Ohua: Implicit Dataflow Programming for Concurrent Systems

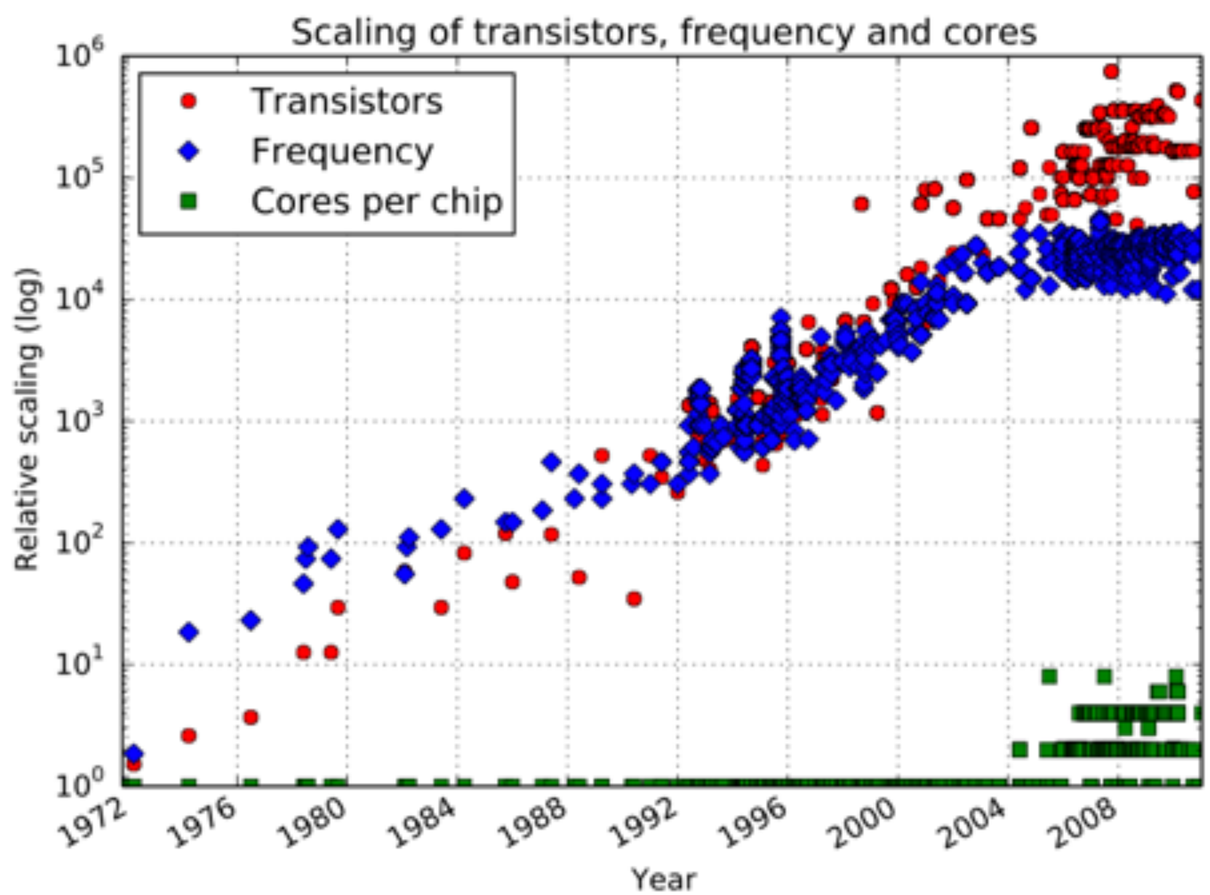
Sebastian Ertel
Compiler Construction Group
TU Dresden,
Germany

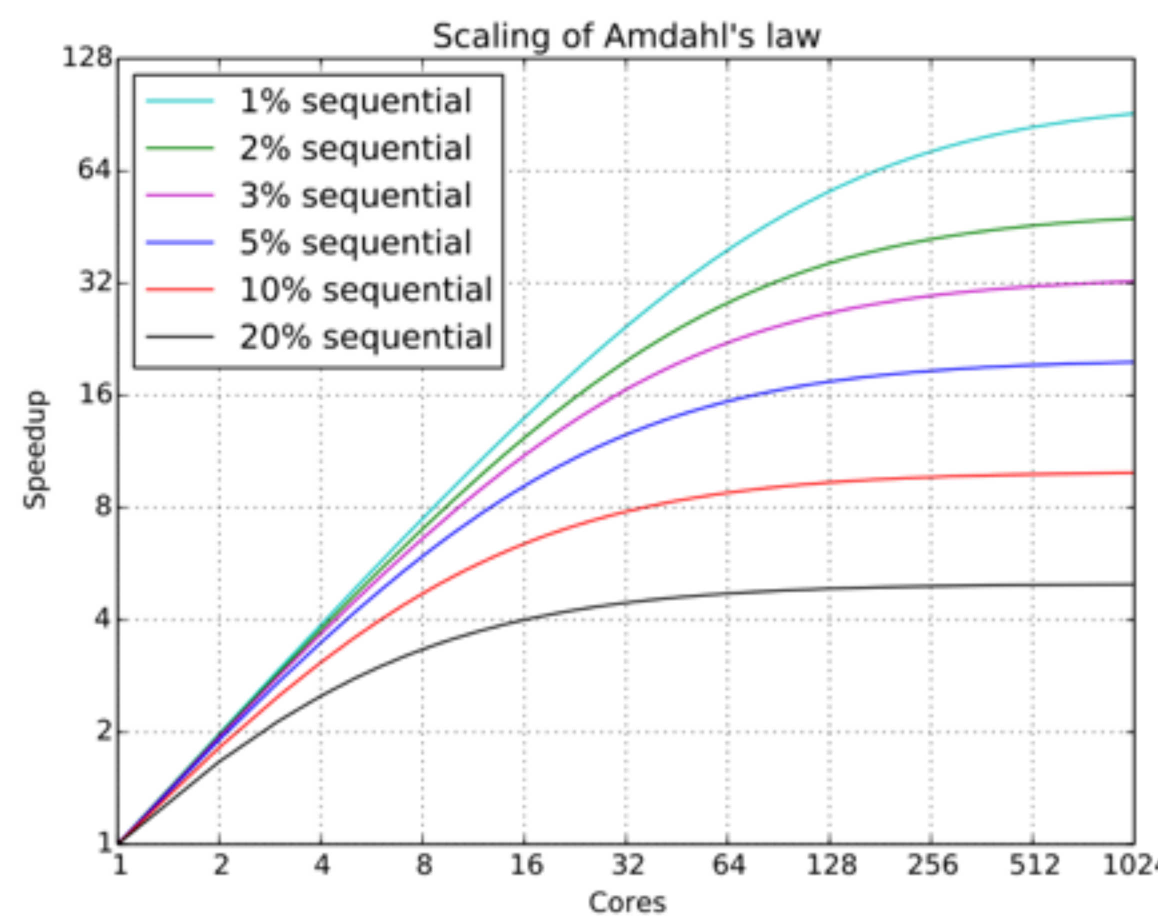
Christof Fetzer
Systems Engineering Group
TU Dresden,
Germany

Pascal Felber
Institut d'informatique
Université de Neuchâtel,
Switzerland

PPPJ 2015

The Multi-Core Future





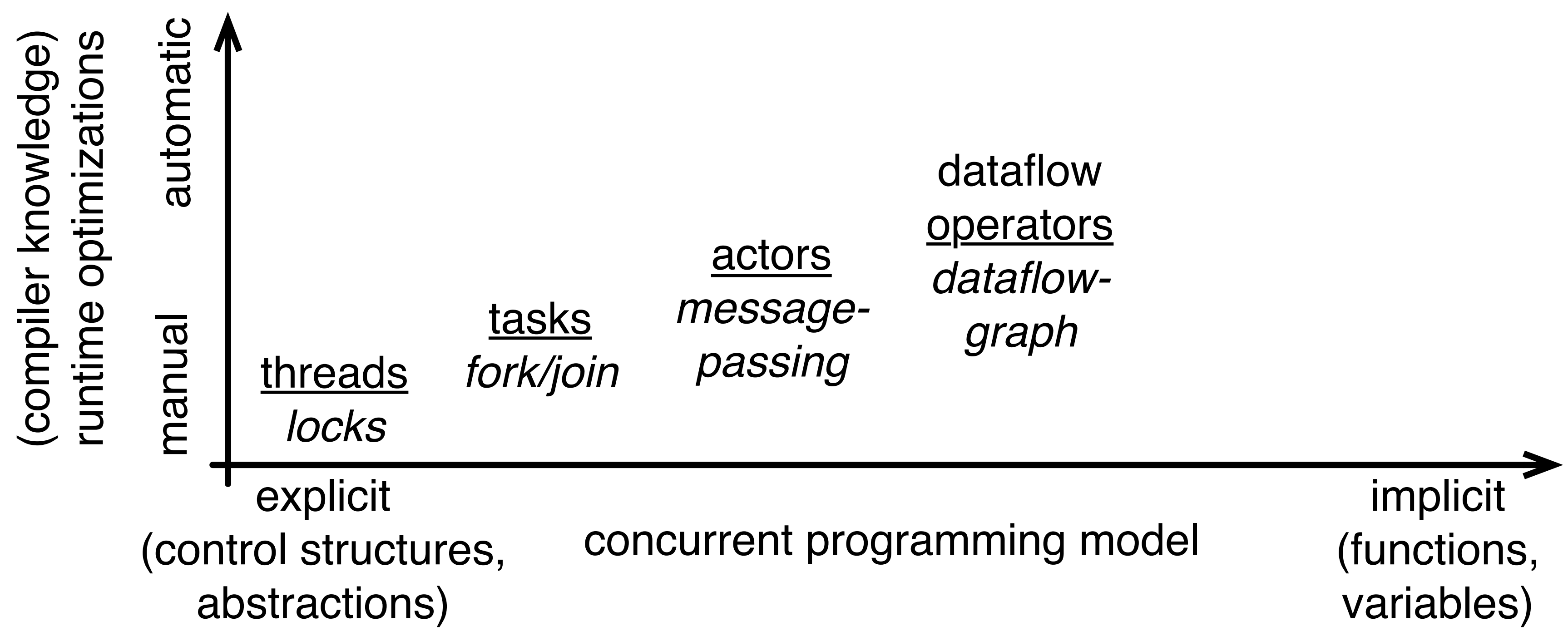
Jons-Tobias Wamhoff. 2014. Exploiting Speculative and Asymmetric Execution on Multicore Architectures (Dissertation)

