Memory Management for Self-Adjusting Computation

Matthew Hammer   Umut Acar

Toyota Technological Institute at Chicago

International Symposium on Memory Management, 2008

Overview of Talk

- Previous frameworks written in SML
- We implement a framework for C

In this talk, we

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

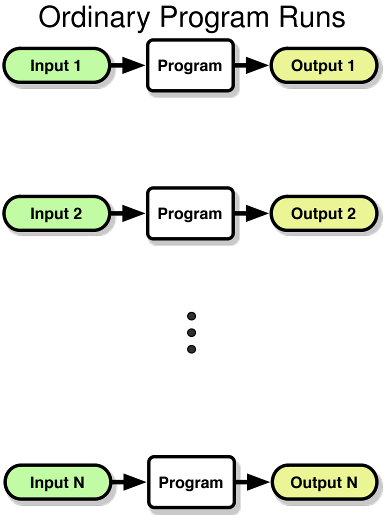
Self-Adjusting Computation

Motivation : Incremental change is pervasive.

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

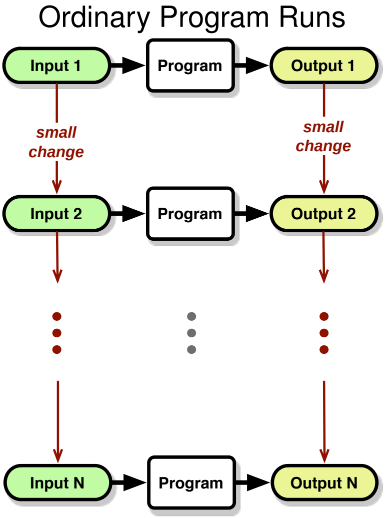
- Applications that interact with a physical environment.
  E.g., Robots.
- Applications that interact with a user.
  E.g., Games, Editors, Compilers, etc.
- Application that rely on modeling or simulation.
  E.g., Scientific Computing, Computational Biology, Motion Simulation.

Self-Adjusting Computation



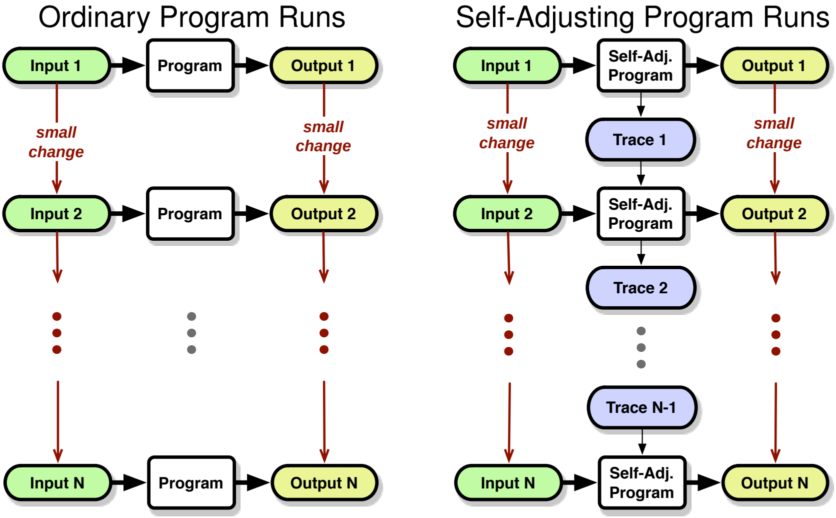
- Ordinary programs often run repeatedly on changing input.
- What if input and output change by only small increments?

Self-Adjusting Computation



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

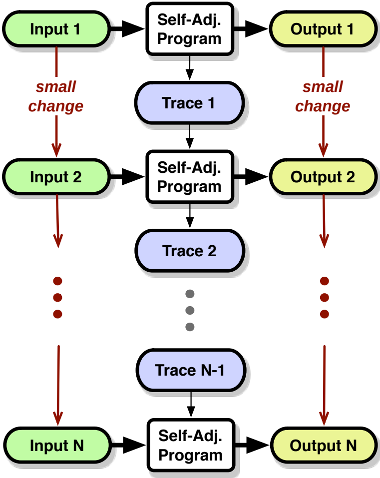
Self-Adjusting Computation



Self-Adjusting Computation

- 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

Self-Adjusting Program Runs



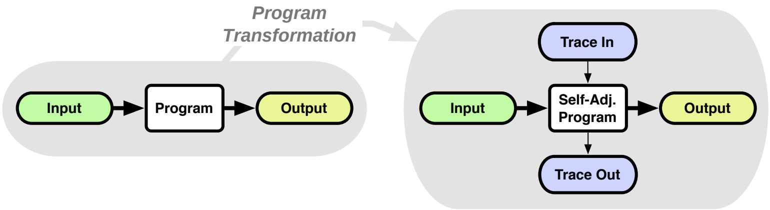
Self-Adjusting Computation

Previous work has shown effectiveness for many applications:

List primitives (map, reverse, ...)	$O(1)$	
Sorting: mergesort, quicksort	$O(\log n)$	
2D Convex hulls	$O(\log n)$	[ESA '06]
Tree contraction [Miller, Reif '85]	$O(\log n)$	[SODA '04].
3D Convex Hulls	$O(\log n)$	[SCG '07]
Meshing in 2D and 3D	$O(\log n)$	[FWCG '07]
Bayesian Inference on Trees	$O(\log n)$	[NIPS '07]
Bayesian Inference on Graphs	$O(s^d \log n)$	[UAI '08]

All bounds are randomized (expected time) and are within an expected constant factor of optimal or best known-bounds.

Writing Self-Adjusting Programs



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

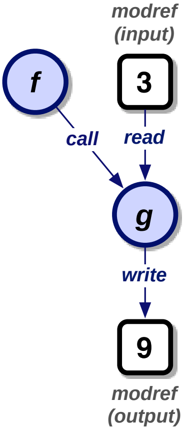
Our Approach

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

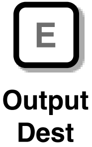
Examples

Self-Adjusting Primitives



- 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.

Example: Mapping a List



Let's **map** a list in a self-adjusting way.

