Memory Management for Self-Adjusting Computation

Matthew Hammer  Umut Acar

Toyota Technological Institute at Chicago

International Symposium on Memory Management, 2008
In this talk, we

- Briefly review self-adjusting computation
- Discuss memory management issues
- Introduce and evaluate our approach
- Compare to previous SML framework

Previous frameworks written in SML
We implement a framework for C
Motivation: Incremental change is pervasive.

Many applications encounter data that changes slowly or incrementally over time.

- Applications that interact with a physical environment. 
  *E.g., Robots.*

- Applications that interact with a user. 
  *E.g., Games, Editors, Compilers, etc.*

- Application that rely on modeling or simulation. 
  *E.g., Scientific Computing, Computational Biology, Motion Simulation.*
Ordinary Program Runs

- Ordinary programs often run repeatedly on changing input.
- What if input and output change by only small increments?
Ordinary programs often run repeatedly on changing input.

What if input and output change by only small increments?
Self-Adjusting Computation

Ordinary Program Runs

Input 1 → Program → Output 1
small change

Input 2 → Program → Output 2

... ...

Input N → Program → Output N

Self-Adjusting Program Runs

Input 1 → Self-Adj. Program → Output 1
small change

Input 2 → Self-Adj. Program → Output 2
small change

... ...

Input N → Self-Adj. Program → Output N
small change

Trace 1

Trace 2

Trace N-1
• Record execution in a program trace
• When input changes, a change propagation algorithm updates the output and trace as if the program was run “from-scratch”.
• Tries to reuse past computation when possible
Previous work has shown effectiveness for many applications:

<table>
<thead>
<tr>
<th>Application</th>
<th>Time Complexity</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>List primitives (map, reverse, . . .)</td>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td>Sorting: mergesort, quicksort</td>
<td>$O(\log n)$</td>
<td></td>
</tr>
<tr>
<td>2D Convex hulls</td>
<td>$O(\log n)$</td>
<td>[ESA ’06]</td>
</tr>
<tr>
<td>Tree contraction [Miller, Reif ’85]</td>
<td>$O(\log n)$</td>
<td>[SODA ’04].</td>
</tr>
<tr>
<td>3D Convex Hulls</td>
<td>$O(\log n)$</td>
<td>[SCG ’07]</td>
</tr>
<tr>
<td>Meshing in 2D and 3D</td>
<td>$O(\log n)$</td>
<td>[FWCG ’07]</td>
</tr>
<tr>
<td>Bayesian Inference on Trees</td>
<td>$O(\log n)$</td>
<td>[NIPS ’07]</td>
</tr>
<tr>
<td>Bayesian Inference on Graphs</td>
<td>$O(s^d \log n)$</td>
<td>[UAI ’08]</td>
</tr>
</tbody>
</table>

All bounds are randomized (expected time) and are within an expected constant factor of optimal or best known-bounds.
Ordinary programs may be *transformed* into self-adjusting ones

- Special operations added to create/update program trace
- Done either by hand, or via compiler support

Previous work focused on supporting **SML** programs
Motivation for this Work

Want to write self-adjusting computations in C.

Benefits

- Performance (both time and space).
- Large user base
- Broad hardware support (e.g., robots)
- Interoperability with other libraries/software

Challenges

- Memory management
- Ensuring Safety & Correct-usage
Motivation for this Work

**Want to write self-adjusting computations in C.**

### Benefits
- Performance (both time and space).
- Large user base
- Broad hardware support (e.g., robots)
- Interoperability with other libraries/software

### Challenges
- **Memory management**
- Ensuring Safety & Correct-usage

This talk will focus on memory management.
Motivation for this Work

Want to write self-adjusting computations in C.

Some Memory Management Options

- Leave it to the programmer?
  — breaks abstractions of framework
- Use an existing collector?
  — previous work suggests performance problems

Our Approach

Couples memory management with the existing change propagation algorithm.

- Memory allocation recorded in program trace
- Dead objects are identified during change propagation
- Dead objects are reclaimed automatically
Motivation for this Work

Want to write self-adjusting computations in C.

Some Memory Management Options

- Leave it to the programmer?
  — breaks abstractions of framework
- Use an existing collector?
  — previous work suggests performance problems

Our Approach

Couples memory management with the existing change propagation algorithm.

