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

```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();
    }
}
```

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();
    }
}
```

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);
    }
}
```
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);
    }
}
```

- Functions instead of channels!
- Stateful functions $\Rightarrow$ 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);
    }
}
```

• Functions instead of channels!
• Stateful functions $\Rightarrow$ 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 ➔ Linker ➔ Dataflow Graph Extraction ➔ Dataflow Compilation ➔ Algorithm Code Transformation

- Algorithm (Clojure) ➔ Linker ➔ Dataflow Graph Extraction ➔ Dataflow Compilation ➔ Algorithm Code Transformation

- Compile-time Data

- Runtime Code
  - Dataflow Graph
  - Compile-time Data

- Section-Mapping Algorithm
  - Execution Engine
  - Hierarchical Scheduler

- Runtime
Ohua
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.

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:

```plaintext
(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:
Closures and Destructuring

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

```plaintext
(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)))
```

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:

\[
\text{(ohua (let [cnn (accept 80)]] (let [[one two three] (balance-data cnn)] (-> one read parse load compose reply) (-> two read parse load compose reply) (-> three read parse load compose reply))))}
\]
Ohua
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 $\Rightarrow$ parallel.
  • Operators scheduled cooperatively $\Rightarrow$ sequential.
Synchronization

• Sections provide lock-free synchronization!

• Consider parallel request handling by type:
  
  ![Diagram of request handling]

  - Shared external resource (file) \(\Rightarrow\) inconsistency.
Synchronization

• Sections provide lock-free synchronization!

• Consider parallel request handling by type:

  \[
  \text{PUT} \rightarrow \text{GET} \\
  \text{accept read} \rightarrow \text{parse-header cond load compose-get-resp reply} \\
  \text{parse-post store compose-post-resp reply}
  \]

• Shared external resource (file) \(\Leftrightarrow\) inconsistency.

  \[
  \text{SYNC SECTION} \\
  \text{accept read} \rightarrow \text{parse-header cond load compose-get-resp reply} \\
  \text{parse-post store compose-post-resp reply}
  \]
Experimental Setup

server (8-core) → request delay: 20 ms → clients spread across 19 - (8-core) machines
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.

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

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

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

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

- Best deployment strategy: call/sgl-rd-rp/non-blocking/normal
Jetty vs. Ohua

(a) Jetty GC
Jetty vs. Ohua
Jetty vs. Ohua

- 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:
  \[
  \text{program} = \text{declarative algorithms} + \text{stateful functions}
  \]

- 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

```scheme
(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:

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

- Restrictions:

  ```java
  (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

- Scenario: small feeds in RAM vs. articles on disk
- Goal: unblock feeds from articles
Type sensitive request handling

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

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