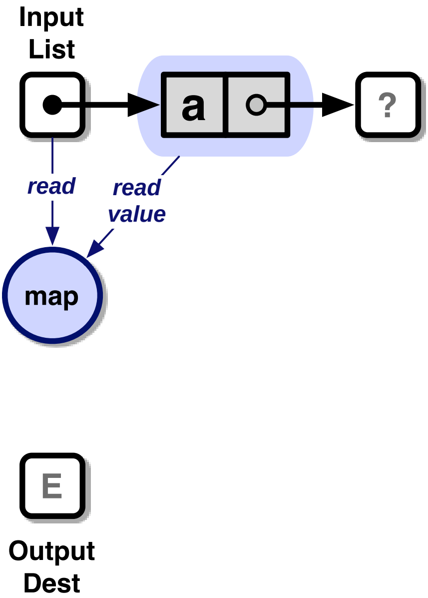
- The input is stored in a **modref**
- We have to **read** it to see a list cell
- We are given an empty modref to **write** the output

Example: Mapping a List

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

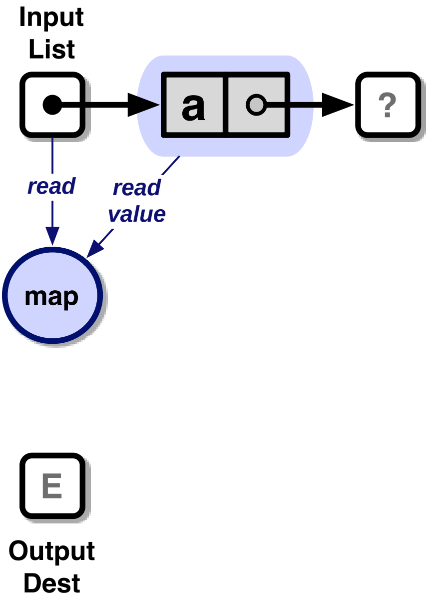
Example: Mapping a List



Cons Case

- Read input cell
- Map **a** $\mapsto$ **a'**
- Allocate output cell
- Write output cell
- Recurse on tails

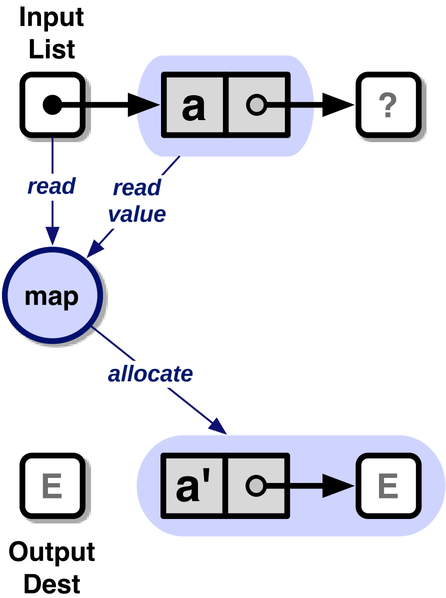
Example: Mapping a List



Cons Case

- Read input cell
- Map **a** $\mapsto$ **a'**
- Allocate output cell
- Write output cell
- Recurse on tails

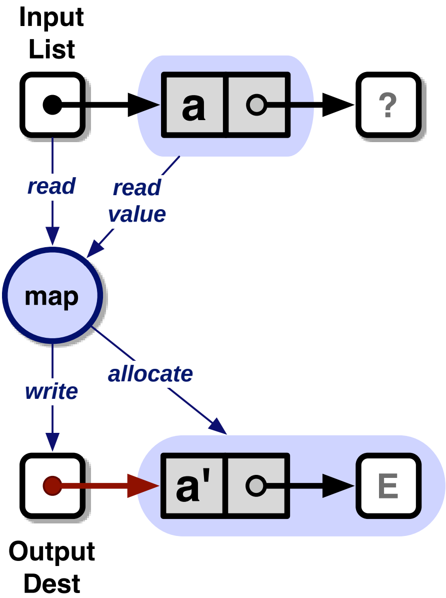
Example: Mapping a List



Cons Case

- Read input cell
- Map **a** $\mapsto$ **a'**
- Allocate output cell
- Write output cell
- Recurse on tails

