Self-Adjusting Stack Machines

Matthew A. Hammer
Georg Neis  Yan Chen  Umut A. Acar

Max Planck Institute for Software Systems

OOPSLA 2011 — October 27, 2011
Portland, Oregon, USA
Static Computation Versus Dynamic Computation

Static Computation:

Fixed Input → Compute → Fixed Output

Dynamic Computation:

Changing Input → Compute → Changing Output

Read Changes → Update → Write Updates
Dynamic Data is Everywhere

Software systems often consume/produce dynamic data

Scientific Simulation

Reactive Systems

Analysis of Internet data
Tractability Requires Dynamic Computations

Static Case
(Re-evaluation “from scratch”)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>compute</td>
<td>1 sec</td>
</tr>
<tr>
<td># of changes</td>
<td>1 million</td>
</tr>
<tr>
<td>Total time</td>
<td>11.6 days</td>
</tr>
</tbody>
</table>
Tractability Requires Dynamic Computations

### Static Case
(Re-evaluation “from scratch”)
- **compute**: 1 sec
- **# of changes**: 1 million
- **Total time**: 11.6 days

### Dynamic Case
(Uses update mechanism)
- **compute**: 10 sec
- **update**: $1 \times 10^{-3}$ sec
- **# of changes**: 1 million
- **Total time**: 16.7 minutes
- **Speedup**: 1000x
Dynamic Computations can be Hand-Crafted

As an input sequence changes, maintain a sorted output.

Changing Input

$1,7,3,6,5,2,4$ $\rightarrow$ compute $\rightarrow$ $1,2,3,4,5,6,7$

Remove 6 $\rightarrow$ $1,7,3,6,5,2,4$ $\rightarrow$ update $\rightarrow$ $1,2,3,4,5,6,7$

Reinsert 6, Remove 2 $\rightarrow$ $1,7,3,6,5,2,4$ $\rightarrow$ update $\rightarrow$ $1,2,3,4,5,6,7$

A binary search tree would suffice here (e.g., a splay tree)

What about more exotic/complex computations?
Can this programming be systematic? What are the right abstractions?

1. How to describe dynamic computations?
   - **Usability**: Are these descriptions easy to write?
   - **Generality**: How much can they describe?

2. How to implement these descriptions?
   - **Efficiency**: Are updates faster than re-evaluation?
   - **Consistency**: Do updates provide the correct result?
In **Self-Adjusting Computation**, ordinary programs describe dynamic computations.

The **self-adjusting program**:

1. Computes initial output from initial input
2. Automatically **updates** output when input changes
Self-adjusting program maintains dynamic dependencies in an execution trace.

Key Idea: Reusing traces $\leadsto$ efficient update
Challenges

Existing work targets functional languages:

- Library support for SML and Haskell
- DeltaML extends MLton SML compiler

Our work targets low-level languages (e.g., C)

- stack-based
- imperative
- no strong type system
- no automatic memory management
Challenges Low-Level Self-Adj. Computation

Efficient update ⇝ complex resource interactions:
- execution trace, call stack, memory manager
Challenges Low-Level Self-Adj. Computation

Efficient **update** $\leadsto$ complex **resource interactions**:
- execution trace, call stack, memory manager
Efficient **update** \(\sim\) complex **resource interactions**: execution trace, call stack, memory manager

- Found change
- Change propagation
- Code revaluation
- Found match
- (repair + edit old trace)
- (make new trace, search old trace)
**Objective**: As tree changes, maintain its valuation

\[
((3 + 4) - 0) + (5 - 6) = 6
\]

\[
((3 + 4) - 0) + ((5 - 6) + 5) = 11
\]
Objective: As tree changes, maintain its valuation

\[
((3 + 4) - 0) + (5 - 6) = 6 \\
((3 + 4) - 0) + ((5 - 6) + 5) = 11
\]

Consistency: Output is correct valuation

Efficiency: Update time is \(O(\#\text{affected intermediate results})\)
Expression Tree Evaluation in C

```c
typedef struct node_s* node_t;

struct node_s {
    enum { LEAF, BINOP } tag;
    union {
        int leaf;
        struct {
            enum { PLUS, MINUS } op;
            node_t left, right;
        } binop;
    } u;
}

int eval (node_t root) {
    if (root->tag == LEAF)
        return root->u.leaf;
    else {
        int l = eval (root->u.binop.left);
        int r = eval (root->u.binop.right);
        if (root->u.binop.op == PLUS) return (l + r);
        else return (l - r);
    }
}
```
The Stack “Shapes” the Computation

```c
int eval (node_t root) {
    if (root->tag == LEAF)
        return root->u.leaf;
    else {
        int l = eval (root->u.binop.left);
        int r = eval (root->u.binop.right);
        if (root->u.binop.op == PLUS) return (l + r);
        else return (l - r);
    }
}
```