- Memory allocation recorded in program trace
- Dead objects are identified during change propagation
- Dead objects are reclaimed automatically
Examples
A modifiable reference (modref) is a memory cell that stores changeable data.

The input, output and intermediate data of the program is instrumented with modrefs.

To access its contents, a modref is read during a function invocation.

To set its contents, a modref is written.

The program trace stores the program’s callgraph and modref dependencies.
Let’s map a list in a self-adjusting way.

- The input is stored in a modref
- We have to read it to see a list cell
- We are given an empty modref to write the output
Let’s map a list in a self-adjusting way.

- The input is stored in a modref
- We have to read it to see a list cell
- We are given an empty modref to write the output
Example: Mapping a List

Input List

map

Cons Case

- Read input cell
- Map $a \mapsto a'$
- Allocate output cell
- Write output cell
- Recurse on tails
Example: Mapping a List

Cons Case
- Read input cell
- Map \( a \mapsto a' \)
- Allocate output cell
- Write output cell
- Recurse on tails
Example: Mapping a List

Cons Case
- Read input cell
- Map \( a \mapsto a' \)
- Allocate output cell
- Write output cell
- Recurse on tails
Example: Mapping a List

Cons Case
- Read input cell
- Map \(a \mapsto a'\)
- Allocate output cell
- Write output cell
- Recurse on tails
Example: Mapping a List

Cons Case
- Read input cell
- Map \(a \mapsto a'\)
- Allocate output cell
- Write output cell
- Recurse on tails
Example: Mapping a List

Nil Case
- Read nil input
- Write nil output
Example: Mapping a List

Input List

? \(\rightarrow\) read

map

Output Dest

Nil Case
- Read \texttt{nil} input
- Write \texttt{nil} output
Example: Mapping a List

Nil Case
- Read \texttt{nil} input
- Write \texttt{nil} output
Example: Mapping a List

Nil Case
- Read nil input
- Write nil output
Example: Mapping a List

Full trace of mapping \([a, b, c, d] \mapsto [a', b', c', d']\)
Example: Mapping a List

User removes b from input, issues **propagate** command
2nd iteration of map is affected by change (old read value doesn’t match new contents)
System begins re-executing the invocation
Invocation is re-executed using new read value
Maps $c \mapsto c'$
Reuses the cons cell holding $c'$
Example: Mapping a List

Previous owner is out-of-date,
Ultimately it’s removed
Example: Mapping a List

Writes the cons cell to the output destination (readers of this modref are now affected, if any)
Recursive call with arguments:

- `input_list ← read( tail( input_list ) )`
- `output_dest ← tail( new_cons_cell )`
Example: Mapping a List

Recursive call matches a call in the trace
Example: Mapping a List

Matching call is reused
Example: Mapping a List

Allocation of b’ cell is garbage
Output & Trace are consistent with removal of b
When a trace is updated via change propagation, old trace objects are modified and/or replaced with new trace objects.

- **Live trace object**
  - Trace object *retained* in the updated trace.

- **Dead trace object**
  - Trace object *removed* from the updated trace.
History Independence Property

A trace updated via change propagation is consistent with a from-scratch run.
History Independence

New trace, via **change propagation**

New Trace, “**from-scratch**”
Dead allocations (aka garbage) can be attributed to:

1. Live invocations that are re-executed
2. Dead invocations that are removed

Enrich program traces

- Record allocations in the program trace
- Manage allocations during change propagation
Challenges: Dangling Pointers

Must avoid dangling pointers in program trace

- Reclaiming dead objects too soon makes dangling pointers
- History independence implies that an updated trace cannot reach dead objects.

Matthew Hammer
Memory Management for Self-Adjusting Computation
Challenges: Dangling Pointers

Must avoid dangling pointers in program trace

- Reclaiming dead objects too soon makes dangling pointers
- **History independence** implies that an updated trace cannot reach dead objects.
Reuse of calls is essential

- The arguments must match
- Made possible by \textit{reusing} allocated objects
Reuse of calls is essential

- The arguments must match
- Made possible by reusing allocated objects
Challenges: Supporting Reuse

Reuse of calls is essential

- The arguments must match
- Made possible by **reusing** allocated objects
Overview of Our Technique

- **Live Allocations**
  Recorded in program trace with owner.