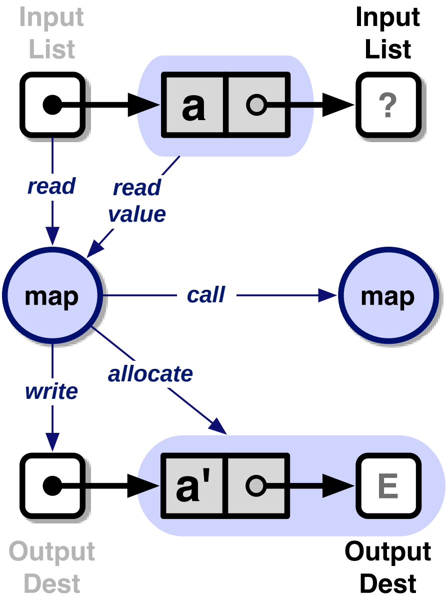
Example: Mapping a List



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

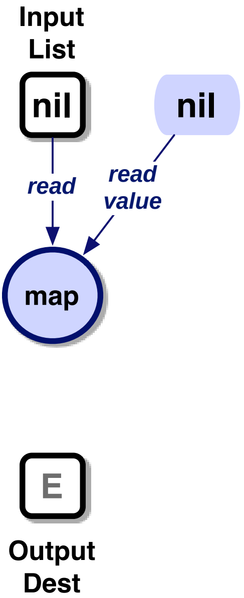
Nil Case

- Read **nil** input
- Write **nil** output

Nil Case

- Read **nil** input
- Write **nil** output

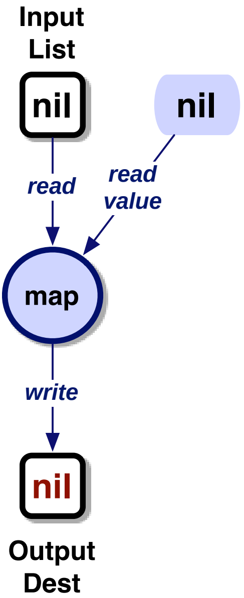
Example: Mapping a List



Nil Case

- Read **nil** input
- Write **nil** output

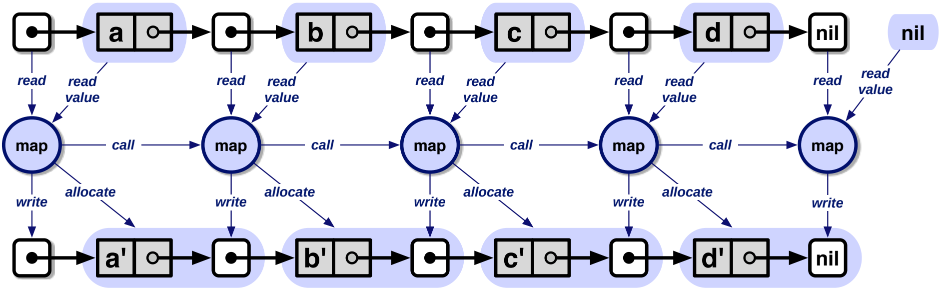
Example: Mapping a List



Nil Case

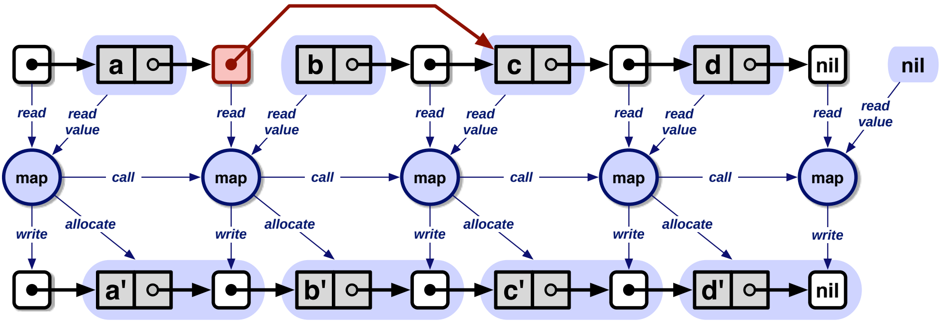
- Read **nil** input
- Write **nil** output

Example: Mapping a List



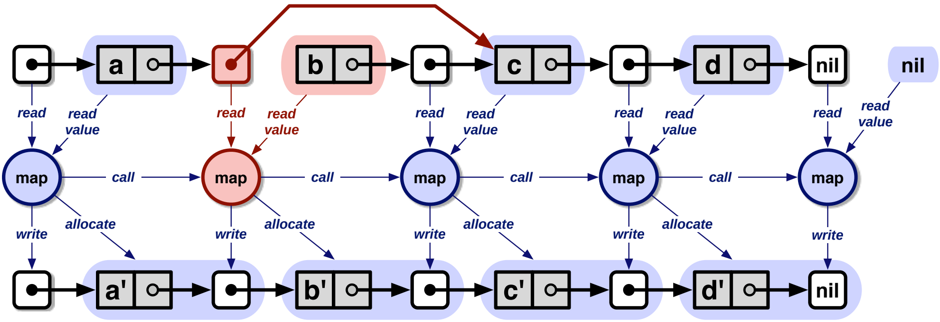
Full trace of mapping **[a,b,c,d]** $\mapsto$ **[a’,b’,c’,d’]**

Example: Mapping a List



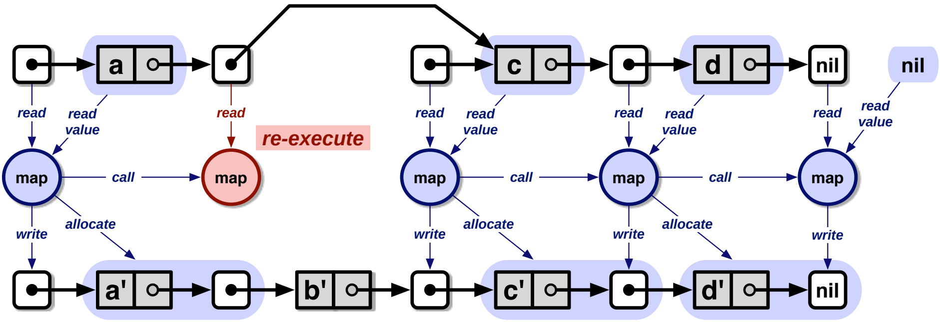
User removes **b** from input, issues **propagate** command

Example: Mapping a List



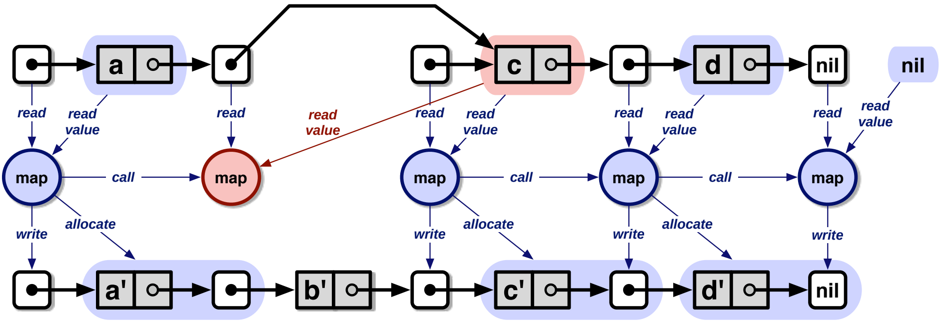
2nd iteration of **map** is affected by change
(old read value doesn't match new contents)

Example: Mapping a List



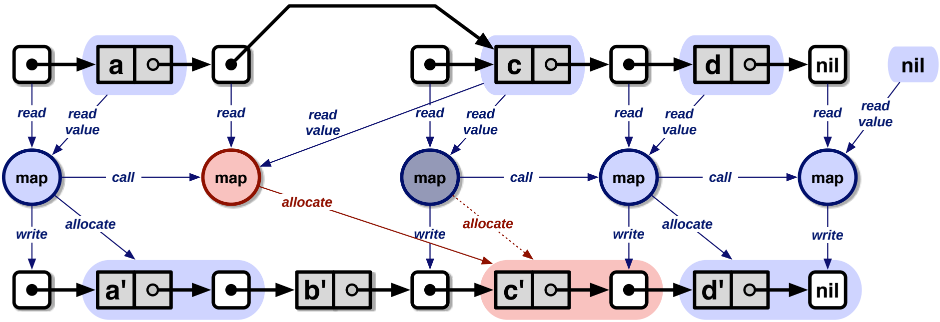
System begins re-executing the invocation