Parallelism

- Lesson learned (for general purpose programming):
  - Work granularity must compensate scheduling overheads:
    - ➪ Coarse-grained parallelism
    - ➪ Concurrent programming

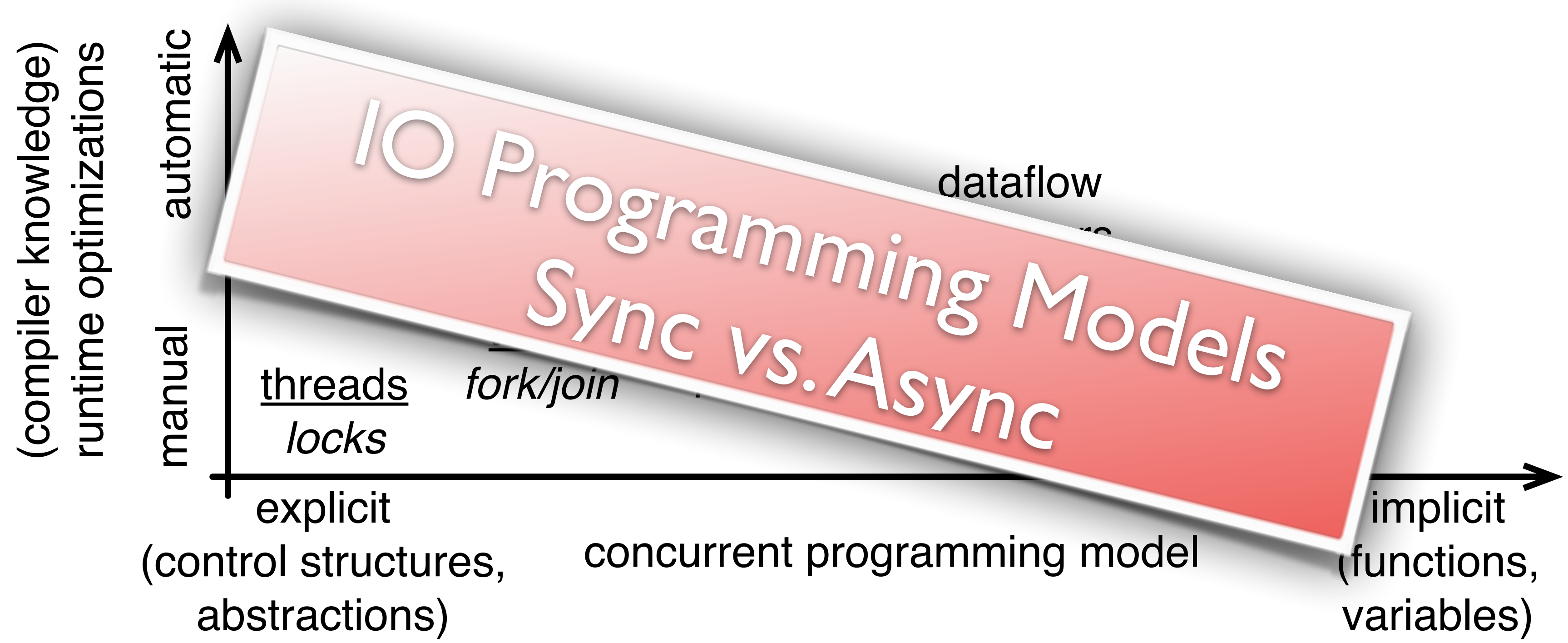
Tim Harris and Satnam Singh. Feedback directed implicit parallelism. ICFP '07

Concurrent Programming



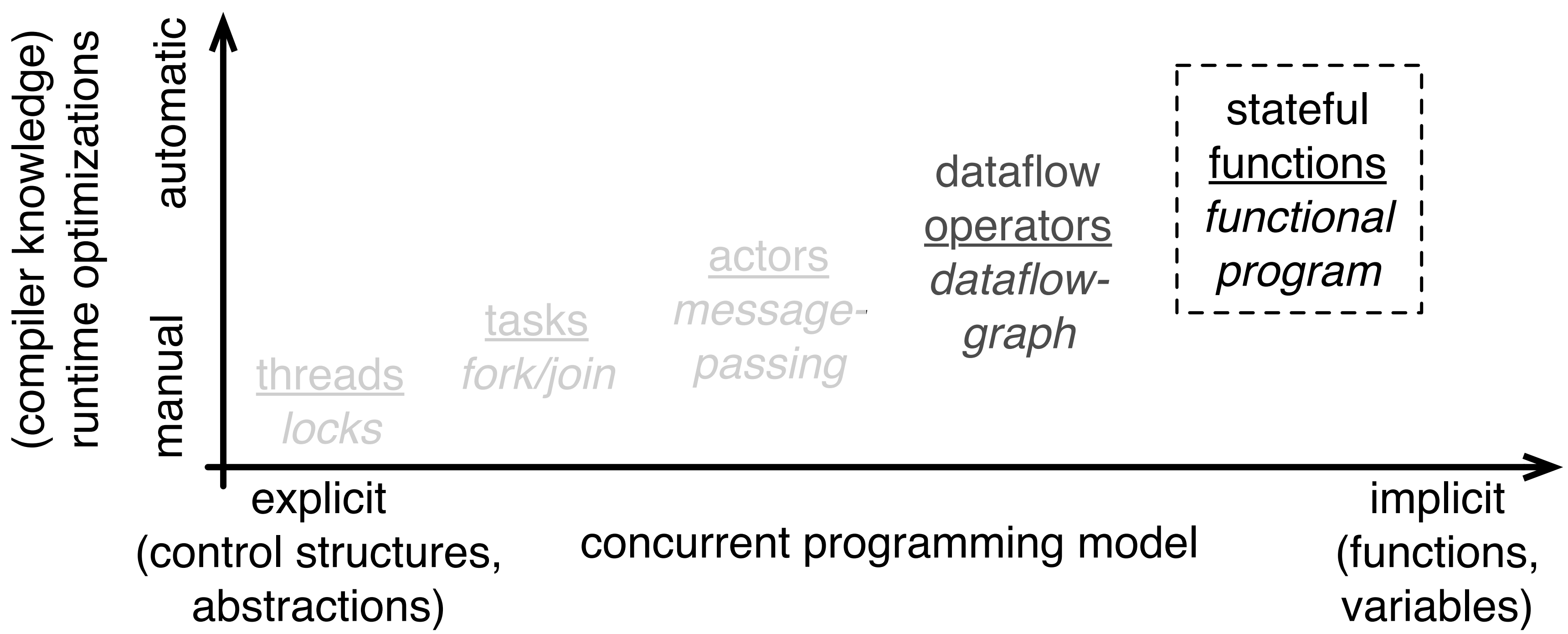
- All models depart from sequential programming style.
  - ➪ Scalability in programming effort suffers.
  - ➪ Limited compiler support for optimizations.

Concurrent Programming



- All models depart from sequential programming style.
  - ➪ Scalability in programming effort suffers.
  - ➪ Limited compiler support for optimizations.

Concurrent Programming



- Key observation:
  - ➪ Algorithms are composed out of functionality.

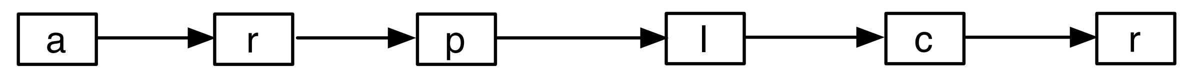
From dataflow programming to stateful functional programming (SFP)

- Example: simple web server

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

  public WebServer() {
    // explicit dataflow graph construction
    a = makeFilter(Accept.class);
    r = makeFilter(Read.class);
    p = makeFilter(Parse.class);
    l = makeFilter(Load.class);
    c = 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 Synthetizer(a).start();
    super.start();
  }
}
```



Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. 2007. Streamflex: high-throughput stream programming in java. OOPSLA '07

From dataflow programming to stateful functional programming (SFP)

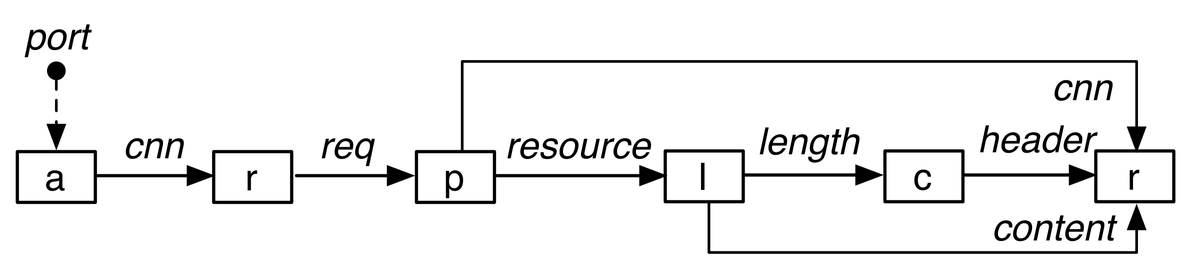
- Example: simple web server

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

  public WebServer() {
    // explicit dataflow graph construction
    a = makeFilter(Accept.class);
    r = makeFilter(Read.class);
    p = makeFilter(Parse.class);
    l = makeFilter(Load.class);
    c = 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 Synthetizer(a).start();
    super.start();
  }
}
```


```clojure
(defn start [port]
  (ohua
    ; most "explicit"/fine-grained data dependency matching
    (let [[cnn req] (read (accept port))]
      (let [[_ resource _] (parse req)]
        (let [[content length] (load resource)]
          (reply cnn (compose length) content))))))
```



Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. 2007. Streamflex: high-throughput stream programming in java. OOPSLA '07

From dataflow programming to stateful functional programming (SFP)

- Example: simple web server

```
class FileLoad extends Filter {
  Channel<String> in, out;
  Map<String, String> cache = new HashMap<>();

  void work() {
    // explicit channel control
    String resource = in.take();
    String contents = null;
    // load file data from disk or cache (omitted)
    out.put(contents);
  }
}
```

Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. 2007. Streamflex: high-throughput stream programming in java. OOPSLA '07

From dataflow programming to stateful functional programming (SFP)

- Example: simple web server

```
class FileLoad extends Filter {
  Channel<String> in, out;
  Map<String, String> cache = new HashMap<>();

  void work() {
    // explicit channel control
    String resource = in.take();
    String contents = null;
    // load file data from disk or cache (omitted)
    out.put(contents);
  }
}
```

```
class FileLoad {
  Map<String, String> cache = new HashMap<>();

  @Function
  String load(String resource) {
    String contents = null;
    // load file data from disk or cache (omitted)
    return contents;
  }
}
```

- Functions instead of channels!
- Stateful functions ➪ state encapsulation
  - Implemented in imperative language/style.

Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. 2007. Streamflex: high-throughput stream programming in java. OOPSLA '07

From dataflow programming to stateful functional programming (SFP)

- Example: simple web server

```java
class FileLoad extends Filter {
  Channel<String> in, out;
  Map<String, String> cache = new HashMap<>();

  void work() {
    // explicit channel control
    String resource = in.take();
    String contents = null;
    // load file data from disk or cache (omitted)
    out.put(contents);
  }
}
```

```java
class FileLoad {
  Map<String, String> cache = new HashMap<>();

  @Function
  String load(String resource) {
    String contents = null;
    // load file data from disk or cache (omitted)
    return contents;
  }
}
```

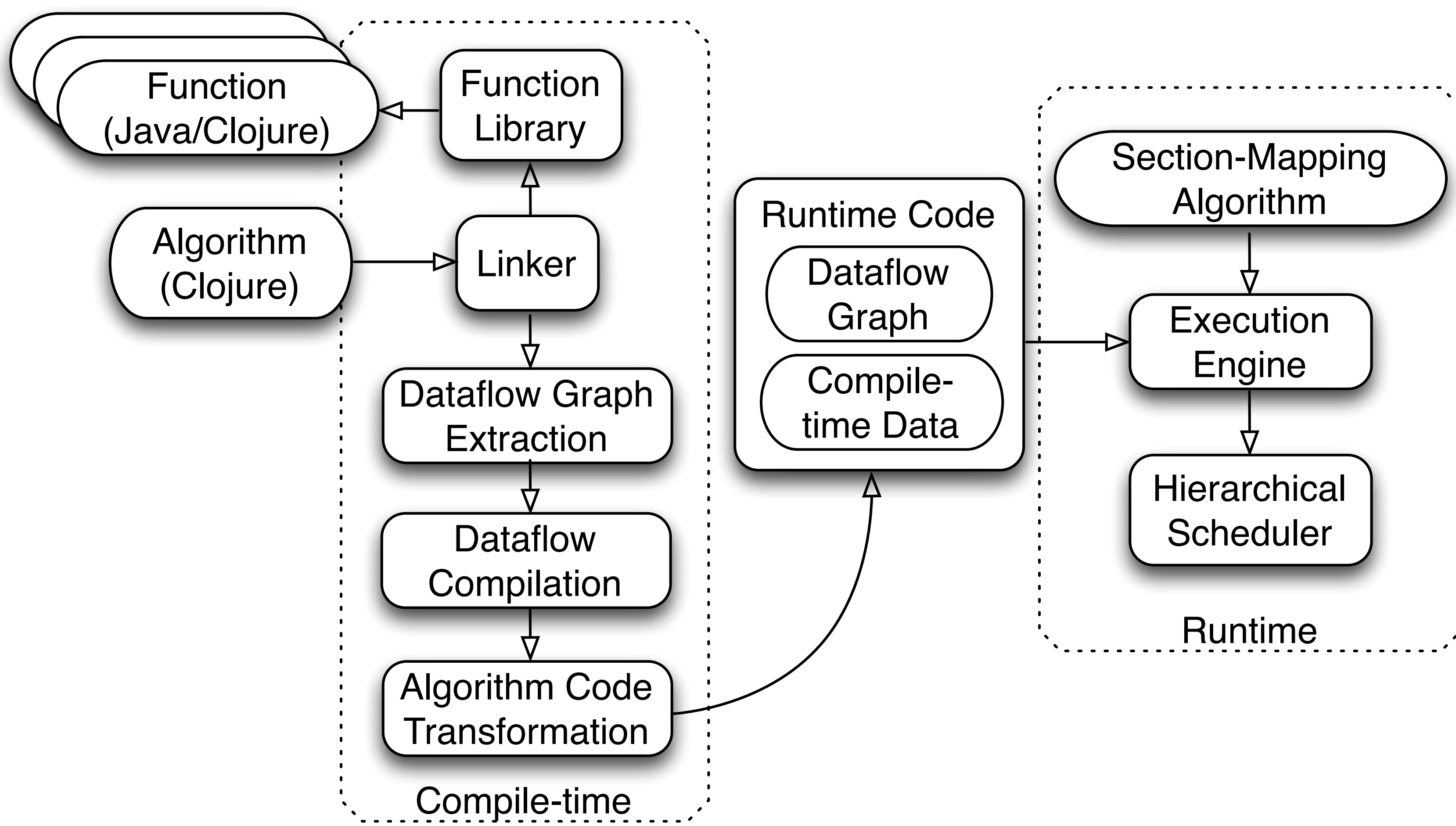
```clojure
(ns com.server.FileLoad
  (:gen-class :state cache :init init
    :methods [[^{Function} load [String] String]]))

(defn -init [] [[] {}]) ; create a hash map
(defn -load [this resource]
  (if (contains? (.cache this) resource)
    (get (.cache this) resource)
    (let [content (slurp resource)]
      (set! (.cache this) (assoc (.cache this) resource content))
      content)))
```

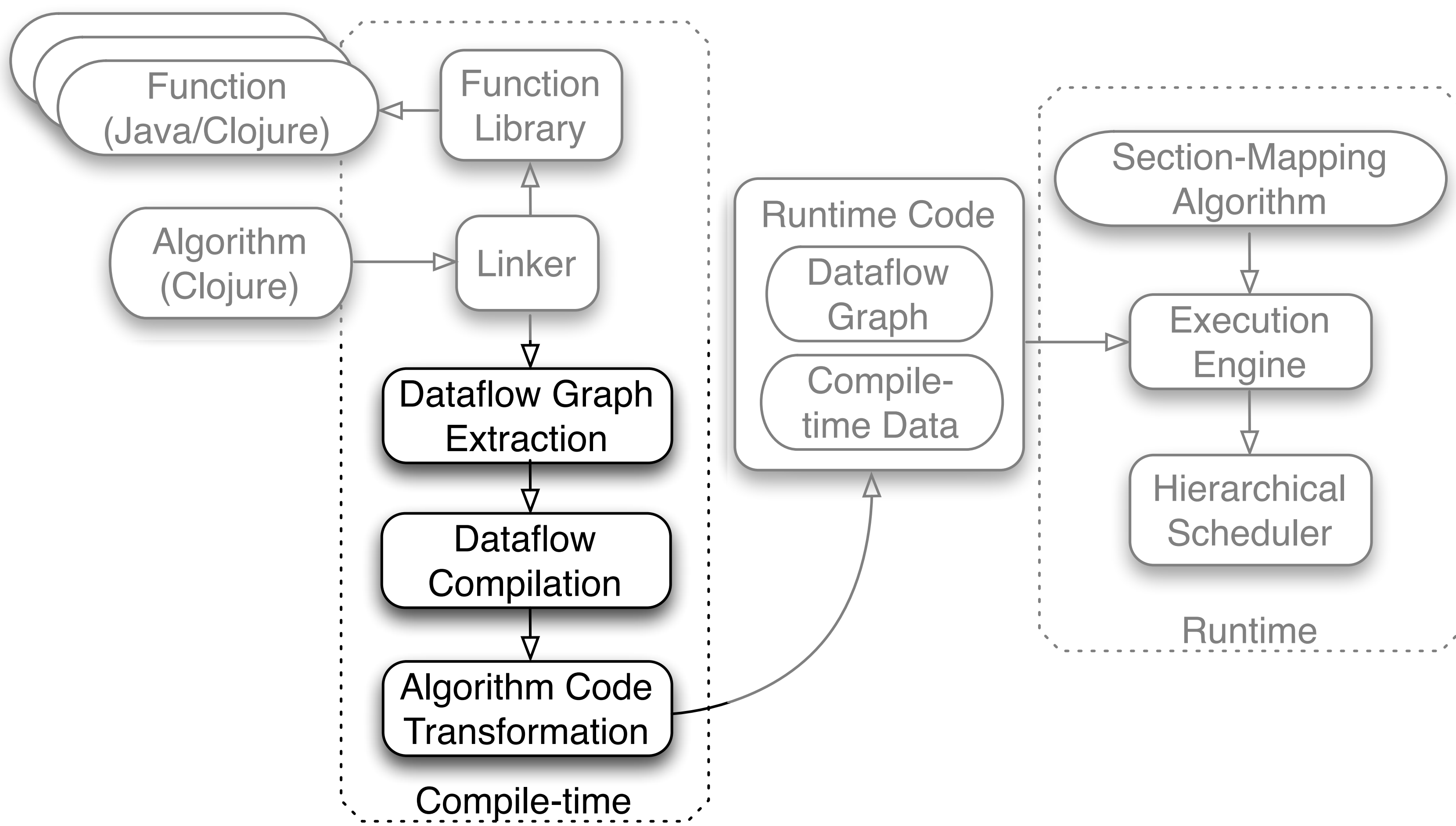
- Functions instead of channels!
- Stateful functions ➪ state encapsulation
  - Implemented in imperative language/style.

Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. 2007. Streamflex: high-throughput stream programming in java. OOPSLA '07

Ohua


Function
(Java/Clojure)
Function
Library
Algorithm
(Clojure)
Linker
Dataflow Graph
Extraction
Dataflow
Compilation
Algorithm Code
Transformation
Compile-time
Runtime Code
Dataflow
Graph
Compile-
time Data
Section-Mapping
Algorithm
Execution
Engine
Hierarchical
Scheduler
Runtime

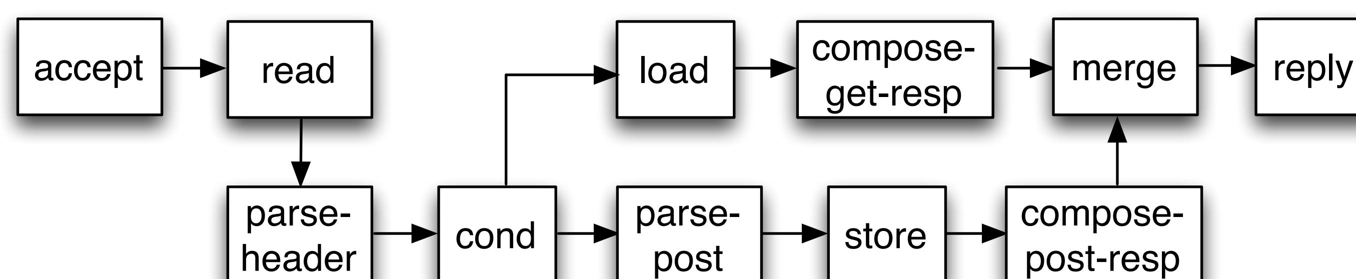
Ohua


Function
(Java/Clojure)
Function
Library
Algorithm
(Clojure)
Linker
Dataflow Graph
Extraction
Dataflow
Compilation
Algorithm Code
Transformation
Compile-time
Runtime Code
Dataflow
Graph
Compile-
time Data
Section-Mapping
Algorithm
Execution
Engine
Hierarchical
Scheduler
Runtime

From Control Flow to Dataflow

- Construction of a control flow dependence graph.

```
(ohua
  (let [[type req] (-> 80 accept read parse-header)]
   (if (= type "GET")
    (-> req load compose-get-resp)
    (-> req parse-post store compose-post-resp))
   reply))
```



- Most special forms of Clojure supported.

Micah Beck, Richard Johnson, and Keshav Pingali. 1991. From control flow to dataflow. J. Parallel Distrib. Comput.

Closures and Destructuring

- Closures in combination with destructuring create opportunities for enhanced parallelism.
- || macro construction for parallel request handling:

Closures and Destructuring

- Closures in combination with destructuring create opportunities for enhanced parallelism.
- || macro construction for parallel request handling:

```
(ohua
 (let [cnn (accept 80)]
   (let [[one two three] (balance-data cnn)]
     (-> one read parse load compose reply)
     (-> two read parse load compose reply)
     (-> three read parse load compose reply))))
```

Closures and Destructuring

- Closures in combination with destructuring create opportunities for enhanced parallelism.
- || macro construction for parallel request handling:

```
(ohua
 (let [cnn (accept 80)]
   (let [[one two three] (balance-data cnn)]
     (-> one read parse load compose reply)
     (-> two read parse load compose reply)
     (-> three read parse load compose reply))))
```

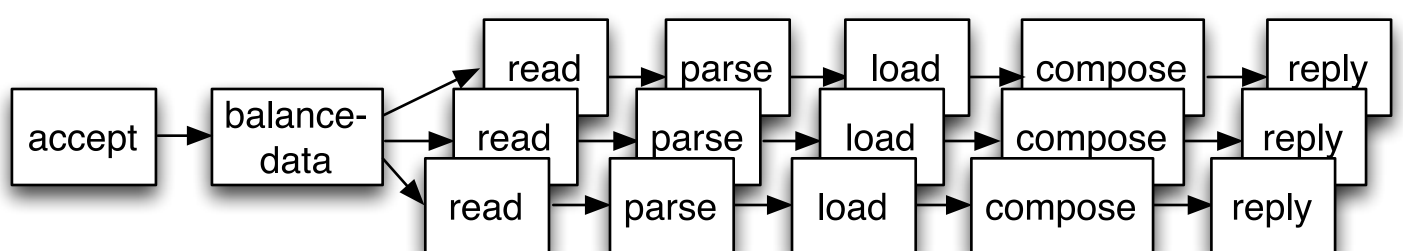
```
 one two three
[cnn nil nil]
[nil cnn nil]
[nil nil cnn]
```

Closures and Destructuring

- Closures in combination with destructuring create opportunities for enhanced parallelism.
- || macro construction for parallel request handling:

```
(ohua
 (let [cnn (accept 80)]
   (let [[one two three] (balance-data cnn)]
     (-> one read parse load compose reply)
     (-> two read parse load compose reply)
     (-> three read parse load compose reply))))
```

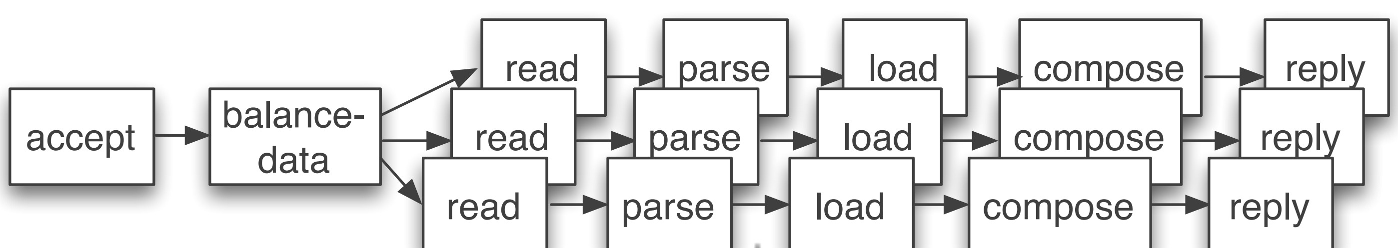
```
 one two three
[cnn nil nil]
[nil cnn nil]
[nil nil cnn]
```



Closures and Destructuring

- Closures in combination with destructuring create opportunities for enhanced parallelism.
- || macro construction for parallel request handling:

```
(ohua
 (let [cnn (accept 80)]
   (let [[one two three] (balance-data cnn)]
     (-> one read parse load compose reply)
     (-> two read parse load compose reply)
     (-> three read parse load compose reply))))
```

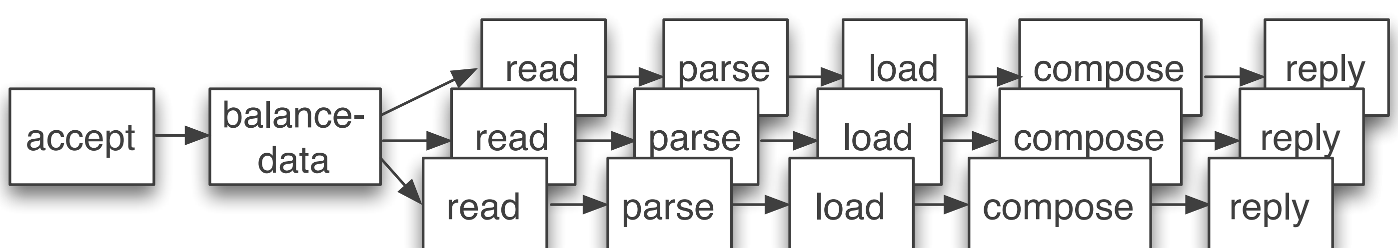


Beware:
Each operator/function *invocation* has its own state!

Closures and Destructuring

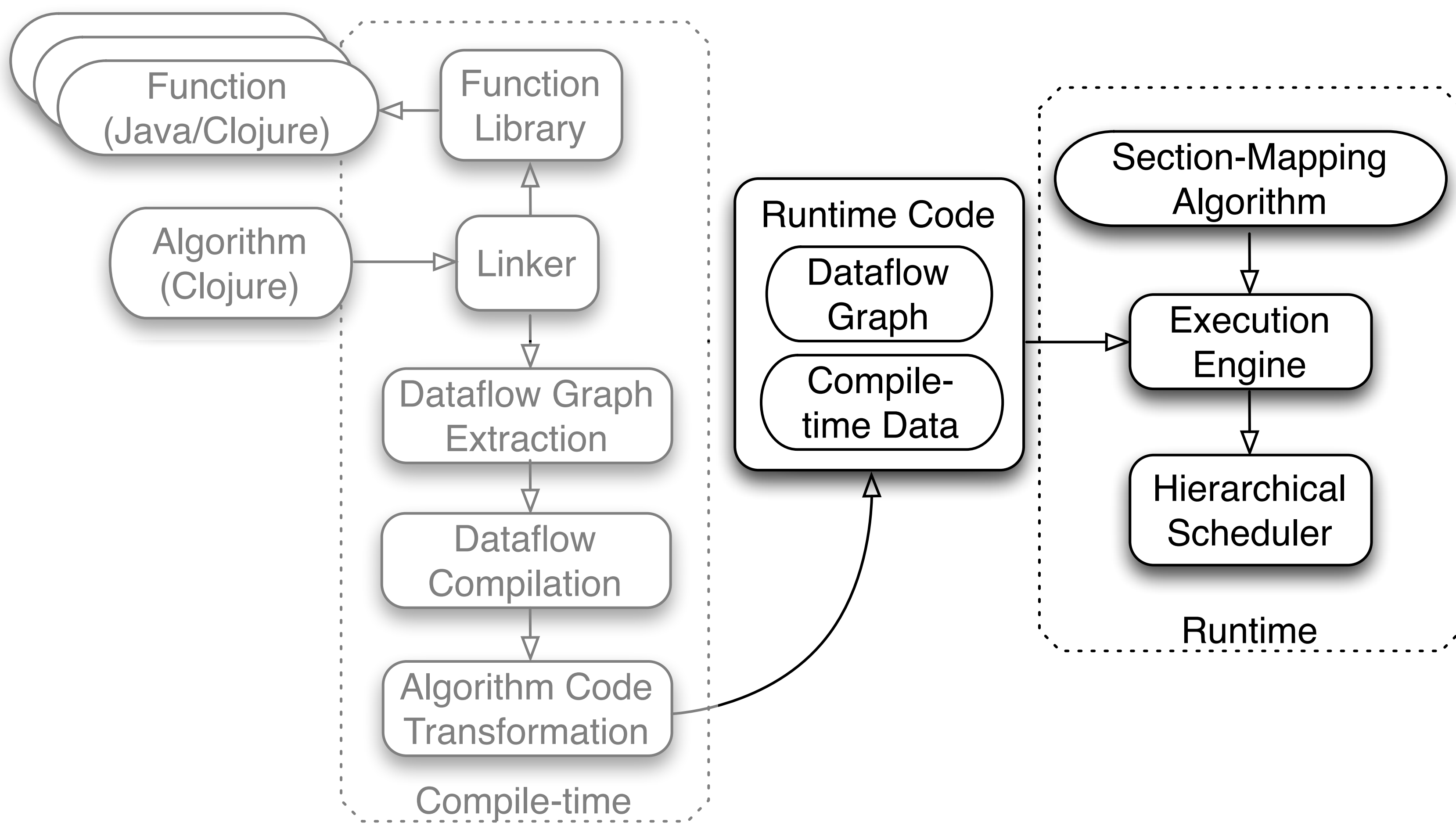
- Closures in combination with destructuring create opportunities for enhanced parallelism.
- || macro construction for parallel request handling:

```
(ohua
 (let [cnn (accept 80)]
   (let [[one two three] (balance-data cnn)]
     (-> one read parse load compose reply)
     (-> two read parse load compose reply)
     (-> three read parse load compose reply))))
```



