Dynamic Expressivity with Static Optimization for Streaming Languages

Robert Soulé  Michael I. Gordon  Saman Amarasinghe  Robert Grimm  Martin Hirzel
Cornell  MIT  MIT  NYU  IBM

DEBS 2013
Problem

“Rate” = number of queue pushes/pops per operator firing

Dynamic rate (varies at runtime)  
requires dynamic expressivity

Static rate (known at compile time)  
enables static optimization

How to get both?

Observation: applications are “mostly static”  
(Thies, Amarasinghe [PACT 2010])
StreamIt, a Streaming Language Designed for Static Optimization

float->float pipeline ABC {
    add float->float filter A() {
        work pop … push 2
        { … }
    }
    add float->float filter B() {
        work pop 3 push 1
        { … }
    }
    add float->float filter C() {
        work pop 2 push …
        { … }
    }
}

⇒ Statically known push/pop rates (SDF = Synchronous Dataflow)
SDF Steady-State Schedule

⇒ Statically known firing order and FIFO queue sizes
Scalarization

Implement FIFO queue via local variables, or even registers (more intricate with “peek”, not shown in this talk)
Fission (Data Parallelism)

Roundrobin Split

\( X_1 \) \( X_2 \)

Roundrobin Merge

\( X_1 \) \( X_2 \)

Roundrobin Split

\( X_1 \) \( X_2 \)

Roundrobin Merge

\( X_1 \) \( X_2 \)

Roundrobin Split

\( X_1 \) \( X_2 \)

Roundrobin Merge

\( X_1 \) \( X_2 \)

\( \rightarrow \) Round-robin split and merge rely on static rates
Dynamic Rates

float→float pipeline Decoder {
    add float→float filter VideoInput() {
        work pop 1 push 1
        { ... }
    }
    add float→float filter Huffman() {
        work pop * push 1
        { ... }
    }
    add float→float filter IQuant() {
        work pop 64 push 64
        { ... }
    }
    add float→float filter IDCT() {
        work pop 8 push 8
        { ... }
    }
}

⇒ No more static optimization?
## Dynamic Scheduling Approaches

<table>
<thead>
<tr>
<th>Scheduling approach</th>
<th>Description</th>
<th>Representative citation</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS Thread</td>
<td>Each operator has its own thread</td>
<td>SPC, Amini et al. [DMSSP 2006]</td>
</tr>
<tr>
<td>Demand</td>
<td>Recruit from thread pool</td>
<td>Aurora, Abadi et al. [VLDBJ 2003]</td>
</tr>
<tr>
<td>No-op</td>
<td>Static rate, send nonce when no data</td>
<td>CQL, Arasu et al. [VLDBJ 2006]</td>
</tr>
</tbody>
</table>
Our Approach:
Locally Static + Globally Dynamic

1. Partitioning into static subgraphs
2. Locally optimize static subgraphs
   2a. Fusion
   2b. Scalarization
   2c. Fission
3. Placement
   3a. Core placement
   3b. Thread placement
4. Globally dynamic scheduling
Partition into Static Subgraphs

**Static subgraph:** Weakly connected component after deleting dynamic edges.

*Partitioning*
Locally Optimize Static Subgraphs

Partitioning
Fusion
Scalarization
Fission
Core Placement

Video Input

* Huffman

IQuant

IDCT

IQuant

IDCT

Video Input

Huffman

IQuant

IDCT

IQuant

IDCT

Static weight estimate and greedy bin-packing

Place fission replicas on all cores
Thread Placement

One pinned thread per static subgraph per core
(must be able to suspend dynamic reader when no input)
Dynamic Scheduling

Use condition variables for hand-off to successor

Legend:
- Control
Data Pipelining

Use buffer for pipeline parallelism

Legend:
- Control
- Data
Dynamic vs. Static Performance

Close enough for heavy operators, but what about light operators?
Amortizing the Thread Switching Overhead

1. Partitioning into static subgraphs
2. Locally optimize static subgraphs
   2a. Fusion
   2b. Scalarization
   2c. Fission
   2d. Batching
3. Placement
   3a. Core placement
   3b. Thread placement
4. Globally dynamic scheduling
Benefit of Batching

File Reader
↓
Work 1
↓
* N dynamic queues
↓
Work 1
↓
File Writer

➔ Amortize thread switching overhead without heavy operators
## Our vs. Other Dynamic Schedulers Performance

<table>
<thead>
<tr>
<th>Scheduling approach</th>
<th>Experiment</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS Thread</td>
<td>32 threads, 1 core, work 31 per operator</td>
<td>Our scheduler is 10x faster</td>
</tr>
<tr>
<td>Demand</td>
<td>Huffman encoder and decoder</td>
<td>Our scheduler is 1.2x faster</td>
</tr>
<tr>
<td>No-op</td>
<td>2 programs: VWAP and predicate filter</td>
<td>Our scheduler is 5.1x and 4.9x faster</td>
</tr>
</tbody>
</table>

⇒ Our scheduler was faster in all cases (see paper for details)
Conclusions

• Static streaming languages such as StreamIt enable powerful optimizations
• But many real-world applications require dynamic rates
• We extend the StreamIt optimizing compiler to handle dynamic rates