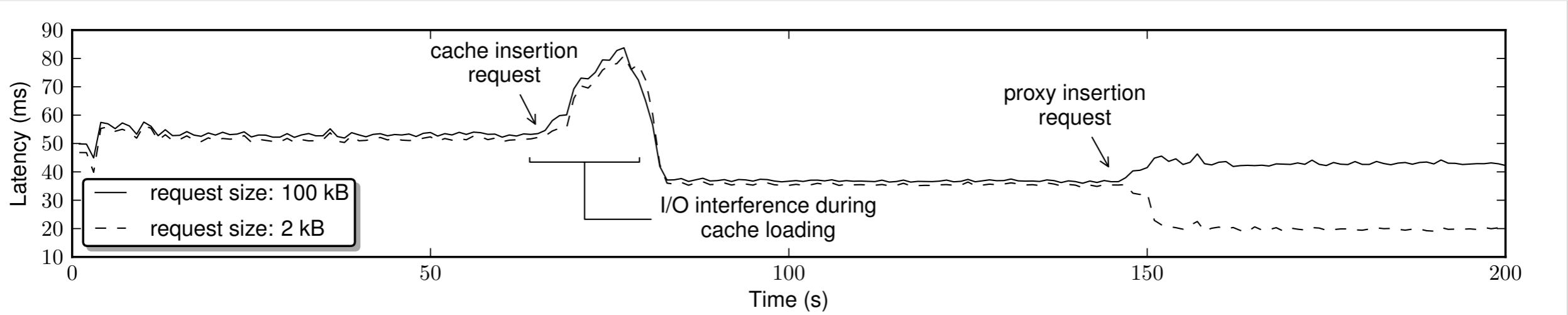
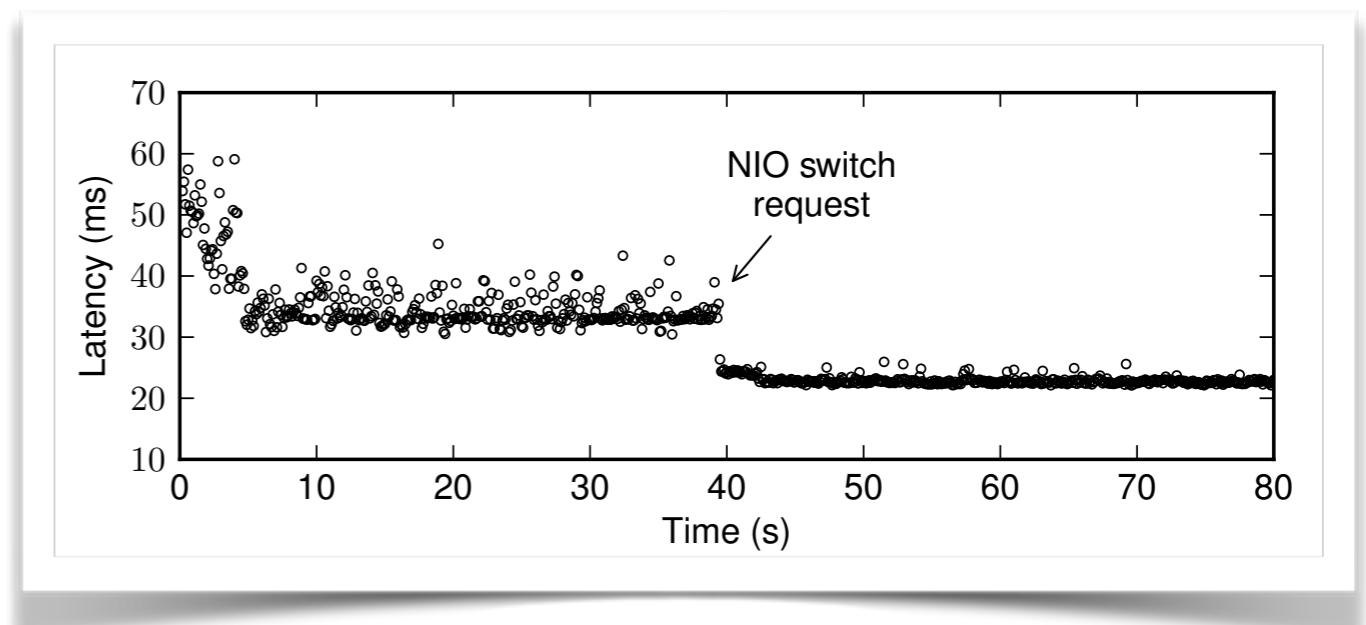
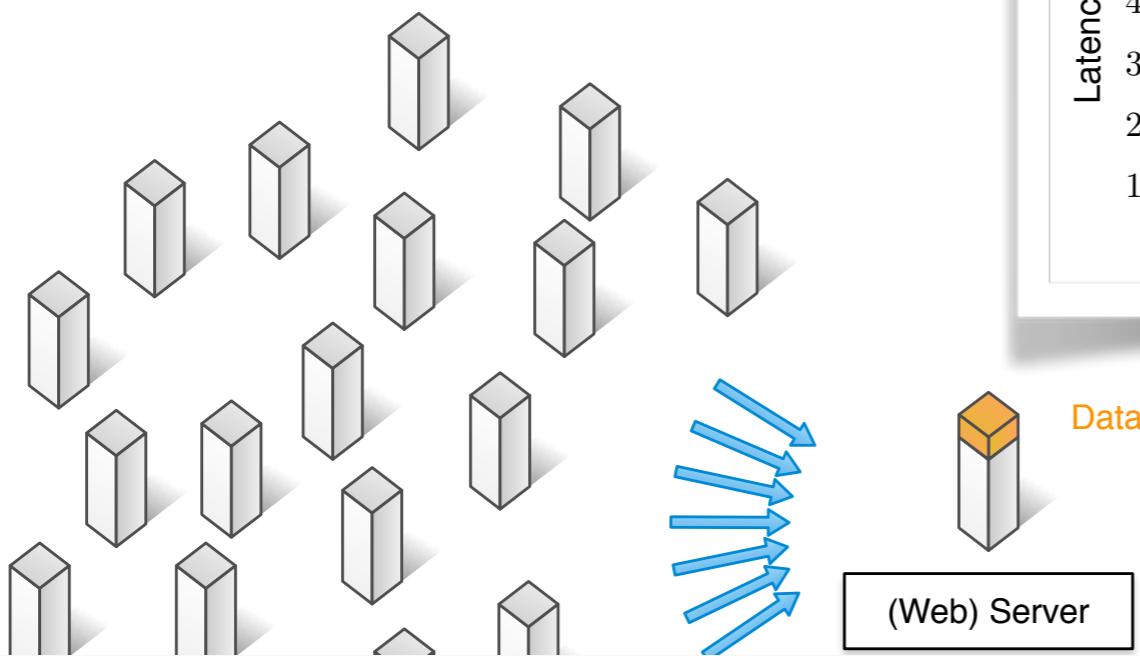
Evaluation