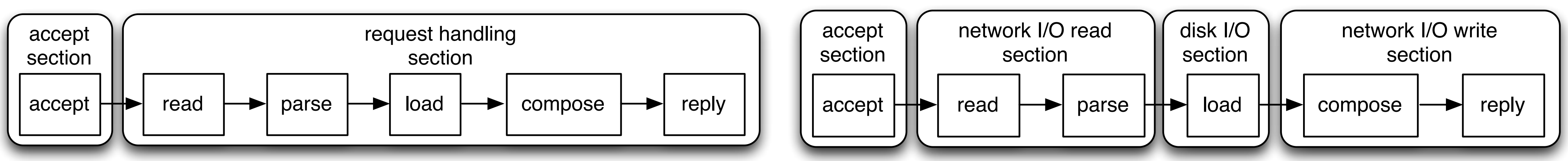

```
(ohua
  (let [cnn (accept 80)]
    (|| 3 cnn (-> read parse load compose reply))))
```

Ohua


Function (Java/Clojure)
Function Library
Algorithm (Clojure)
Linker
Dataflow Graph Extraction
Dataflow Compilation
Algorithm Code Transformation
Compile-time
Runtime Code
Dataflow Graph
Compile-time Data
Section-Mapping Algorithm
Execution Engine
Hierarchical Scheduler
Runtime

Sections

- Dataflow execution model: pipeline parallelism.
- Section: set of one or more operators.

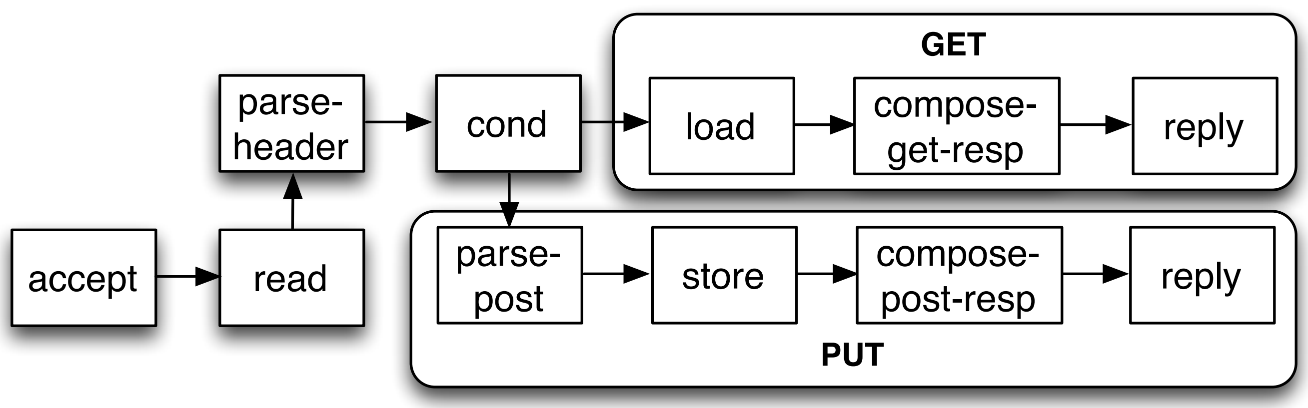


- Construct section mapping from execution restrictions.

- Hierarchical scheduling:
  - Sections executed across thread pool ➪ parallel.
  - Operators scheduled cooperatively ➪ sequential.

Synchronization

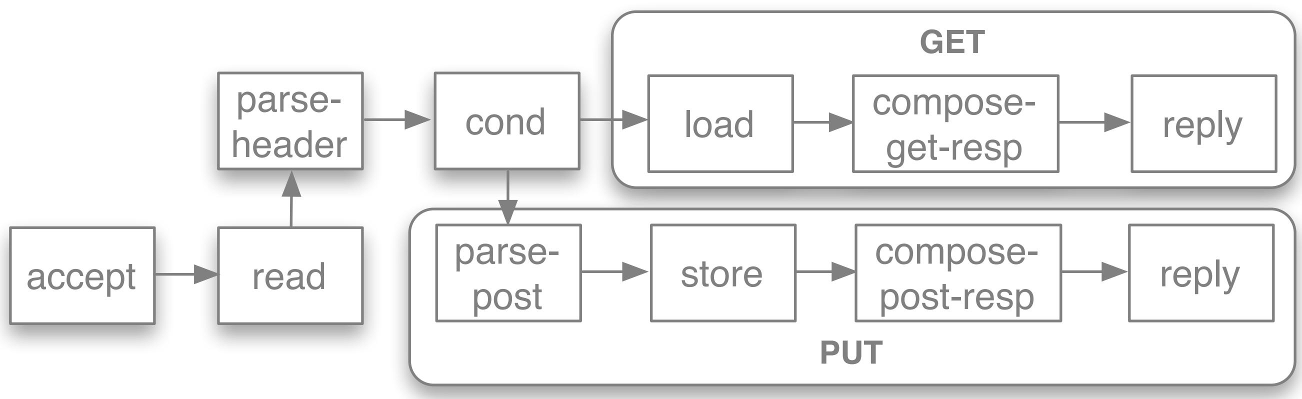
- Sections provide lock-free synchronization!
- Consider parallel request handling by type:



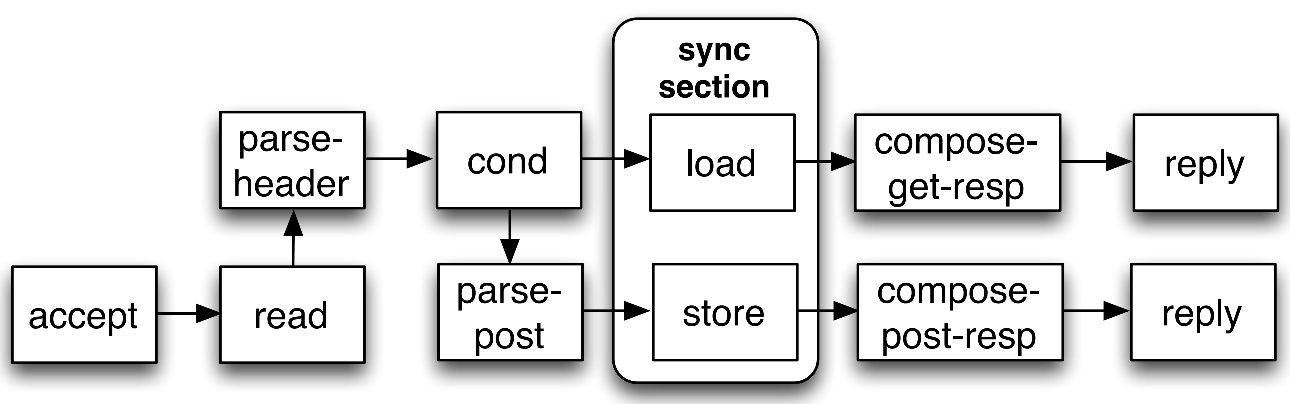
- Shared external resource (file) ➪ inconsistency.

