CEAL: A C-based Language for Self-Adjusting Computation

Matthew Hammer  Umut Acar  Yan Chen

Toyota Technological Institute at Chicago

PLDI 2009
Programs usually run from-scratch on new inputs
Programs usually run \textit{from-scratch} on new inputs

Interested when small input change $\Rightarrow$ small output change
Include program trace $T$ with program output.

Idea: Small $\epsilon$ and small $\delta$ often implies small $\Delta$. 

Matthew Hammer

CEAL: A C-Based Language for Self-Adj. Computation
Self-adjusting computation

- **Initial run** records a program trace
- **Change propagation** \((cp)\) updates the output & trace
Goal of self-adjusting computation

Given input change ($\epsilon$), update output & trace in time proportional to trace distance ($\Delta$)

<table>
<thead>
<tr>
<th>Application</th>
<th>From-scratch</th>
<th>Insertion/deletion</th>
</tr>
</thead>
<tbody>
<tr>
<td>List Primitives</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \cdot \log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$O(n \cdot \log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Convex Hulls (2d)</td>
<td>$O(n \cdot \log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Convex Hulls (3d)</td>
<td>$O(n \cdot \log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

From-scratch time vs trace distance for an insertion/deletion
Linear vs logarithmic time

- Interesting problems take $O(n)$ time (or more)
- Suppose we can update output in $O(\log n)$ time
- Is the speedup worth it?

$O(n)$ increases exponentially faster than $O(\log n)$
Challenges for Self-Adjusting Computation

- Challenge 1
  - Impractical to trace every operation
  - What operations should be traced?

- Challenge 2
  - Trace must support efficient incremental updates
  - How should the trace be structured?
What operations should be traced?

Idea: Distinguish between **stable** and **changeable** data

- Programmer manages changeable data in **modrefs** (modifiable references)
- Analogous to conventional references
- Trace records modref operations

<table>
<thead>
<tr>
<th>Modref operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>modref_t* modref()</code></td>
<td>Create an empty modref</td>
</tr>
<tr>
<td><code>void write(modref_t *m, void *p)</code></td>
<td>Write to a modref</td>
</tr>
<tr>
<td><code>void* read(modref_t *m)</code></td>
<td>Read from a modref</td>
</tr>
</tbody>
</table>
Example: Evaluating expression trees

cenal eval (modref_t *in, modref_t *out) {
    node_t *node = read (in);
    if (node->kind == LEAF)
        write (out, node->leaf_value);
    else {
        modref_t *m_a = modref ();
        modref_t *m_b = modref ();
        eval (node->left_child, m_a);
        eval (node->right_child, m_b);
        int a = read (m_a);
        int b = read (m_b);

        if (node->binary_op == PLUS) {
            write (out, a + b);
        } else {
            write (out, a - b);
        }
    }
}

Key Idea
Input & output stored in modrefs

TODO: illustrate execution of the two cases
How is trace structured? how do we update it?

Need to identify & record dependencies between data & code

Idea: When a modref is changed, rerun code with new value
How is trace structured? how do we update it?

Need to identify & record dependencies between data & code

**Idea**: When a modref is changed, rerun code with new value

**What code do we rerun?**
How is trace structured? how do we update it?

Need to identify & record dependencies between data & code

Idea: When a modref is changed, rerun code with new value

What code do we rerun?

Normal form

Every read followed by a tail call

\[ x = \text{read}(m); \text{tail } f(x, y) \]

- Use of \( m \)’s value recorded as a closure of \( f \)
- Closure can be rerun if & when \( m \) changes
Tracing return values

TODO: make this slide more concise

▶ How to trace & rerun functions with return values?
▶ How do we rerun the caller without recording call stack?

Idea: Return results through modref arguments

▶ Don’t want to record call stack
▶ Restrict all functions with reads to return void
▶ Destination-passing style returns results in modrefs
▶ modrefs track all callee-to-caller dataflow
**Goal**: Compile CEAL into C code. Target C code is linked with a runtime library.