- 30 concurrent clients
- Request delay: 1ms

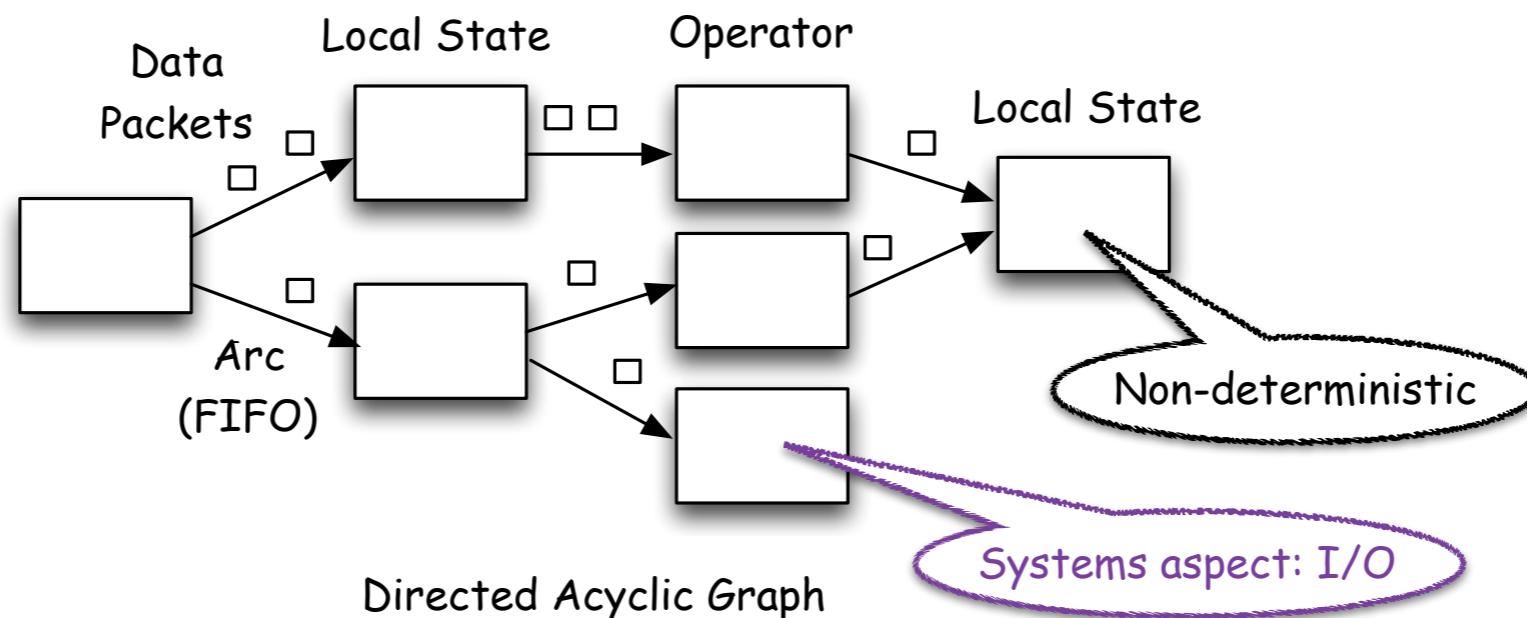


Evaluation

- 30 concurrent clients
- Request delay: 1ms



Dataflow



Database Systems

Data Management Systems

Data Streaming Systems

Server Systems

Embedded Systems

Signal Processing Systems

Parallelism

Heterogeneous Architectures

J. B. Dennis. Data flow supercomputers. Computer, 13(11):48–56, Nov. 1980.

J. P. Morrison. Flow-Based Programming. Nostrand Reinhold, 1994

Arvind and D. E. Culler. Annual review of computer science vol. 1, 1986. chapter Dataflow architectures. Annual Reviews Inc.

Arvind, K. P. Gostelow, and W. Plouffe. Indeterminacy, monitors, and dataflow.. 1977. SOSP '77. ACM.