Stack usage breaks computation into three parts:

- **Part A**: Return value if LEAF
- **Part B**: Evaluate the right child
- **Part C**: Apply BINOP to intermediate results; return
int eval (node_t root) {
    if (root->tag == LEAF)
        return root->u.leaf;
    else {
        int l = eval (root->u.binop.left);
        int r = eval (root->u.binop.right);
        if (root->u.binop.op == PLUS) return (l + r);
        else return (l - r);
    }
}

Stack usage breaks computation into three parts:

- **Part A**: Return value if LEAF
  Otherwise, evaluate BINOP, starting with left child
int eval (node_t root) {
    if (root->tag == LEAF)
        return root->u.leaf;
    else {
        int l = eval (root->u.binop.left);
        int r = eval (root->u.binop.right);
        if (root->u.binop.op == PLUS) return (l + r);
        else return (l - r);
    }
}

Stack usage breaks computation into three parts:

- **Part A**: Return value if LEAF
  Otherwise, evaluate BINOP, starting with left child
- **Part B**: Evaluate the right child
The Stack “Shapes” the Computation

```c
int eval (node_t root) {
    if (root->tag == LEAF)
        return root->u.leaf;
    else {
        int l = eval (root->u.binop.left);
        int r = eval (root->u.binop.right);
        if (root->u.binop.op == PLUS) return (l + r);
        else return (l - r);
    }
}
```

Stack usage breaks computation into three parts:

- **Part A**: Return value if LEAF
  Otherwise, evaluate BINOP, starting with left child
- **Part B**: Evaluate the right child
- **Part C**: Apply BINOP to intermediate results; return
Dynamic Execution Traces

Input Tree

Execution Trace

Matthew A. Hammer
Self-Adjusting Stack Machines 14
How to Update the Output?

Original Input

Changed Input

Goals:

- **Consistency**: Respect the (static) program’s meaning
- **Efficiency**: Reuse original computation when possible
How to Update the Output?

**Original Input**

```
+   
|   +
|---+   -
|   |   |
+   -
+   |
+   +
+   |
3   |
\   |
\   -
\   +
\   |
3 4 5 6
```

**Changed Input**

```
+   
|   +
|---+   -
|   |   |
+   -
+   |
+   +
+   |
3   |
\   |
\   -
\   +
\   |
3 4 5 6
```

**Goals:**

- **Consistency**: Respect the (static) program’s meaning
- **Efficiency**: Reuse original computation when possible

**Idea**: Transform the first trace into second trace
Unaffected/Reuse

Affected/Re-eval

Aged/Re-eval

New Evaluation

A

B

C

A+

A-

B+

C+

A0

A1

A2

A3

A4

A5

A6

B+

B-

C-

B0

B1

B2
Before Update

A_+  →  A_  →  A_+  →  A_3  →  A_+  →  B_+  →  B_  →  B_+  →  B_  →  C_+  →  C_  →  C_+  →  C_

After Update

A_+  →  A_  →  A_+  →  A_3  →  A_+  →  B_+  →  B_  →  B_+  →  B_  →  C_+  →  C_  →  C_+  →  C_
1. How to describe dynamic computations?
   ✓ **Usability**: Are these descriptions easy to write?
   ✓ **Generality**: How much can they describe?

2. How to implement this description?
   ? **Correctness**: Do updates provide the correct result?
   ? **Efficiency**: Are updates faster than re-evaluation?
Overview of Formal Semantics

- IL: Intermediate language for C-like programs
- IL has instructions for:
  - Mutable memory: `alloc`, `read`, `write`
  - Managing local state via a stack: `push`, `pop`
  - Saving/restoring local state: `memo`, `update`
Overview of Formal Semantics

- IL: Intermediate language for C-like programs
- IL has instructions for:
  - Mutable memory: `alloc`, `read`, `write`
  - Managing local state via a stack: `push`, `pop`
  - Saving/restoring local state: `memo`, `update`
- Transition semantics: two abstract stack machines:
  - **Reference machine**: defines “normal” semantics
  - **Tracing machine**: defines self-adjusting semantics
    - Can **compute** an output and a trace
    - Can **update** output/trace when memory changes
    - Automatically marks garbage in memory
- We prove that these **stack machines** are consistent
Consistency theorem, Part 1: No Reuse

Tracing machine is consistent with reference machine (when tracing machine runs “from-scratch”, with no reuse)
Tracing machine is consistent with from-scratch runs (When it reuses some existing trace Trace$_0$)
Consistency theorem: Main result

Main result uses Part 1 and Part 2 together:

Tracing machine is consistent with reference machine
How to Program Dynamic Computations?