CEAL to C: a two step process

- Normalize CEAL code (put into **normal form**)
- **Translate** the (normal form) CEAL code to C
Normalization
Normalization via control-flow graphs

Idea: Transform the program as a control-flow graph

Nodes
- a designated root node
- a function node
- a command (read, write, etc.), conditional, or return statement

Entry Nodes
- a function entry $\equiv$ a function node
- a read entry $\equiv$ successor of a read

Edges
- Control edges: (tail) call & goto
- Entry edges: from root to entry nodes
Example: Program graph for eval

```c
ceal eval (modref_t *in, modref_t *out) {
    node_t *node = read (in);
    if (node->kind == LEAF) {
        write (out, node->leaf_value);
    } else {
        modref_t *m_a = modref ();
        modref_t *m_b = modref ();
        eval (node->left_child, m_a);
        eval (node->right_child, m_b);
        int a = read (m_a);
        int b = read (m_b);
        if (node->binary_op == PLUS) {
            write (out, a + b);
        } else {
            write (out, a - b);
        }
    }
    return; }
```
ceal eval (modref_t *in, modref_t *out) {
    node_t *node = read (in);
    if (node->kind == LEAF) {
        write (out, node->leaf_value);
    } else {
        modref_t *m_a = modref ();
        modref_t *m_b = modref ();
        eval (node->left_child, m_a);
        eval (node->right_child, m_b);
        int a = read (m_a);
        int b = read (m_b);
        if (node->binary_op == PLUS) {
            write (out, a + b);
        } else {
            write (out, a - b);
        }
    }
    return; }

**Def: Dominator relation**

Node $a$ **dominates** $b$ if every path from root to $b$ contains $a$.

**Def: Immediate dominator relation**

Node $a$ is the **immediate dominator** of $b$ if

- $a \neq b$
- $a$ dominates $b$
- Every other dominator of $b$ dominates $a$

Every node has a (unique) immediate dominator, except root.

**Def: Dominator tree**

Immediate dominator relation forms a tree (root is root node).
Dominators & critical nodes

Dominator examples

- Root \textbf{r} dominates all nodes
- 1 dominates 2, but not 3
- 3 dominates 4 & 6–10, but not 11
- 12 dominates 13 & 15, but not 18

Root \textbf{r} is \textbf{immediate dominator} of

- Every entry node
- Nodes not dominated by any entry (18)

Define: Critical nodes

Nodes immediately dominated by the root
Define critical nodes as root’s children:
Nodes 1, 2, 11, 12 & 18.
Define **critical nodes** as root’s children:

Nodes 1, 2, 11, 12 & 18.

Define **units** as subtrees of critical nodes

Lemma: every cross-unit edge targets a critical node.

Corollary: If each unit becomes a separate function, then cross-unit edges can become calls.
Define **critical nodes** as root’s children:
Nodes 1, 2, 11, 12 & 18.

Define **units** as subtrees of critical nodes

**Lemma:** every cross-unit edge targets a critical node.

**Corollary:** If each unit becomes a separate function, then cross-unit edges can become calls.
Normalization: The Algorithm

Main Ideas:
- Units $\leadsto$ separate functions
- Cross-unit edges $\leadsto$ tail calls (args $\equiv$ live vars)

Algorithm
1. Compute the dominator tree
2. For each critical node, not yet a function node:
   - Create a new function node for unit
   - Redirect incoming critical edges to new function node
     
     (not always necessary; omitting minor details)
Example: New functions, Redirected edges