Synchronization

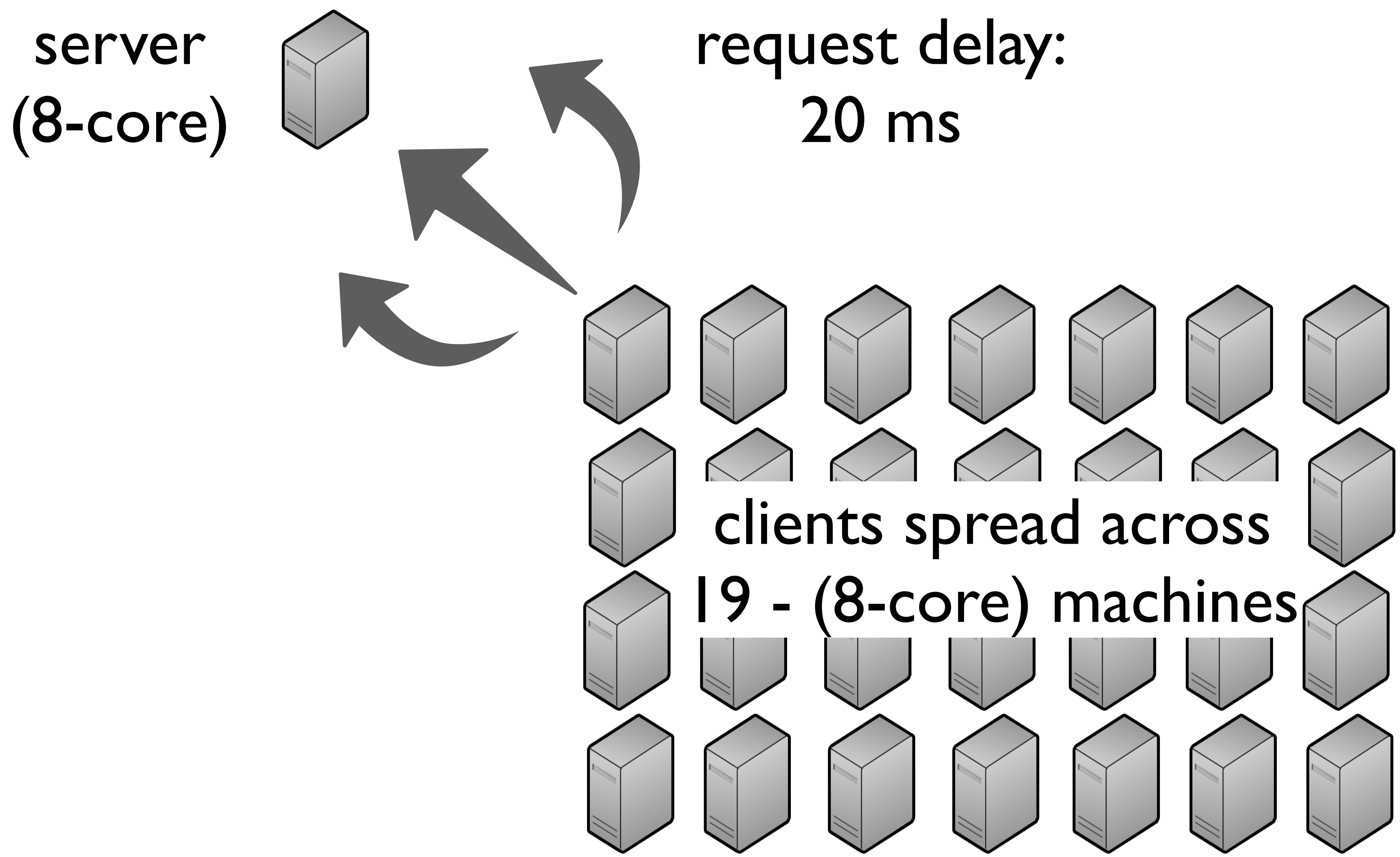
- Sections provide lock-free synchronization!
- Consider parallel request handling by type:



- Shared external resource (file) ➪ inconsistency.



Experimental Setup

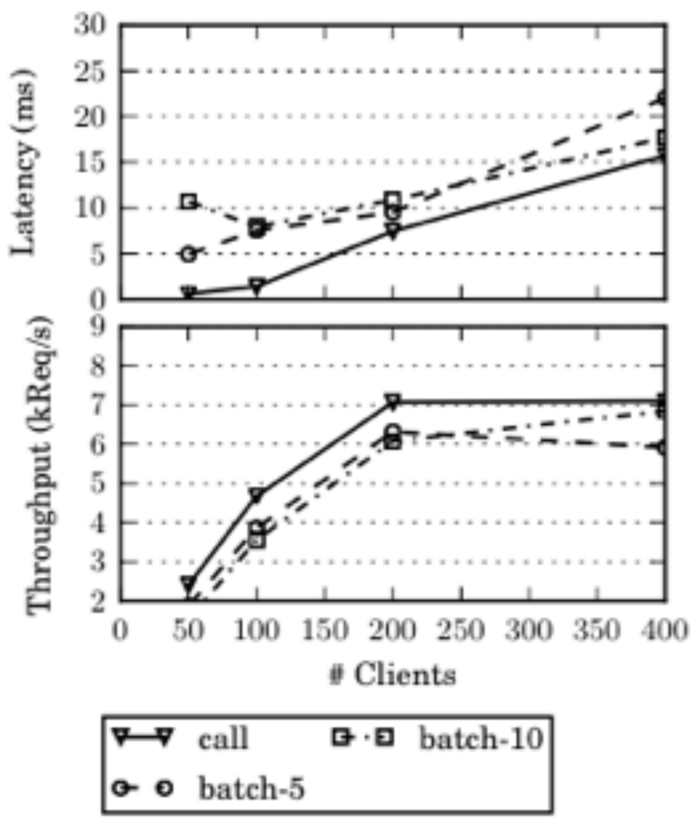


Section Strategies

- Goal: web server fine-tuning without code changes.

Section Strategies

- Goal: web server fine-tuning without code changes.

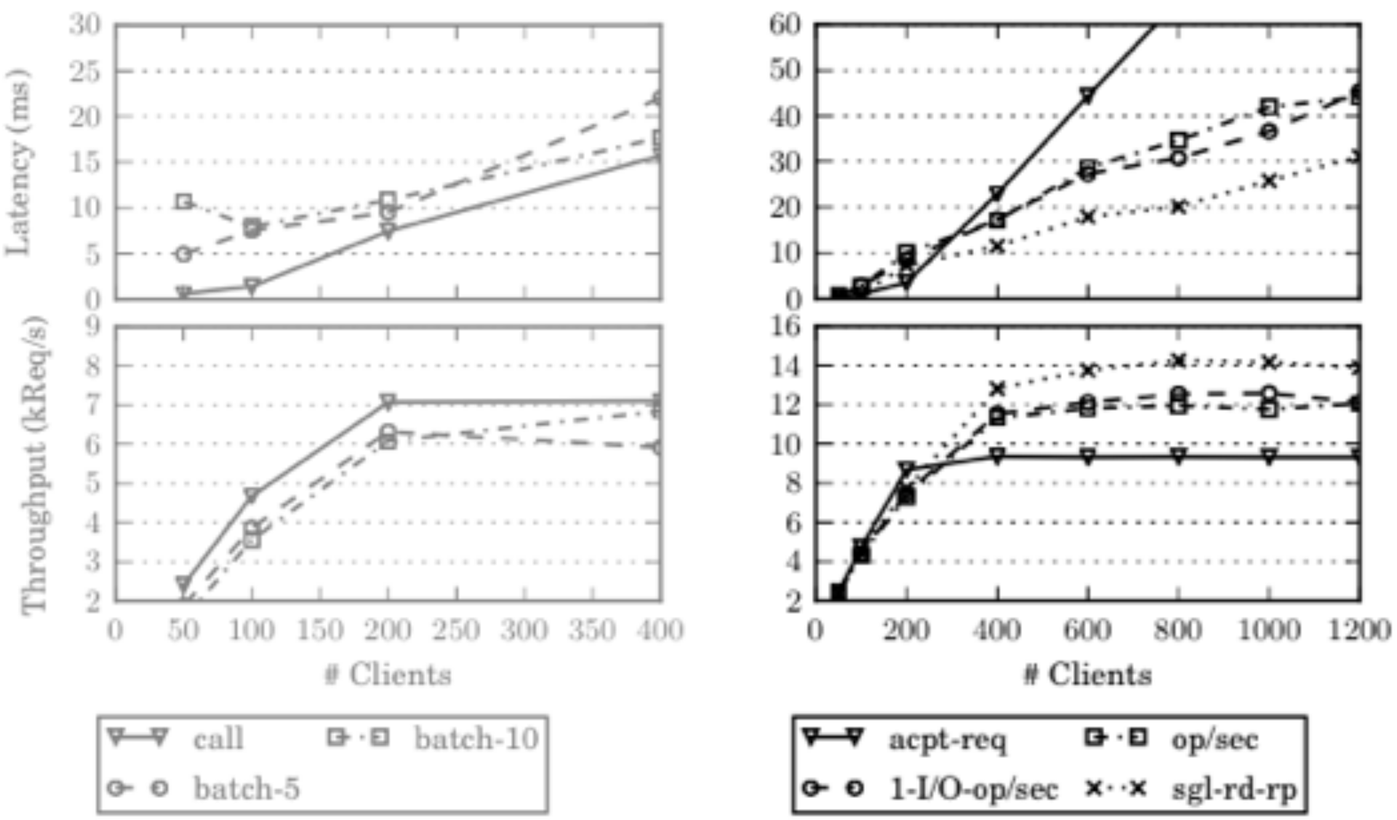


(a) Operator Scheduling Overhead

- Best deployment strategy: call

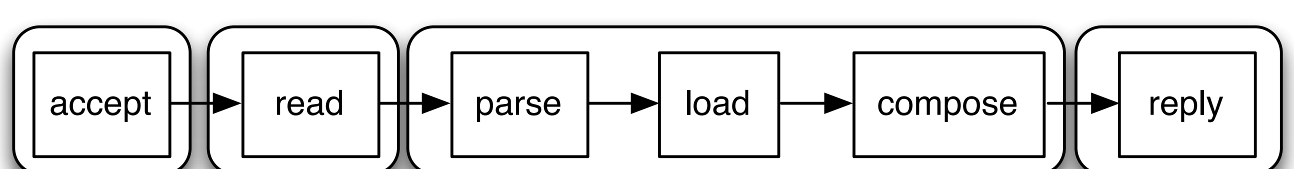
Section Strategies

- Goal: web server fine-tuning without code changes.