Example: Mapping a List



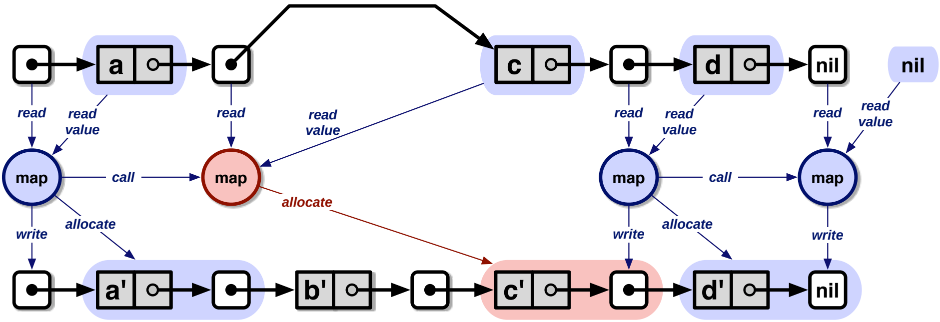
Invocation is re-executed using new read value

Example: Mapping a List



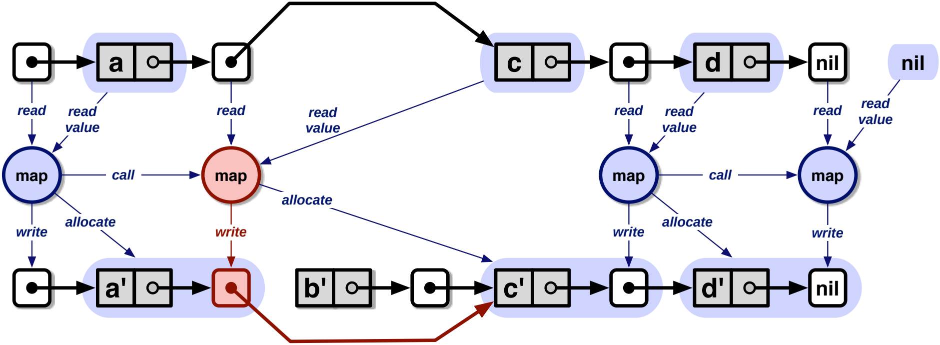
Maps **c** $\mapsto$ **c'**
Reuses the cons cell holding **c'**

Example: Mapping a List



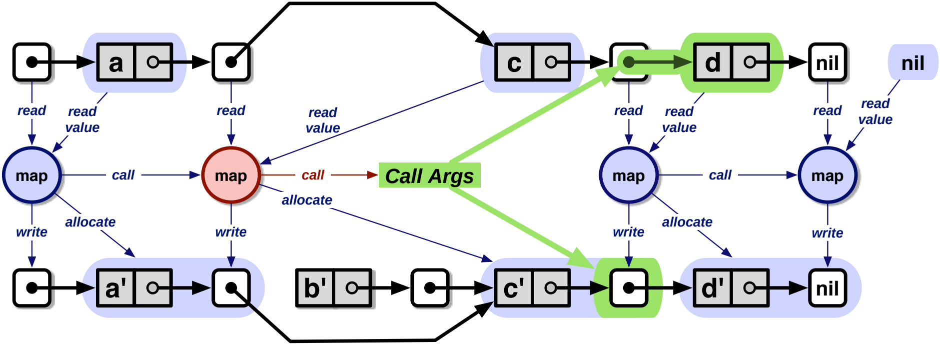
Previous owner is out-of-date,
Ultimately it's removed

Example: Mapping a List



Writes the cons cell to the output destination
(readers of this modref are now affected, if any)

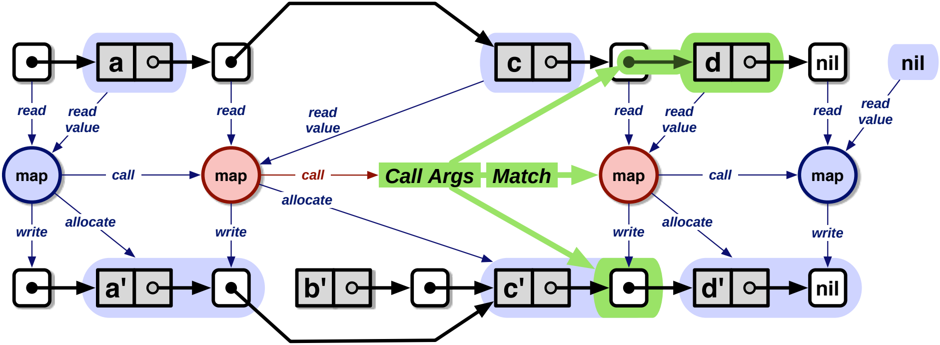
Example: Mapping a List



Recursive call with arguments:

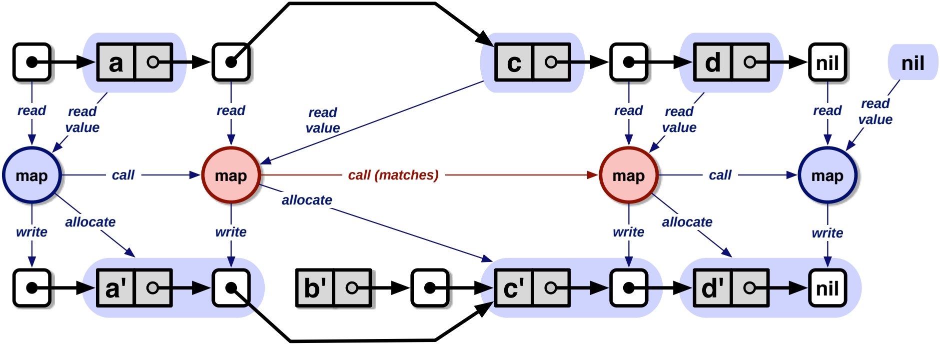
- *input_list* ← **read(tail(** *input_list* **))**
- *output_dest* ← **tail(** *new_cons_cell* **)**

Example: Mapping a List



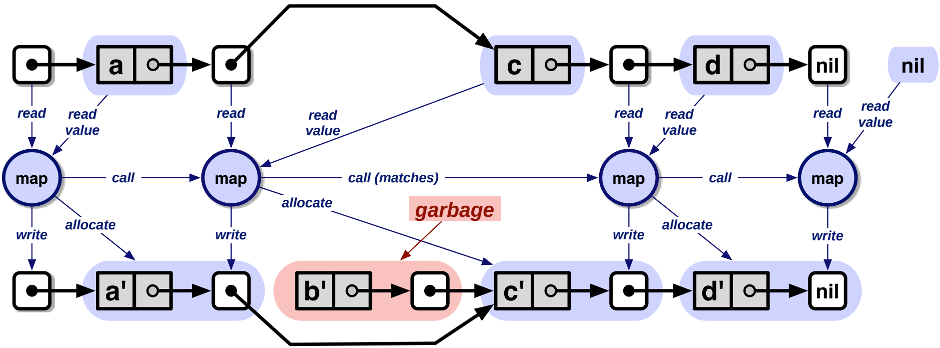
Recursive call matches a call in the trace