Before redirection

After redirection

Node 1 already a function node, so no new function needed
ceal eval (modref_t *in, modref_t *out) {
    node_t *node = read (in); tail eval_a (node, out);
}

a ceal eval_a (node_t *node, modref_t *out) {
    if (node->kind == LEAF) {
        write (out, node->leaf_value); tail eval_d ();
    } else {
        ...
        int a = read (m_a); tail eval_b (out, a, m_b);
    }
}

b ceal eval_b (modref_t *out, int a, modref_t *m_b) {
    int b = read (m_b); tail eval_c (out, a, b);
}

c ceal eval_c (modref_t *out, int a, int b) {
    if (node->binary_op == PLUS) {
        write (out, a + b); tail eval_d ();
    } else {
        write (out, a - b); tail eval_d ();
    }
}

d ceal eval_d () {
    return;
}
Translation
Translation Overview

Translation Basics

- Translation introduces closures for tail calls
- For reads: associates closure with read modref
- Uses a trampoline to run closures iteratively

Selective trampolining

- Only need to record closures for reads
- So, only trampoline tail calls that follow reads
- Other tail calls treated like ordinary calls
- Stack grows only temporarily (until a read)

See paper for details
Translation Overview

Translation Basics

- Translation introduces closures for tail calls
- For reads: associates closure with read modref
- Uses a trampoline to run closures iteratively

Selective trampolining

- Only need to record closures for reads
- So, only trampoline tail calls that follow reads
- Other “tail calls” treated like ordinary calls
- Stack grows only temporarily (until a read)

Matthew Hammer
CEAL: A C-Based Language for Self-Adj. Computation
Translation Overview

Translation Basics

- Translation introduces closures for tail calls
- For reads: associates closure with read modref
- Uses a trampoline to run closures iteratively

Selective trampolining

- Only need to record closures for reads
- So, only trampoline tail calls that follow reads
- Other “tail calls” treated like ordinary calls
- Stack grows only temporarily (until a read)

See paper for details
Performance Evaluation
Evaluation Example: Quicksort

Overhead: about 6x (a constant)
Speedup: $1.4 \times 10^4$ (increases linearly with input size)
## Results summary

<table>
<thead>
<tr>
<th>Application</th>
<th>n</th>
<th>Cnv.</th>
<th>Self.</th>
<th>O.H.</th>
<th>Ave. Update</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>10.0M</td>
<td>0.5</td>
<td>7.4</td>
<td>14.2</td>
<td>2.1 × 10⁻⁶</td>
<td>2.4 × 10⁵</td>
</tr>
<tr>
<td>map</td>
<td>10.0M</td>
<td>0.7</td>
<td>11.9</td>
<td>17.2</td>
<td>1.6 × 10⁻⁶</td>
<td>4.2 × 10⁵</td>
</tr>
<tr>
<td>reverse</td>
<td>10.0M</td>
<td>0.6</td>
<td>11.9</td>
<td>18.8</td>
<td>1.6 × 10⁻⁶</td>
<td>3.9 × 10⁵</td>
</tr>
<tr>
<td>minimum</td>
<td>10.0M</td>
<td>0.8</td>
<td>10.9</td>
<td>13.8</td>
<td>4.8 × 10⁻⁶</td>
<td>1.6 × 10⁵</td>
</tr>
<tr>
<td>sum</td>
<td>10.0M</td>
<td>0.8</td>
<td>10.9</td>
<td>13.9</td>
<td>7.0 × 10⁻⁵</td>
<td>1.1 × 10⁴</td>
</tr>
<tr>
<td>quicksort</td>
<td>1.0M</td>
<td>3.5</td>
<td>22.4</td>
<td>6.4</td>
<td>2.4 × 10⁻⁴</td>
<td>1.4 × 10⁴</td>
</tr>
<tr>
<td>quickhull</td>
<td>1.0M</td>
<td>1.1</td>
<td>12.3</td>
<td>11.5</td>
<td>2.3 × 10⁻⁴</td>
<td>4.6 × 10³</td>
</tr>
<tr>
<td>diameter</td>
<td>1.0M</td>
<td>1.0</td>
<td>12.1</td>
<td>12.0</td>
<td>1.2 × 10⁻⁴</td>
<td>8.3 × 10³</td>
</tr>
<tr>
<td>exptrees</td>
<td>10.0M</td>
<td>1.0</td>
<td>7.2</td>
<td>7.2</td>
<td>1.4 × 10⁻⁶</td>
<td>7.1 × 10⁵</td>
</tr>
<tr>
<td>mergesort</td>
<td>1.0M</td>
<td>6.1</td>
<td>37.6</td>
<td>6.1</td>
<td>1.2 × 10⁻⁴</td>
<td>5.1 × 10⁴</td>
</tr>
<tr>
<td>distance</td>
<td>1.0M</td>
<td>1.0</td>
<td>11.0</td>
<td>11.0</td>
<td>1.3 × 10⁻³</td>
<td>7.5 × 10²</td>
</tr>
<tr>
<td>tcon</td>
<td>1.0M</td>
<td>2.6</td>
<td>20.6</td>
<td>7.9</td>
<td>1.0 × 10⁻⁴</td>
<td>2.5 × 10⁴</td>
</tr>
</tbody>
</table>