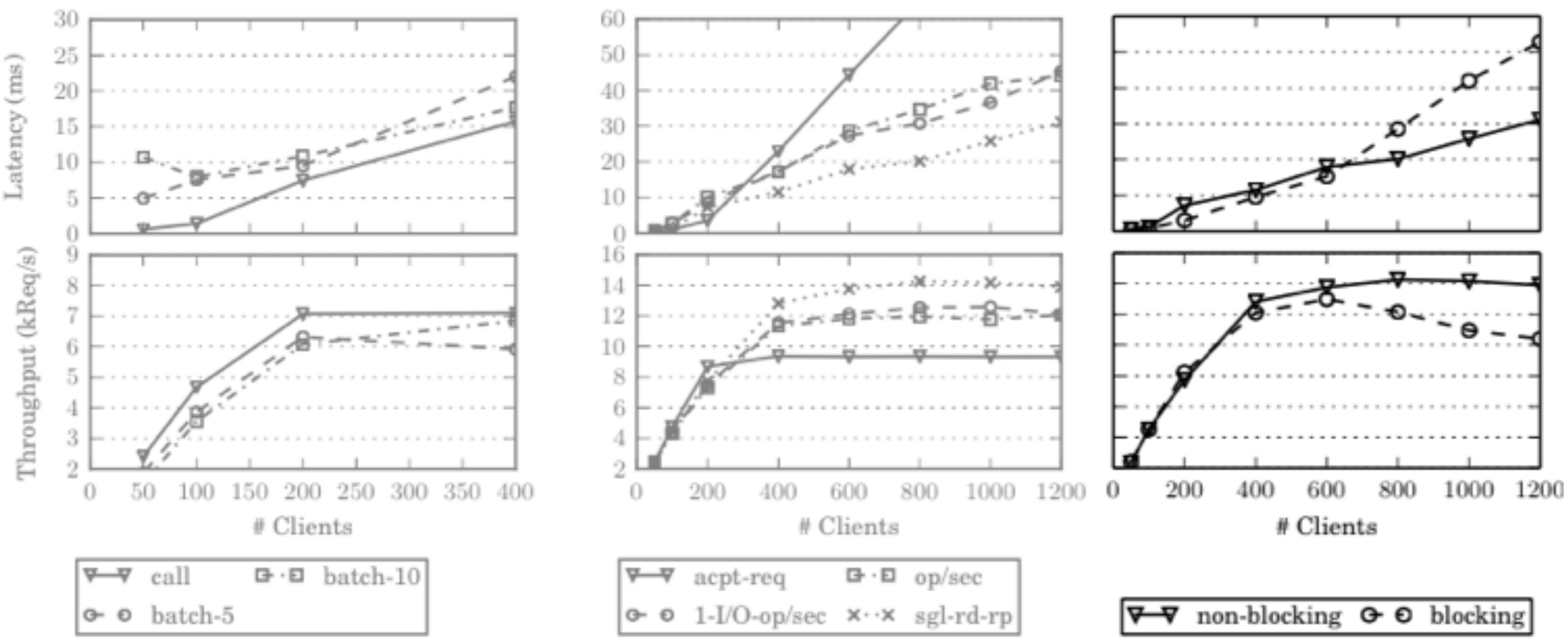
(a) Operator Scheduling Overhead (b) Section Mapping Comparison

- Best deployment strategy: call/sgl-rd-rp



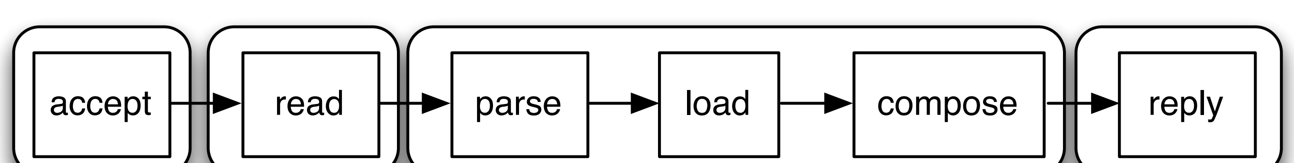
Section Strategies

- Goal: web server fine-tuning without code changes.



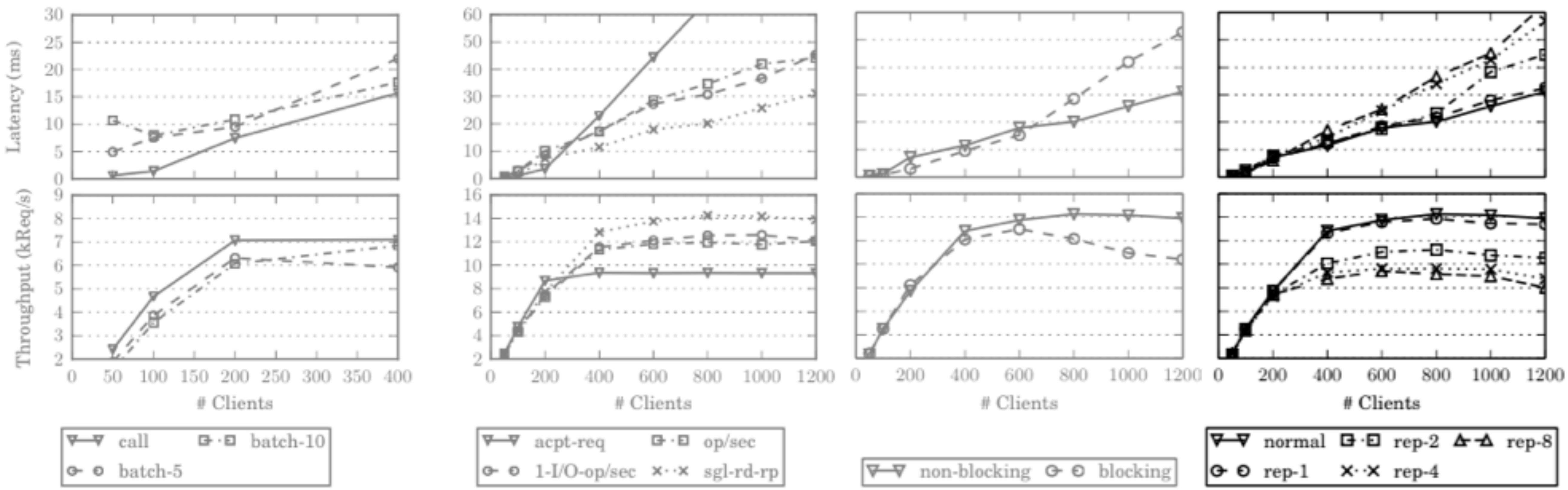
(a) Operator Scheduling Overhead (b) Section Mapping Comparison (c) Section Scheduling Overhead

- Best deployment strategy: call/sgl-rd-rp/non-blocking



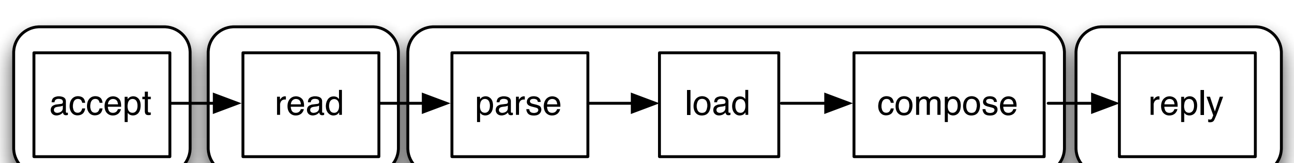
Section Strategies

- Goal: web server fine-tuning without code changes.

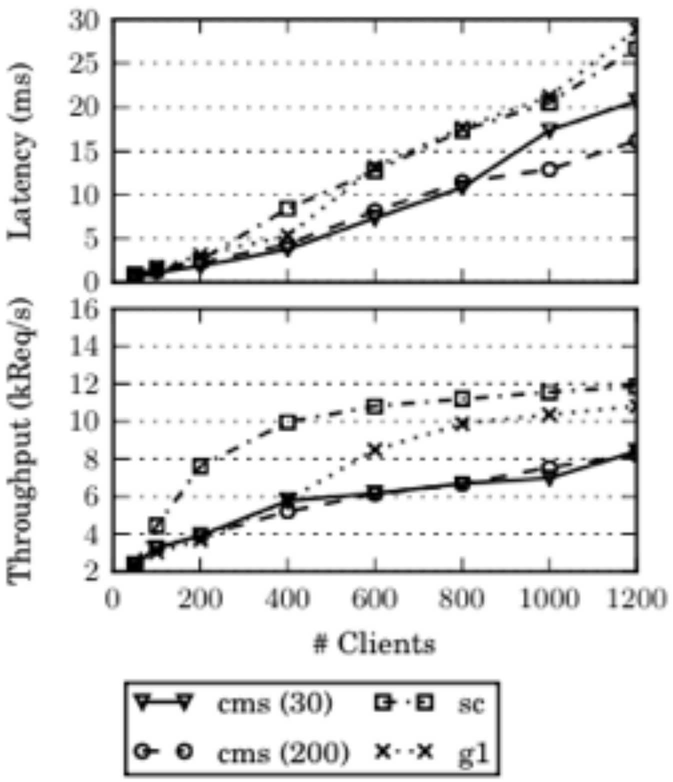


(a) Operator Scheduling Overhead (b) Section Mapping Comparison (c) Section Scheduling Overhead **(d) Data Parallelism Impact**