- **Dead Allocations**
  Removed / Re-executed owner
  Maintained in a list during propagation.

- **Reuse**
  Each assigned a new (live) owner.
  Matching done via user-supplied keys.

- **Reclamation**
  Change propagation complete ⇒
  All dead allocations are garbage
  (i.e., unreachable)

Paper has more details
Overview of Our Technique

- **Live Allocations**
  Recorded in program trace with owner.

- **Dead Allocations**
  Removed / Re-executed owner
  Maintained in a list during propagation.

- **Reuse**
  Each assigned a new (live) owner.
  Matching done via user-supplied keys.

- **Reclamation**
  Change propagation complete ⇒
  All dead allocations are garbage
  (i.e., unreachable)

Paper has more details
Overview of Our Technique

- **Live Allocations**
  Recorded in program trace with owner.

- **Dead Allocations**
  Removed / Re-executed owner
  Maintained in a list during propagation.

- **Reuse**
  Each assigned a new (live) owner.
  Matching done via user-supplied keys.

- **Reclamation**
  Change propagation complete \(\Rightarrow\)
  All dead allocations are garbage
  (i.e., unreachable)

Paper has more details
Overview of Our Technique

- **Live Allocations**
  Recorded in program trace with owner.
- **Dead Allocations**
  Removed / Re-executed owner
  Maintained in a list during propagation.
- **Reuse**
  Each assigned a new (live) owner.
  Matching done via user-supplied keys.
- **Reclamation**
  Change propagation complete ⇒
  All dead allocations are garbage
  (i.e., unreachable)

Paper has more details
Overview of Our Technique

- **Live Allocations**
  Recorded in program trace with owner.

- **Dead Allocations**
  Removed / Re-executed owner
  Maintained in a list during propagation.

- **Reuse**
  Each assigned a new (live) owner.
  Matching done via user-supplied keys.

- **Reclamation**
  Change propagation complete ⇒
  All dead allocations are garbage
  (i.e., unreachable)

Paper has more details
Overview of Our Technique

- **Live Allocations**
  Recorded in program trace with owner.

- **Dead Allocations**
  Removed / Re-executed owner
  Maintained in a list during propagation.

- **Reuse**
  Each assigned a new (live) owner.
  Matching done via user-supplied keys.

- **Reclamation**
  Change propagation complete ⇒
  All dead allocations are garbage
  (i.e., unreachable)

Paper has more details
• Implemented as a library for C
• Primitives are “low-level”,
  (e.g., we don’t enforce correct usage)
• Dead objects reclaimed automatically
Modref Primitives

Creation  \texttt{modref}(\texttt{key}_1, \ldots, \texttt{key}_n)
Writing   \texttt{write}(l, v)
Reading   \texttt{read}(l)

Modrefs . . .
- May be \textit{indexed} by keys.
- Hold \textit{changeable} values.
- Track \textit{read-dependencies}.

Other Primitives

Allocation  \texttt{new}(\texttt{size}, f_i, \texttt{key}_1, \ldots, \texttt{key}_n)
Invocation \texttt{call}(f, \texttt{arg}_1, \ldots, \texttt{arg}_n)
Reads must be in *Normal Form, i.e.,* within a use of `call`

### Not Normal

```c
int x = read(m_1);
int y = x + 1;
write(m_2, y);
```

### Normal

```c
call(increment, read(m_1), m_2);
```

```c
void increment(int x, modref_t* m) {
    write(m, x + 1);
}
```
Reads must be in *Normal Form, i.e.,* within a use of `call`

**Not Normal**

```c
int x = read(m_1);
int y = x + 1;
write(m_2, y);
```

**Normal**

```c
call(incrementer, read(m_1), m_2);
```

```c
void incrementer(int x, modref_t* m) {
    write(m, x + 1);
}
```
List Cell Structure

typedef struct {
  void*  head;
  modref_t*  tail;
} cell_t;

List Cell Allocation

cell_t*  c = new(sizeof(cell_t), cell_init, head);

List Cell Initialization

void cell_init(cell_t*  c, void**  keys) {
  c->head = keys[0];
  c->tail = modref();
}