**Average Overhead**  6–19x  
**Average Speedups**  3.6 × 10⁴ (for $n = 1M$)  
                       1.4 × 10⁵ (for $n = 10M$)  

Matthew Hammer  
CEAL: A C-Based Language for Self-Adj. Computation
**Related Work**

Other self-adjusting/incremental language support:

<table>
<thead>
<tr>
<th>Authors</th>
<th>Conference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acar et. al., PLDI’05</td>
<td>SAC library for ML (Sting)</td>
<td></td>
</tr>
<tr>
<td>Shankar &amp; Bodík, PLDI’07</td>
<td>Invariant checks in Java (Ditto)</td>
<td></td>
</tr>
<tr>
<td>Hammer &amp; Acar, ISMM’08</td>
<td>SAC library for C</td>
<td></td>
</tr>
<tr>
<td>Ley-Wild et. al., ICFP’08</td>
<td>DeltaML language &amp; compiler</td>
<td></td>
</tr>
</tbody>
</table>

**DeltaML** is most comparable system
- Compiler support for general-purpose SAC
- Similar modref-like primitives
- Similar benchmarks
## CEAL vs DeltaML: Summary

### Normalized Measurements (DeltaML / CEAL)

<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>9.3</td>
<td>6.2</td>
<td>4.4</td>
</tr>
<tr>
<td>map</td>
<td>9.3</td>
<td>7.1</td>
<td>4.4</td>
</tr>
<tr>
<td>reverse</td>
<td>8.0</td>
<td>5.8</td>
<td>4.2</td>
</tr>
<tr>
<td>minimum</td>
<td>4.6</td>
<td>8.8</td>
<td>2.9</td>
</tr>
<tr>
<td>sum</td>
<td>4.6</td>
<td>3.5</td>
<td>2.9</td>
</tr>
<tr>
<td>quicksort</td>
<td>26.9</td>
<td>15.6</td>
<td>4.8</td>
</tr>
<tr>
<td>quickhull</td>
<td>5.1</td>
<td>3.3</td>
<td>1.1</td>
</tr>
<tr>
<td>diameter</td>
<td>5.8</td>
<td>4.3</td>
<td>1.5</td>
</tr>
</tbody>
</table>

- **From-Scratch**: CEAL 5–27 times faster (9 on average)
- **Change propagation**: CEAL 3–9 times faster (7 on average)
- **Max live**: CEAL uses up to 5 times less space (3 on average)
Concluding remarks

CEAL: In Summary

- C-based language for self-adjusting computation
- Compiles directly to (portable) C code
- Promising performance results

On-going & future directions

- Support for return values
- Implicit modifiable operations (using type annotations)
- Finer-grained code dependencies for reads
  (At what point can re-execution stop?)
Thank You!
Questions?