Example: Mapping a List



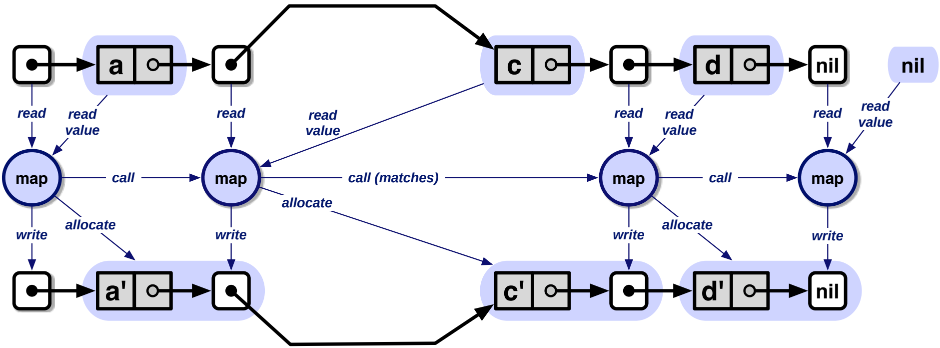
Matching call is reused

Example: Mapping a List



Allocation of **b’** cell is garbage

Example: Mapping a List



Output & Trace are consistent with removal of **b**

Program Traces

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.

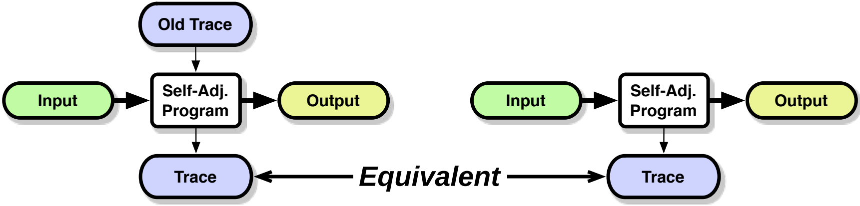
Dead trace object

Trace object **removed** from the updated trace.

History Independence

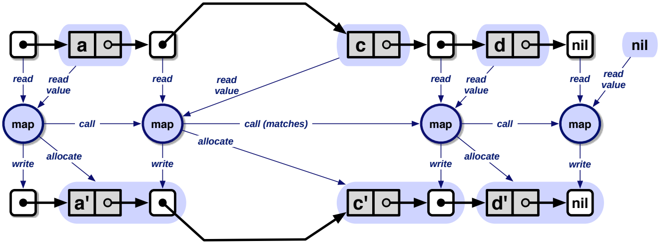
History Independence Property

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

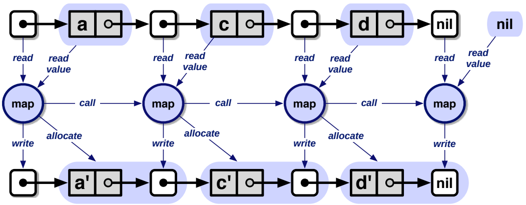


History Independence

New trace, via **change propagation**



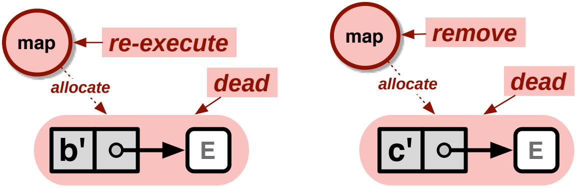
New Trace, “**from-scratch**”



The Rough Idea for Identifying Garbage

Dead allocations (aka garbage) can be attributed to:

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



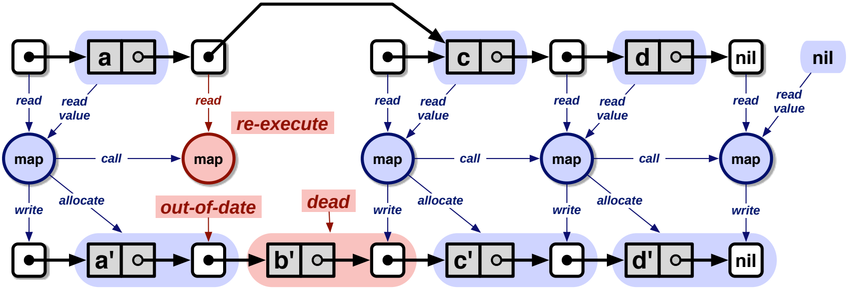
Enrich program traces

- Record allocations in the program trace
- Manage allocations during change propagation

Challenges: Dangling Pointers

Must avoid dangling pointers in program trace

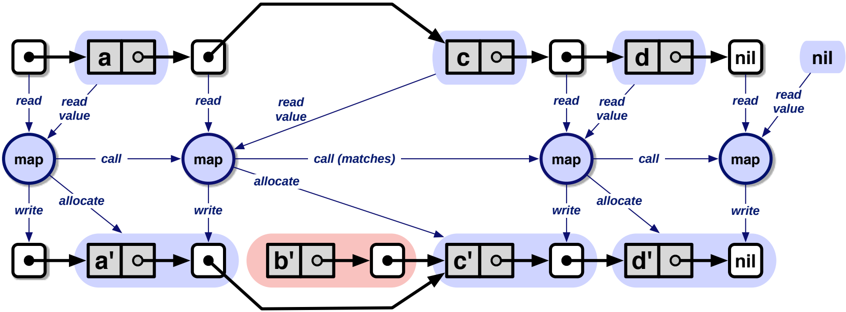
- Reclaiming dead objects too soon makes dangling pointers
- **History independence** implies that an updated trace cannot reach dead objects.



Challenges: Dangling Pointers

Must avoid dangling pointers in program trace

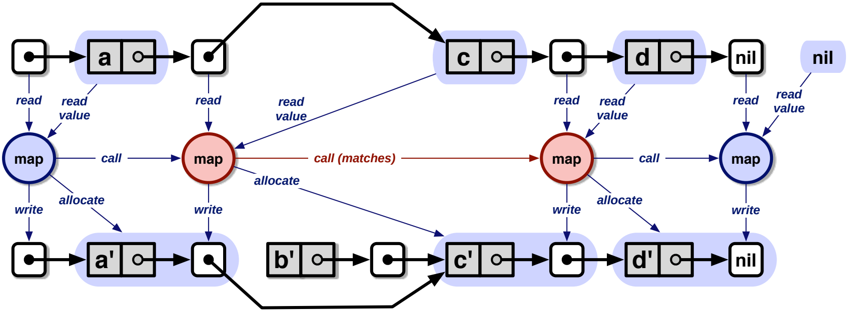
- Reclaiming dead objects too soon makes dangling pointers
- **History independence** implies that an updated trace cannot reach dead objects.



