Self-Adjusting Machines

Matthew A. Hammer

University of Chicago
Max Planck Institute for Software Systems

Thesis Defense
July 20, 2012
Chicago, IL
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”)

<table>
<thead>
<tr>
<th>compute</th>
<th>1 sec</th>
</tr>
</thead>
<tbody>
<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 → compute → 1,2,3,4,5,6,7

Changing Output

Remove 6

1,7,3,6,5,2,4 → update → 1,2,3,4,5,7

Reinsert 6, Remove 2

1,7,3,6,5,2,4 → update → 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?
Self-Adjusting Computation

Offers a systematic way to program **dynamic computations**

\[ \text{Domain knowledge} \quad + \quad \text{Library primitives} \]

\[ \text{Self-Adjusting Program} \]

The **library primitives**:

1. **Compute** initial output and trace from initial input
2. **Change propagation** updates output and trace
High-level versus low-level languages

Existing work uses/targets **high-level languages** (e.g., SML)

In **low-level languages** (e.g., C), there are **new challenges**

<table>
<thead>
<tr>
<th>Language feature</th>
<th>High-level help</th>
<th>Low-level gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type system</td>
<td>Indicates mutability</td>
<td>Everything mutable</td>
</tr>
<tr>
<td>Functions</td>
<td>Higher-order traces</td>
<td>Closures are manual</td>
</tr>
<tr>
<td>Stack space</td>
<td>Alters stack profile</td>
<td>Bounded stack space</td>
</tr>
<tr>
<td>Heap management</td>
<td>Automatic GC</td>
<td>Explicit management</td>
</tr>
</tbody>
</table>

C is based on a low-level **machine model**

This model lacks **self-adjusting primitives**
Thesis statement
By making their resources explicit, **self-adjusting machines** give an operational account of **self-adjusting computation** suitable for interoperation with **low-level languages**;

via practical **compilation** and **run-time techniques**, these machines are **programmable, sound and efficient**.

Contributions

<table>
<thead>
<tr>
<th>Surface language, C-based</th>
<th>Programmable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract machine model</td>
<td>Sound</td>
</tr>
<tr>
<td>Compiler</td>
<td>Realizes static aspects</td>
</tr>
<tr>
<td>Run-time library</td>
<td>Realizes dynamic aspects</td>
</tr>
<tr>
<td>Empirical evaluation</td>
<td>Efficient</td>
</tr>
</tbody>
</table>
Example: Dynamic Expression Trees

**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 \(O(\#\text{affected intermediate results})\)
```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
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
```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
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
Updating inputs, traces and outputs

A_+ → B_+ → C_+ → A_+ → B_+ → C_+ → A_+ → B_+ → C_+ → A_0 → B_+ → C_+ → A_5 → B_+ → C_+ → A_5
Core self-adjusting primitives

Stack operations: push & pop
Trace checkpoints: memo & update points

A\_ + B\_ + C\_ + A\_ - B\_ - C\_ - A\_ + B\_ + C\_ + A\_ + B\_ + C\_ + A\_ - B\_ - C\_ - A\_ 5

A\_ 3 A\_ 4

A\_ + B\_ + C\_ + A\_ 0

A\_ 3 A\_ 4

B\_ + C\_ + B\_ + C\_ + B\_ + C\_ + B\_ + C\_

A\_ + B\_ + C\_ + A\_ 5 A\_ 6

A\_ + B\_ + C\_ + A\_ 5 A\_ 6

Update

Update

(new evaluation)
Abstract model:
Self-adjusting machines
Overview of abstract machines

- **IL: Intermediate language**
  - Uses static-single assignment representation
  - Distinguishes *local* from *non-local* mutation
- **Core IL constructs:**
  - *Stack operations*: push, pop
  - *Trace checkpoints*: memo, update
- **Additional IL constructs:**
  - *Modifiable memory*: alloc, read, write
  - (Other extensions possible)
Two abstract machines given by small-step transition semantics:

- **Reference machine**: defines normal semantics
- **Self-adjusting 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 abstract machines are consistent
i.e., updated output is always consistent with normal semantics
An IL program is **store agnostic** when each stack frame has a fixed return value; hence, not affected by **update** points.

**destination-passing style** (DPS) transformation:
- Assigns a **destination** in memory for each stack frame
- Return values are these destinations
- Converts stack dependencies into memory dependencies
- **memo** and **update** points **reuse** and **update** destinations

- Lemma: DPS-conversion preserves program meaning
- Lemma: DPS-conversion achieves store agnosticism
Consistency theorem, Part 1: No Reuse

Self-adjusting machine is consistent with reference machine when self-adjusting machine runs “from-scratch”, with no reuse
Self-adjusting machine is consistent with from-scratch runs
When it reuses some existing trace $\text{Trace}_0$
Consistency theorem: Main result

Main result uses Part 1 and Part 2 together:

Self-adjusting machine is consistent with reference machine
Concrete
Self-adjusting machines
Overview of design and implementation

- Abstract model guides design
- Compiler addresses static aspects
- Run-time (RT) addresses dynamic aspects

Phases

- Front-end translates CEAL surface language into IL
- Compiler analyses and transforms IL
- Compiler produces C target code, links with RT library
- Optional optimizations cross-cut compiler and RT library
Compiler transformations