Allocated blocks are immutable (after being initialized).
Interface Example: Mapping a List

Apply a function $f$ to each element of a given list.

```c
void map(cell_t* c1,
         void* (*f)(void* x),
         modref_t* result)
{
    if (c1 == NULL)
        write(result, NULL);
    else {
        void* y = f(c1->head);
        cell_t* c2 = new(sizeof(cell_t), cell_init, y);
        write(result, c2);
        call(map, read(c1->tail), f, c2->tail);
    }
}
```

To map a list input to output using $f$:

```c
modref_t* output = modref();
call(map, read(input), f, output);
```
Evaluation
Part I
### Benchmarks

#### List Primitives
- **filter**, **map**, **minimum**, and **sum**

#### Sorting
- **quicksort** and **mergesort**

#### Computational Geometry
- **quickhull** finds convex hull
- **diameter** finds diameter of a set of points
- **distance** finds distance between two sets of points

#### Tree Algorithms
- **bstverif** verifies invariants of a binary search tree
- **exprtree** evaluates an expression tree
Overhead

How much **slower** is the self-adjusting program when running “from-scratch”?
Speedup

How much faster can the self-adjusting program update the output for a small change?
### Overhead & Speedup

<table>
<thead>
<tr>
<th>Application</th>
<th>Input Size</th>
<th>Overhead</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>$10^6$</td>
<td>4.2</td>
<td>$1.7 \times 10^5$</td>
</tr>
<tr>
<td>map</td>
<td>$10^6$</td>
<td>2.4</td>
<td>$3.0 \times 10^5$</td>
</tr>
<tr>
<td>minimum</td>
<td>$10^6$</td>
<td>2.6</td>
<td>$1.3 \times 10^5$</td>
</tr>
<tr>
<td>sum</td>
<td>$10^6$</td>
<td>2.4</td>
<td>$1.5 \times 10^4$</td>
</tr>
<tr>
<td>quicksort</td>
<td>$10^5$</td>
<td>2.1</td>
<td>$5.6 \times 10^3$</td>
</tr>
<tr>
<td>mergesort</td>
<td>$10^5$</td>
<td>1.8</td>
<td>$1.3 \times 10^4$</td>
</tr>
<tr>
<td>quickhull</td>
<td>$10^5$</td>
<td>2.1</td>
<td>$1.9 \times 10^3$</td>
</tr>
<tr>
<td>diameter</td>
<td>$10^5$</td>
<td>2.3</td>
<td>$1.9 \times 10^3$</td>
</tr>
<tr>
<td>distance</td>
<td>$10^5$</td>
<td>2.0</td>
<td>$3.5 \times 10^3$</td>
</tr>
<tr>
<td>exprtree</td>
<td>$10^6$</td>
<td>2.3</td>
<td>$1.0 \times 10^4$</td>
</tr>
<tr>
<td>bstverif</td>
<td>$10^6$</td>
<td>3.9</td>
<td>$1.2 \times 10^5$</td>
</tr>
</tbody>
</table>

- On a dual 2Ghz PowerPC G5, 6 GB of memory
- GCC 4.0.2 with "-O3 -combine"
<table>
<thead>
<tr>
<th>Application</th>
<th>Input Size</th>
<th>Overhead</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>$10^6$</td>
<td>4.2</td>
<td>$1.7 \times 10^5$</td>
</tr>
<tr>
<td>map</td>
<td>$10^6$</td>
<td>2.4</td>
<td>$3.0 \times 10^5$</td>
</tr>
<tr>
<td>minimum</td>
<td>$10^6$</td>
<td>2.6</td>
<td>$1.3 \times 10^5$</td>
</tr>
<tr>
<td>sum</td>
<td>$10^6$</td>
<td>2.4</td>
<td>$1.5 \times 10^4$</td>
</tr>
<tr>
<td>quicksort</td>
<td>$10^5$</td>
<td>2.1</td>
<td>$5.6 \times 10^3$</td>
</tr>
<tr>
<td>mergesort</td>
<td>$10^5$</td>
<td>1.8</td>
<td>$1.3 \times 10^4$</td>
</tr>
<tr>
<td>quickhull</td>
<td>$10^5$</td>
<td>2.1</td>
<td>$1.9 \times 10^3$</td>
</tr>
<tr>
<td>diameter</td>
<td>$10^5$</td>
<td>2.3</td>
<td>$1.9 \times 10^3$</td>
</tr>
<tr>
<td>distance</td>
<td>$10^5$</td>
<td>2.0</td>
<td>$3.5 \times 10^3$</td>
</tr>
<tr>
<td>exprtree</td>
<td>$10^6$</td>
<td>2.3</td>
<td>$1.0 \times 10^4$</td>
</tr>
<tr>
<td>bstverif</td>
<td>$10^6$</td>
<td>3.9</td>
<td>$1.2 \times 10^5$</td>
</tr>
</tbody>
</table>