Challenges: Supporting Reuse

Reuse of calls is essential

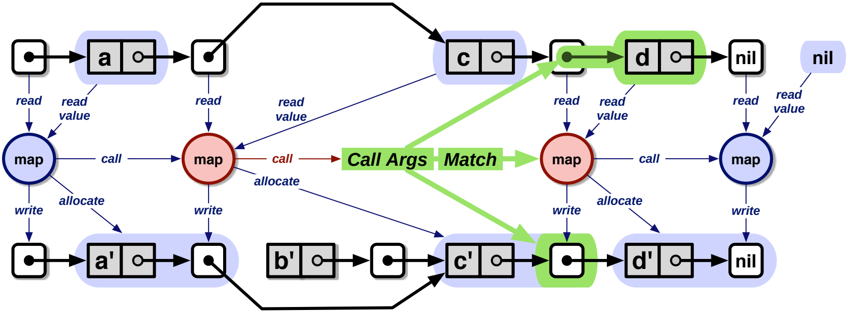
- The arguments must match
- Made possible by **reusing** allocated objects



Challenges: Supporting Reuse

Reuse of calls is essential

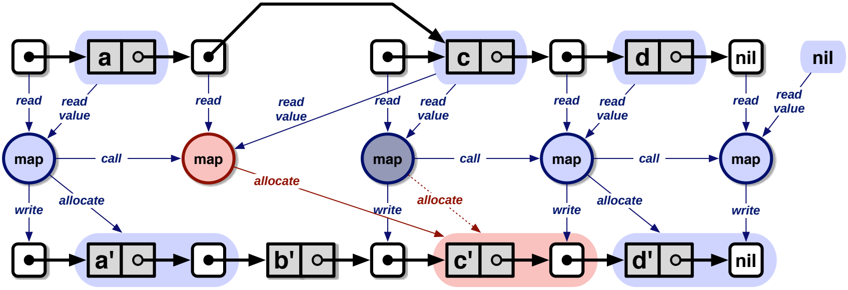
- The arguments must match
- Made possible by **reusing** allocated objects



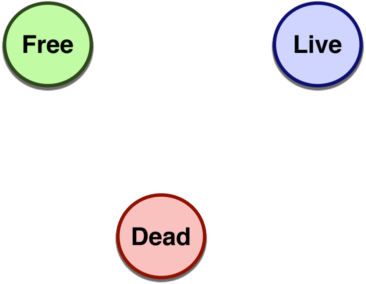
Challenges: Supporting Reuse

Reuse of calls is essential

- The arguments must match
- Made possible by **reusing** allocated objects



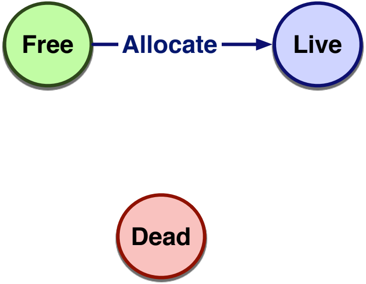
Overview of Our Technique



- **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

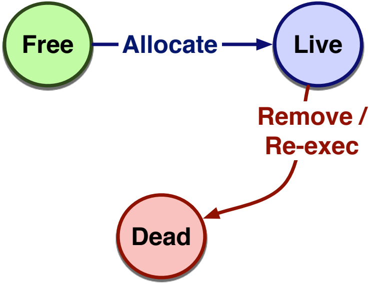
Overview of Our Technique



- **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

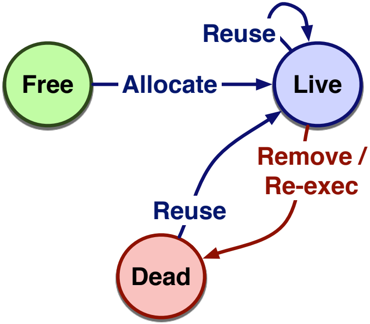
Overview of Our Technique



- **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 ⇒
  All dead allocations are garbage
  (i.e., unreachable)

Paper has more details

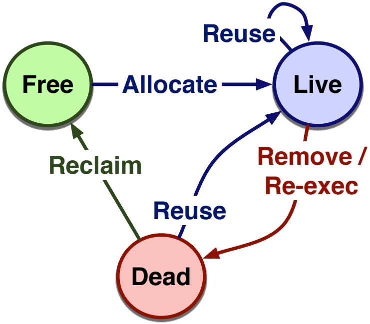
Overview of Our Technique



- **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

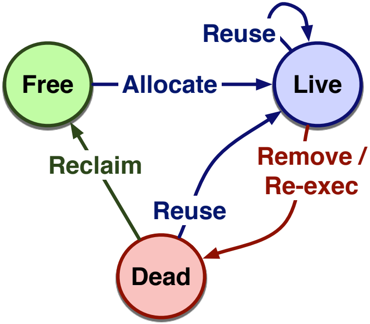
Overview of Our Technique



- **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 $\Rightarrow$
  All dead allocations are garbage
  (i.e., unreachable)

Paper has more details

Implementation: Overview

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

Programming Interface

Modref Primitives

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

Modrefs ...

- May be *indexed* by keys.
- Hold *changeable* values.
- Track *read-dependencies*.