**Destination-passing style (DPS) conversion**
- Required by our abstract model
- Converts stack dependencies into memory dependencies
- Inserts additional *memo* and *update* points

**Normalization**
- Required by C programming model
- Lifts *update* points into top-level functions
- Exposes those code blocks for reevaluation by RT
Compiler analyses

- guide necessary transformations
- guide optional optimizations

Special uses

- memo/update analysis: selective DPS conversion
- live variable analysis: translation of memo/update points
- dominator analysis: normalization, spatial layout of trace
From compiler to run-time system

**Trace nodes**
- Indivisible block of traced operations
- Operations share overhead (e.g., closure information)
- Compiler produces **trace node descriptors** in target code

**Run-time system**
- RT interace based on trace node descriptors (from compiler)
  - redo callback — code at update points
  - undo callback — revert traced operations
- Change propagation incorporates **garbage collection**
Optimizations

**Spars**er traces — *avoid tracing when possible*

1. **Stable references**  
   Programmer uses type qualifier

2. **Selective DPS**  
   Compiler analysis of update points

**Cheaper traces — more efficient representation**

3. **Write-once memory**  
   Programmer uses type qualifier

4. **Trace node sharing**  
   Compiler analysis coalesces traced ops
Evaluation
From-scratch time: Constant overhead

Exptrees From-Scratch

Time (s)

Input Size

Self-Adj

Static
Average update time: Constant time

Exptrees Ave Update

Time (ms)

Input Size

Self-Adj
Speed up = From-scratch / Update

Exptrees Speedup

Self-Adj

Input Size

Matthew A. Hammer
Self-Adjusting Machines
Evolution of our approach

Stage 1: **First run-time library**
- Change propagation & memory management
  - Very high programmer burden

Stage 2: **First compiler**
- Lower programmer burden
  - No return values
  - Memo points are non-orthogonal
    (conflated with read and alloc primitives)
  - No model for consistency or optimizations

Stage 3: **New compiler & run-time library**
- Self-adjusting machine semantics guides reasoning about consistency & optimizations
- Very low programmer burden
**Stage 1, RT library: vs SML library**

- **SML-GC** is comparable to **C**
- **SML+GC** are 10x slower
## Normalized Measurements \([(\text{CEAL} / \text{DeltaML}) \times 100]\)

<table>
<thead>
<tr>
<th>App.</th>
<th>From-Scratch</th>
<th>Ave. Update</th>
<th>Max Live</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>11%</td>
<td>16%</td>
<td>23%</td>
</tr>
<tr>
<td>map</td>
<td>11%</td>
<td>14%</td>
<td>23%</td>
</tr>
<tr>
<td>reverse</td>
<td>13%</td>
<td>17%</td>
<td>24%</td>
</tr>
<tr>
<td>minimum</td>
<td>22%</td>
<td>11%</td>
<td>38%</td>
</tr>
<tr>
<td>sum</td>
<td>22%</td>
<td>29%</td>
<td>34%</td>
</tr>
<tr>
<td>quicksort</td>
<td>4%</td>
<td>6%</td>
<td>21%</td>
</tr>
<tr>
<td>quickhull</td>
<td>20%</td>
<td>30%</td>
<td>91%</td>
</tr>
<tr>
<td>diameter</td>
<td>17%</td>
<td>23%</td>
<td>67%</td>
</tr>
<tr>
<td><strong>Averages</strong></td>
<td><strong>15%</strong></td>
<td><strong>18%</strong></td>
<td><strong>40%</strong></td>
</tr>
<tr>
<td>Stage 3, Machine model: Multiple targets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. <strong>Stable references</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programmer uses type qualifier</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. <strong>Selective DPS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler analysis of update points</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. <strong>Write-once memory</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programmer uses type qualifier</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. <strong>Trace node sharing</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler analysis coalesces traced ops</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Stage 3, Machine model: Average update times

Update Time (norm. by no-opt)

- exprees
- map
- reverse
- filter
- sum
- minimum
- quicksort
- mergesort
- quickhull
- diameter
- distance
- mean

Legend:
- all-opt
- no-seldps
- no-share
- no-stable
- no-owcr
Stage 3, Machine model: Maximum live space

Max Live Space (norm by no-opt)

- exptrees
- map
- reverse
- filter
- sum
- minimum
- quicksort
- mergesort
- quickhull
- diameter
- distance
- mean

Categories:
- all-opt
- no-seldps
- no-share
- no-stable
- no-owcr
Stage 3, Machine model: Previous approaches

- **Delta-ML**: order of magnitude slower
- **CEAL** (stage 2) slightly faster than **all-opt** (stage 3)
  - CEAL uses non-orthogonal allocation primitive
Thesis statement

By making their resources explicit, **self-adjusting machines** give an operational account of **self-adjusting computation** suitable for interoperation with **low-level languages**;

via practical **compilation** and **run-time techniques**, these machines are **programmable**, **sound** and **efficient**.

Contributions

- Surface language, C-based
- Abstract machine model
- Compiler
- Run-time library
- Empirical evaluation

- **Programmable**
- **Sound**
- Realizes static aspects
- Realizes dynamic aspects
- **Efficient**