- Average overhead is 2 to 3x;
- Overhead is scalable, i.e., $O(1)$
- Speedups range from three to five orders of magnitude
Evaluation Part II: Comparison to SML
## Evaluation: Setup & Measurements

### Measurements

<table>
<thead>
<tr>
<th>SML+GC</th>
<th>SML code including GC time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SML-GC</td>
<td>SML code excluding GC time</td>
</tr>
</tbody>
</table>

### Benchmarks

**List Primitives and Sorting:** filter, map, minimum, sum

**Computational Geometry:** quickhull, diameter

### Setup

**SML:** MLton with 

```
-runtime "ram-slop 1.0"
```
First Observations

- **SML** timings excluding GC comparable to **C** timings.
- **SML** timings including GC become 10x slower.
First Observations

- **SML** timings excluding GC comparable to **C** timings.
- **SML** timings including GC become 10x slower.
MLton uses a set of conventional tracing collectors (copying and mark-sweep).

**Analysis**

For tracing collectors, each reclaimed location costs

$$O \left( \frac{1}{1 - r} \right)$$

where $0 \leq r < 1$ is the fraction of live memory.

**Observation**

Execution traces often consume large fractions of available memory, *i.e.*, $r$ can approach 1 during normal usage.
Quicksort: Tracing GC Cost

Tracing GC Cost
(bytes traversed by GC) / (bytes allocated)

Traversed / Allocated
Input Size ($n \times 10^3$)
Quicksort Change Propagation

Cost increases for larger input-sizes (with larger traces).

Plot of $\frac{1}{1-r} - 1$
Generational Approaches

What about generations?

- By partitioning objects into two or more generations GC avoids tracing the entire heap for each collection.
- Generational approach makes several assumptions.
- Program traces violate each of these.

<table>
<thead>
<tr>
<th>Generational Assumption</th>
<th>Violation by Program Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects die young</td>
<td>New objects are long-lived</td>
</tr>
<tr>
<td>Old objects are unlikely to die</td>
<td>Removed objects are often old</td>
</tr>
<tr>
<td>Old-to-new pointers are rare</td>
<td>Old-to-new pointers are common</td>
</tr>
</tbody>
</table>
Evaluation: From-Scratch Time (sec)

- Filter
- Map
- Minimum
- Sum
- Quicksort
- Mergesort
- Quickhull
- Diameter

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>SML+GC</th>
<th>SML-GC</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter</td>
<td></td>
<td></td>
<td>74.2</td>
</tr>
<tr>
<td>Map</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minimum</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quicksort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mergesort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quickhull</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diameter</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison Summary

Self-Adj. C vs Self-Adj. SML

- 40-75% reduction of space usage.
- Excluding SML GC time, they are comparable.
- Including SML GC time, C versions up to 10x faster.
### Related Work

#### Reference Counting
- Also has $O(1)$ bound
- Well-known challenges with overhead of counters, and with cyclic structures.

#### Region-based Approaches
- Also organize objects according to “scope”
- Usually don’t support objects moving between regions
Future Work

On-going

Front-end for C and improved runtime:
- Simpler interface
  (e.g., reads used more naturally)
- Imperative modrefs
  (i.e., multiple writes)
- More optimizations and safety-checks

Future

- Integration with existing, tracing collectors
- Integration with existing, region-based approaches
• Memory management of self-adjusting propagation . . .
  • Couples nicely with tracing and change propagation.
  • But requires some care for correctness and reuse.
• The result realizes the asymptotic bounds we wanted.
• The C implementation outperforms previous implementations in both time and space.
Thank You!

Questions?