Other Primitives

Allocation	**new** $(\mathtt{size}, f_i, \mathtt{key}_1, \ldots, \mathtt{key}_n)$
Invocation	**call** $(f, \mathtt{arg}_1, \ldots, \mathtt{arg}_n)$

Interface: Normal Form Programs

Reads must be in *Normal Form*, *i.e.*, within a use of `call`

Not Normal

```
int x = read(m1);
int y = x + 1;
write(m2, y);
```

Normal

```
call(incr, read(m1), m2);
```

```
void incr(int x, modref_t* m) {
  write(m, x + 1);
}
```

Interface: Normal Form Programs

Reads must be in *Normal Form*, *i.e.*, within a use of `call`

Not Normal

```
int x = read(m1);
int y = x + 1;
write(m2, y);
```

Normal

```
call(incr, read(m1), m2);
```

```
void incr(int x, modref_t* m) {
  write(m, x + 1);
}
```

Interface: Allocation

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.

```
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`:

```
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**

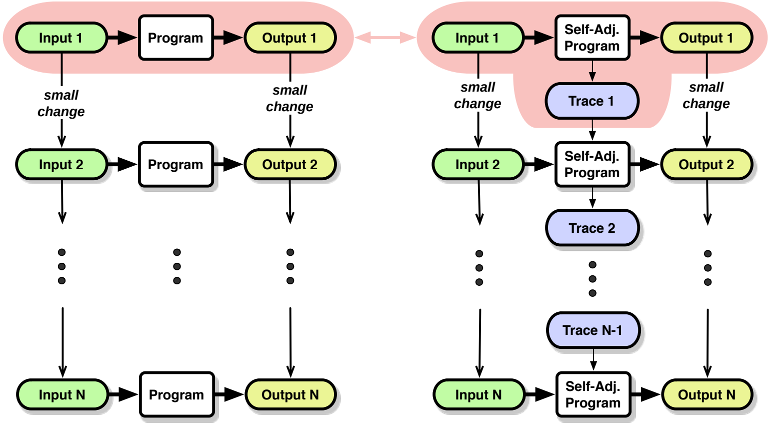
Computational Geometry

- **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

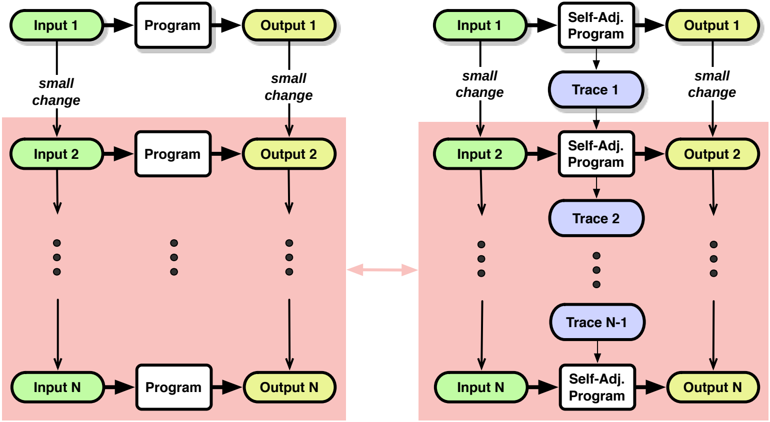
Overhead



Overhead

How much **slower** is the self-adjusting program when running "from-scratch"?

Speedup



Speedup

How much **faster** can the self-adjusting program update the output for a small change?

Overhead & Speedup

Application	Input Size	Overhead	Speedup
`filter`	10^6	4.2	1.7×10^5
`map`	10^6	2.4	3.0×10^5
`minimum`	10^6	2.6	1.3×10^5
`sum`	10^6	2.4	1.5×10^4
`quicksort`	10^5	2.1	5.6×10^3
`mergesort`	10^5	1.8	1.3×10^4
`quickhull`	10^5	2.1	1.9×10^3
`diameter`	10^5	2.3	1.9×10^3
`distance`	10^5	2.0	3.5×10^3
`exprtree`	10^6	2.3	1.0×10^4
`bstverif`	10^6	3.9	1.2×10^5

- On a dual 2Ghz PowerPC G5, 6 GB of memory
- GCC 4.0.2 with "`-O3 -combine`"

Overhead & Speedup

Application	Input Size	Overhead	Speedup
`filter`	10^6	4.2	1.7×10^5
`map`	10^6	2.4	3.0×10^5
`minimum`	10^6	2.6	1.3×10^5
`sum`	10^6	2.4	1.5×10^4
`quicksort`	10^5	2.1	5.6×10^3
`mergesort`	10^5	1.8	1.3×10^4
`quickhull`	10^5	2.1	1.9×10^3
`diameter`	10^5	2.3	1.9×10^3
`distance`	10^5	2.0	3.5×10^3
`exprtree`	10^6	2.3	1.0×10^4
`bstverif`	10^6	3.9	1.2×10^5

- 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

SML+GC SML code including GC time

SML-GC SML code excluding GC time

Benchmarks

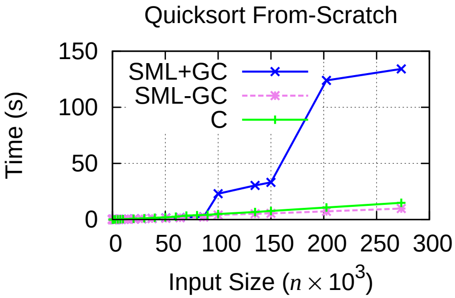
List Primitives and Sorting: filter, map, minimum, sum

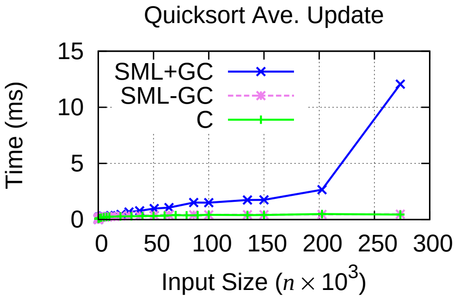
Computational Geometry: quickhull, diameter

Setup

SML: MLton with “`-runtime "ram-slop 1.0"`”

Quicksort: Timing comparison

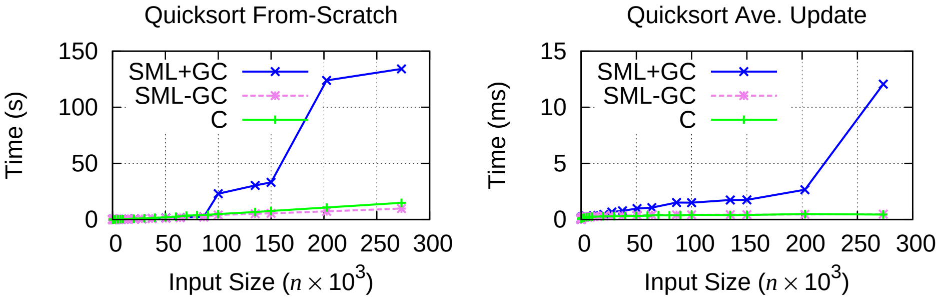




First Observations

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

Quicksort: Timing comparison



First Observations

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

Tracing GC Cost

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.

