Current Research

SSSG Seminar
Sven Stork
17th November 2008
Outline

- Motivation
- Petri Nets
- Dataflow Architectures
- My Idea
- Summary
Motivation

• Writing concurrent applications is **hard** and **error prone**
  – User needs to deal with concurrency at low level abstraction (e.g. threads, locks, ...)
  – Current programming languages were designed
  – User needs to consider all possible executions paths/combinations

• **Question**
  Is there an easier way to write a concurrent programs without dealing of such drawbacks?
Outline

• Motivation
• Petri Nets
• Dataflow Architectures
• My Idea
• Summary
Petri Nets

A Petri net is a directed bipartite graph.

A Petri net consists of:
- Tokens
- Places
- Transitions
Petri Nets

- Non-Determinism
- Trigger multiple actions
- Wait for all previous actions to finish
Outline

- Motivation
- Petri Nets
- Dataflow Architectures
- My Idea
- Summary
Dataflow Architecture

- deterministic Petri Net
- Express data dependencies between statements
  - Program is data dependency graph
- Execution of functions depends on availability of their input data
- At every moment of time all possible execution units are know
  - MAXIMUM parallelism
Dataflow Architecture

Input: $u, v, w$;

$x = u - (v + w)$;

$y = u \times (v + w)$;

Output: $x, y$;
Dataflow

**Pro**
- Expresses all the parallelism in a program
- No shared data
  - Data is consumed and produced
  - No synchronisation required

**Contra**
- Hard to write programs
- Inefficient
  - Always creating and consuming data is expensive
Outline

- Motivation
- Petri Nets
- Dataflow Architectures
- My Idea
- Summary
My Idea

```java
void readItems(Queue q) { ... }

void updateItems(Queue q,
                Deps d,
                Stats s) { ... }

void removeDuplicates(Queue q,
                      Deps d){ ... }

Deps computeDependencies(Queue q) { ... }

Stats computeStatistics(Queue q) { ... }

void main () {
    Queue q = new Queue();

    readItems(q);
    Stats s = computeStatistics(q);
    Deps s  = computeDependencies(q);
    removeDuplicates(q, d);
    updateItems(q, s d);
}
```
My Idea

• Requirements of operations on data:
  • Access
    – Read
      • Function only read the specified object
    – Write
      • Function read+write the specified object
void readItems(@Write Queue q) { ... }

void updateItems(@Write Queue q,  
                 @Read Deps d,  
                 @Read Stats s) { ... }

void removeDuplicates(@Write Queue q,  
                      @Read Deps d){ ... }

Deps computeDependencies(@Read Queue q) { ... }

Stats computeStatistics(@Read Queue q) { ... }

void main () {
    Queue q = new Queue();

    readItems(q);
    Stats s = computeStatistics(q);
    Deps d = computeDependencies(q);
    removeDuplicates(q, d);
    updateItems(q, s, d);
}
My Idea

Write to q

Read to q

Write to q

1. `readItems`
2. `computeDependencies`
3. `computeStatistics`
4. `removeDuplicates`
5. `updateItems`
My Idea

• Ordering
  – Lexical defined
    \[\ldots \text{removeDuplicates}(q, d); \quad \text{?} \quad \text{updateItems}(q, s d); = \ldots \text{removeDuplicates}(q, d);\]
  – By permission splitting/joining
    \[
    \begin{array}{c}
    \text{Write (1)} \\
    \text{Read (0.5)} \\
    \text{Read (0.25)} \\
    \text{Read (0.25)} \\
    \text{Read (0.25)} \\
    \text{Read (0.25)}
    \end{array}
    \]
My Idea

Write to q

readItems

q@Write(1)

computeDependencies
computeStatistics

q@Read(0.5) q@Read(0.5)

d@Write(1) d@Write(1)

removeDuplicates

q@Write(1) q@Write(1)

d@Write(1) q@Write(1) s@Write(1)

updateItems

s@Write(1)
void producer(@Write Queue q) {
    while (condition1) {
        ...
    }
}

void consumer(@Write Queue q) {
    while (condition2) {
        ...
    }
}

void main () {
    Queue q = new Queue();
    producer(q);
    consumer(q);
}
My Idea

• **Problem:**
  Write permissions force strict sequentially

• What if the consumer could start to work on the already produced items?
  – Pipelining

• What if the producer never stops?
  – Example: contiguous reads data from a sensor

• Solution ==> Introduce @Shared
  (allow only “protected” access)
void producer(@Shared Queue q) {
    while (condition1) {
        ...
    }
}

void consumer(@Shared Queue q) {
    while (condition2) {
        ...
    }
}

void main () {
    Queue q = new Queue();
    producer(q);
    consumer(q);
}
My Ideas

- **Access-Matrix**

<table>
<thead>
<tr>
<th></th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reading</td>
</tr>
<tr>
<td>Single</td>
<td>ReadOnly (RO)</td>
</tr>
<tr>
<td>Concurrent</td>
<td>ReadOnly (RO)</td>
</tr>
</tbody>
</table>
My Idea

• Granularity
  – Data
    • Collection of Objects
    • Objects
    • Partitions of Objects
    • (Every Field of an Object)
  – Code
    • Methods
    • Blocks
    • Statements
My Idea

• Interesting Questions
  – What to do at runtime/compile time?
  – How to avoid deadlocks?
  – How and when to select granularity?

• Runtime System
  – How to represent dependencies?
  – How to deal with blocking operations?
  – How to efficiently implement synch. tasks?
Summary

- Dataflow Architectures are nice for representing concurrency
- Use Permissions to data to derive dataflow graph from a program
  - Permissions = tokens
- A lot of open questions ...