- Best deployment strategy: call/sgl-rd-rp/non-blocking/normal

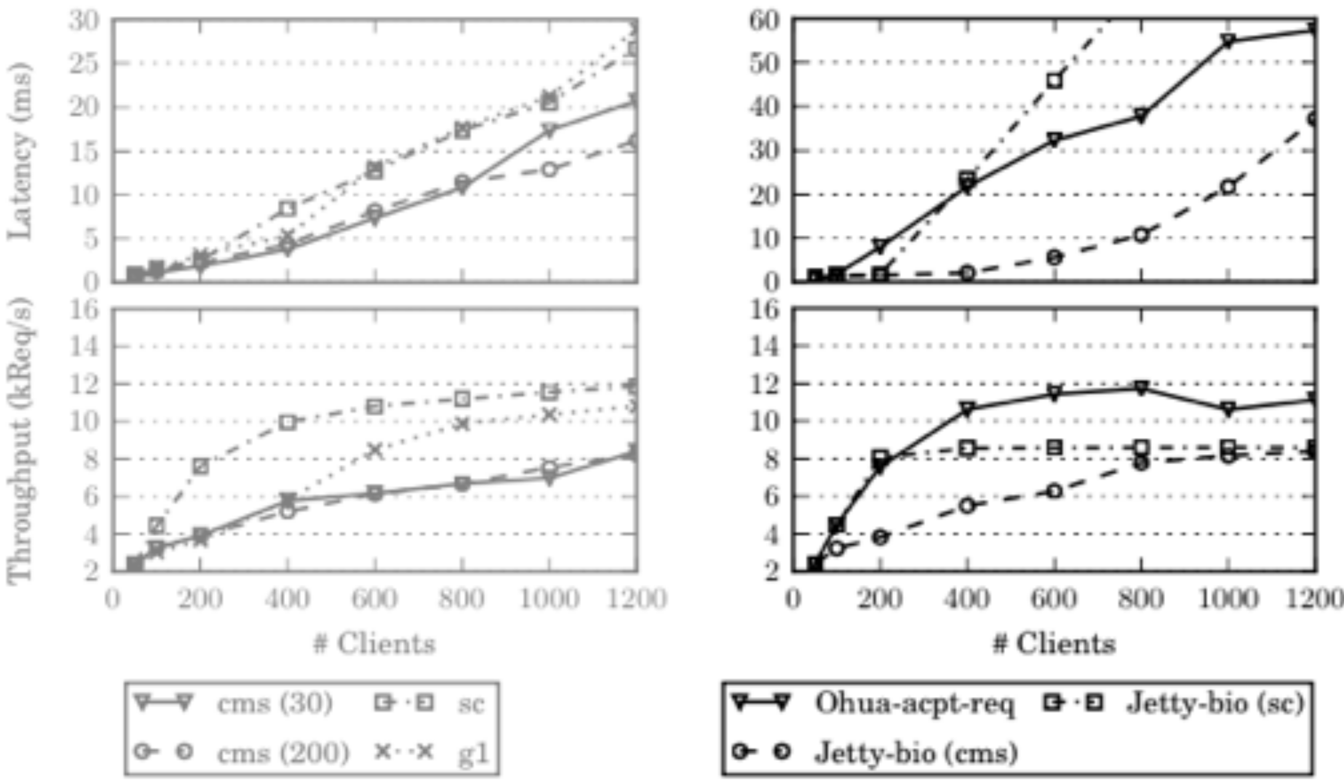


Jetty vs. Ohua



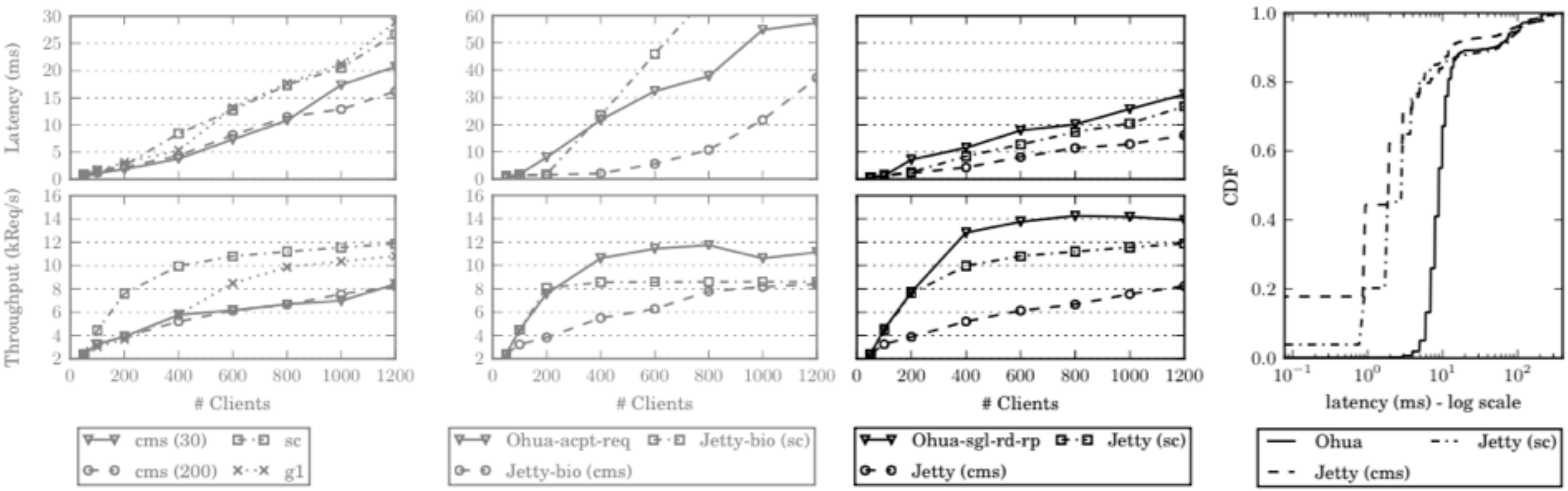
(a) Jetty GC

Jetty vs. Ohua



(a) Jetty GC (b) BIO

Jetty vs. Ohua



(a) Jetty GC (b) BIO (c) Async NIO (d) Req. latency

- Ohua is ➪ comparable to Jetty (NIO) in terms of latency,
  ➪ outperforms Jetty (NIO) in terms of throughput,
  ➪ but without pre-selecting the GC and
  ➪ without mixing programming models.

Stateful Functional Programming (SFP)

- SFP programming model:

  program = *declarative* algorithms + *stateful* functions

- Benefits:
  - ➪ Scalable system construction via algorithm composition.
  - ➪ Implicit parallelism for algorithms.

- Future work: (tip of the ice berg)
  - ➪ No deadlocks or data races.
  - ➪ Leverage dataflow experience for runtime optimizations.

Thanks for your attention!

Questions?

https://clojars.org/ohua

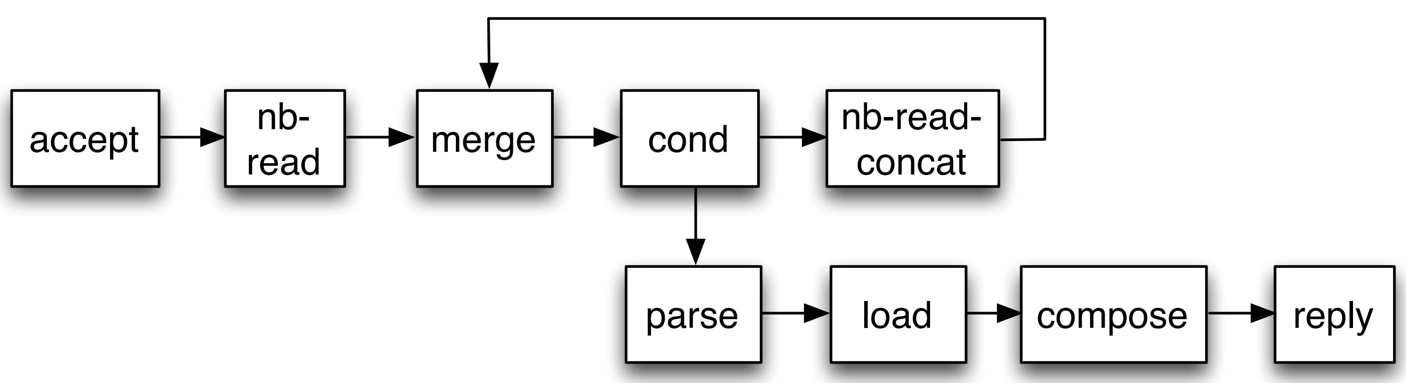
https://bitbucket.org/sertel/ohua

https://bitbucket.org/sertel/ohua-server

Loops

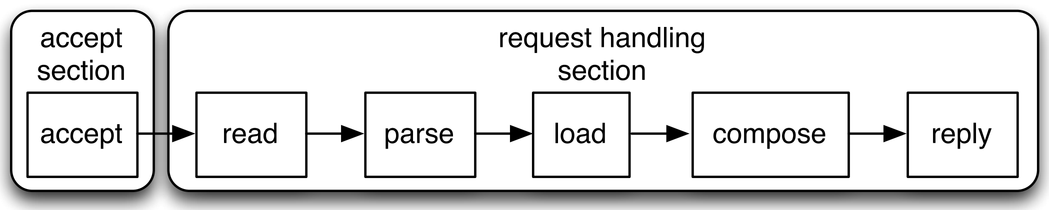
- Example: non-blocking read

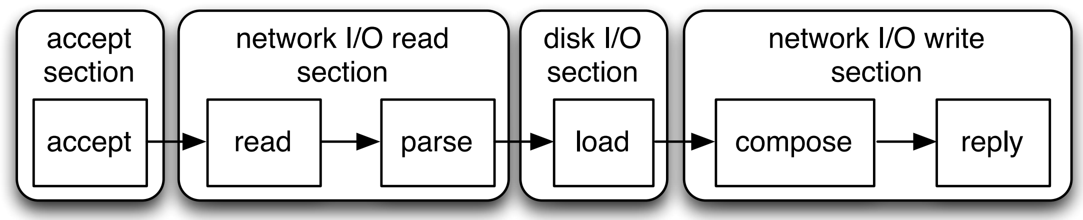
```
(ohua
  (reply (compose (load (parse
                        (loop [[read-data cnn] (nb-read (accept 80))]
                          (if (.endsWith read-data "\n\n") ; stop on blank line
                            [read-data cnn]
                            (recur (nb-read-concat cnn read-data)))))))))
```



- Beware of state inside loops!

Sections



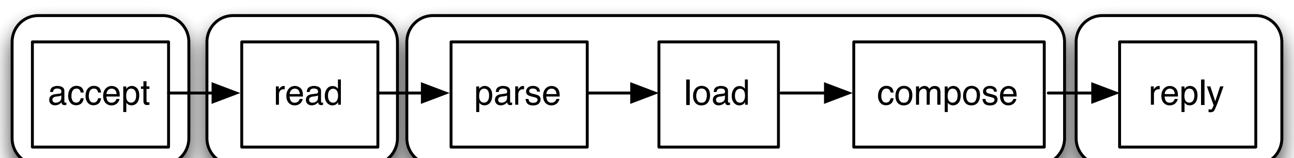


- Manual section configuration via regular expressions:

```
'(["accept.*"] ["read.*" "parse.*" "load.*" "compose.*" "reply.*"])
```

```
'(["accept.*"] ["read.*" "parse.*"] ["load.*"] ["compose*" "reply.*"])
```

- Restrictions:




```
(doto (new java.util.Properties)
  (.put "section-strategy" "com.ohua.LooseConfigurableSectionMapping")
  (.put "section-config" '(["parse.*" "load.*" "compose.*"])))
```

- Ultimate goal: automatic section configuration.

Type sensitive request handling

```
(ohua
  (let [[_ resource-ref] (-> 80 accept read parse)]
    (if (.startsWith resource-ref "news/")
      (-> resource-ref load-ram write reply)
      (-> resource-ref load-hd write reply))))
```

- Scenario: small feeds in RAM vs. articles on disk
- Goal: unblock feeds from articles

Type sensitive request handling

```
(ohua
  (let [[_ resource-ref] (-> 80 accept read parse)]
    (if (.startsWith resource-ref "news/")
      (-> resource-ref load-ram write reply)
      (-> resource-ref load-hd write reply))))
```

- Scenario: small feeds in RAM vs. articles on disk
- Goal: unblock feeds from articles