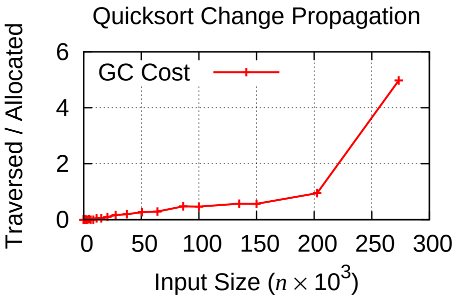
Observation

Execution traces often consume large fractions of available memory, *i.e.*, r can approach 1 during normal usage.

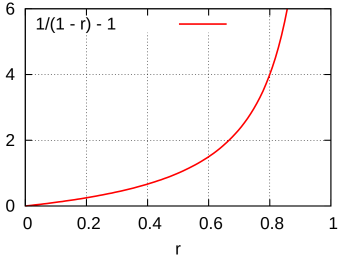
Quicksort: Tracing GC Cost

Tracing GC Cost

(bytes traversed by GC) / (bytes allocated)



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



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

Generational Approaches

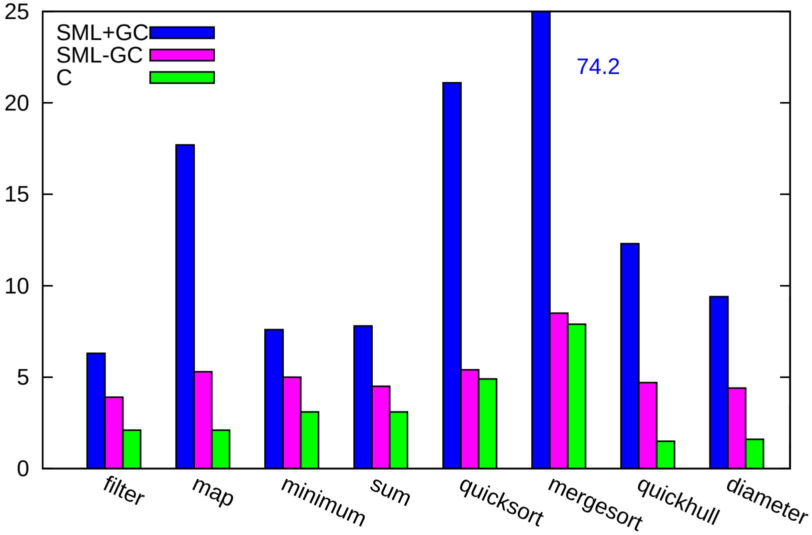
What about generations?

- 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.

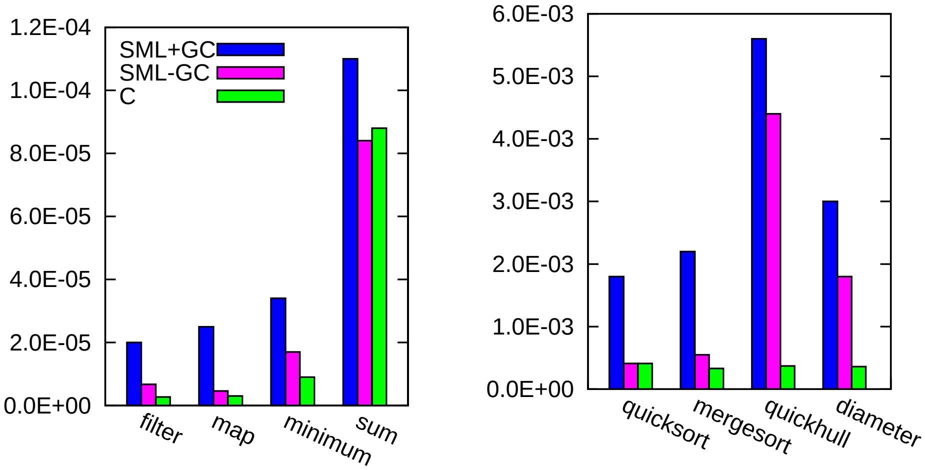
Generational Assumption
Objects die young
Old objects are unlikely to die
Old-to-new pointers are rare

Violation by Program Trace
New objects are long-lived
Removed objects are often old
Old-to-new pointers are common

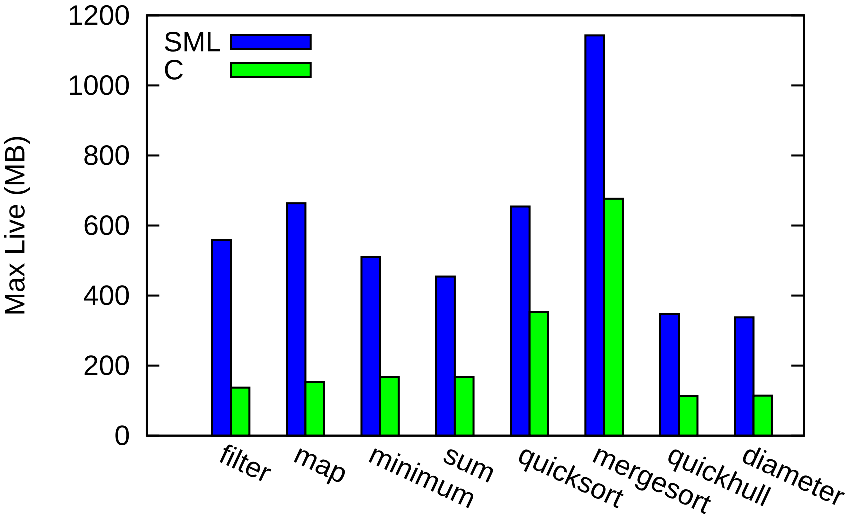
Evaluation: From-Scratch Time (sec)



Evaluation: Average Update Time (sec)



Evaluation: Space



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:

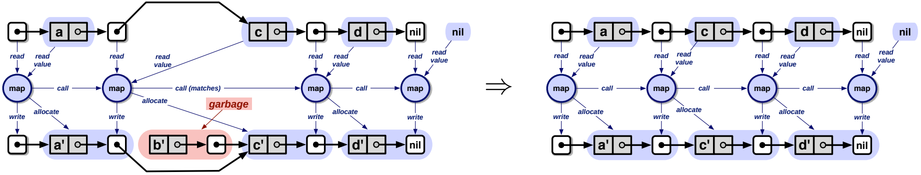
- Simpler interface
  (e.g., reads used more naturally)
- Imperative modrefs
  (i.e., multiple writes)
- More optimizations and safety-checks

Future

- Integration with existing, tracing collectors
- Integration with existing, region-based approaches

Summary

- 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.



Thanks, Questions

Thank You!

Questions?