1. How to describe dynamic computations?
   ✓ **Usability**: Are these descriptions easy to write?
   ✓ **Generality**: How much can they describe?

2. How to implement this description?
   ✓ **Correctness**: Do updates provide the correct result?
   ? **Efficiency**: Are updates faster than re-evaluation?
Overview of Our Implementation

- **Compiler**: produces C targets from C-like source code
- **Run-time**: maintains traces, performs **efficient updates**

Diagram:

CEAL Compiler

- Translate
- Transform
- Translate

C ➔ IL ➔ IL ➔ C + RT
Dynamic Expression Trees: From-Scratch Time

Exptrees From-Scratch

Time (s)

Input Size

Self-Adj

Static

Matthew A. Hammer
Self-Adjusting Stack Machines
Dynamic Expression Trees: Speed up

Exptrees Speedup

Speedup

Self-Adj

Input Size

0 250K 500K 750K

0.0 x 10^0

1.0 x 10^4

1.5 x 10^4

2.0 x 10^4

2.5 x 10^4

Matthew A. Hammer

Self-Adjusting Stack Machines
## Summary of Empirical Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>N</th>
<th>Initial Overhead (Compute / Static)</th>
<th>Speed-up (Static / Update)</th>
</tr>
</thead>
<tbody>
<tr>
<td>exptrees</td>
<td>$10^6$</td>
<td>8.5</td>
<td>$1.4 \times 10^4$</td>
</tr>
<tr>
<td>map</td>
<td>$10^6$</td>
<td>18.4</td>
<td>$3.0 \times 10^4$</td>
</tr>
<tr>
<td>reverse</td>
<td>$10^6$</td>
<td>18.4</td>
<td>$3.8 \times 10^4$</td>
</tr>
<tr>
<td>filter</td>
<td>$10^6$</td>
<td>10.7</td>
<td>$4.9 \times 10^4$</td>
</tr>
<tr>
<td>sum</td>
<td>$10^6$</td>
<td>9.6</td>
<td>$1.5 \times 10^3$</td>
</tr>
<tr>
<td>minimum</td>
<td>$10^6$</td>
<td>7.7</td>
<td>$1.4 \times 10^4$</td>
</tr>
<tr>
<td>quicksort</td>
<td>$10^5$</td>
<td>8.2</td>
<td>$6.9 \times 10^2$</td>
</tr>
<tr>
<td>mergesort</td>
<td>$10^5$</td>
<td>7.2</td>
<td>$7.8 \times 10^2$</td>
</tr>
<tr>
<td>quickhull</td>
<td>$10^5$</td>
<td>3.7</td>
<td>$2.2 \times 10^3$</td>
</tr>
<tr>
<td>diameter</td>
<td>$10^5$</td>
<td>3.4</td>
<td>$1.8 \times 10^3$</td>
</tr>
<tr>
<td>distance</td>
<td>$10^5$</td>
<td>3.4</td>
<td>$7.9 \times 10^2$</td>
</tr>
</tbody>
</table>
Our Contributions

A consistent self-adjusting semantics for low-level programs
Our Contributions

A consistent **self-adjusting semantics** for low-level programs

Our **abstract machine semantics**

- Describes **trace editing & memory management**
  - Implementation of run-time system
- But requires programs with a **particular structure**
  - Implementation of compiler transformations
Our Contributions

A consistent **self-adjusting semantics** for low-level programs

**Our abstract machine semantics**
- Describes **trace editing & memory management**
  - Implementation of run-time system
- But requires programs with a **particular structure**
  - Implementation of compiler transformations

**Our intermediate language** is low-level, yet abstract
- orthogonal annotations for self-adjusting behavior
- no type system needed
  - Implementation of C front end
Self-adjusting computation is a language-based technique to derive dynamic programs from static programs.

Summary of contributions:
- A self-adjusting semantics for low-level programs. This semantics defines self-adjusting stack machines.
- A compiler and run-time that implement the semantics.
- A front end that embeds much of C.