Self-Adjusting Machines

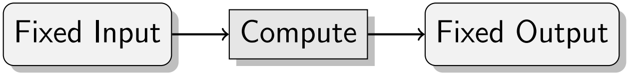
Matthew A. Hammer

University of Chicago
Max Planck Institute for Software Systems

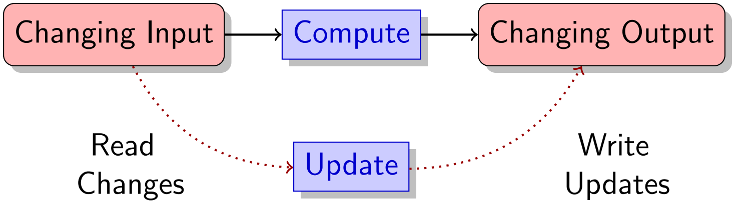
Thesis Defense
July 20, 2012
Chicago, IL

Static Computation Versus Dynamic Computation

Static Computation:

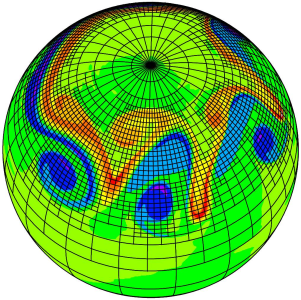


Dynamic Computation:

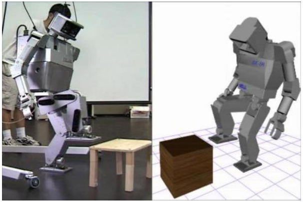


Dynamic Data is Everywhere

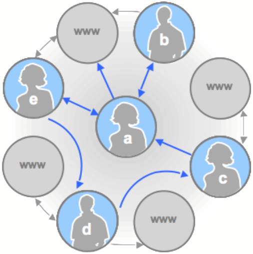
Software systems often consume/produce dynamic data



Scientific Simulation



Reactive Systems



Analysis of Internet data

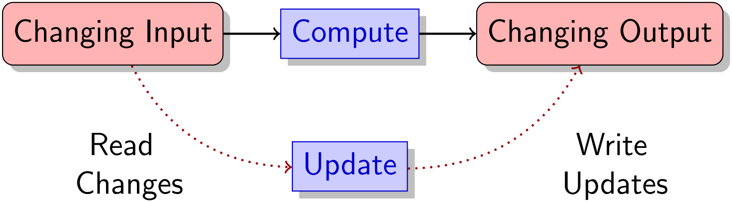
Tractability Requires Dynamic Computations

Changing Input → Compute → Changing Output

Static Case
(Re-evaluation "from scratch")

compute	1 sec
# of changes	1 million
Total time	11.6 days

Tractability Requires Dynamic Computations



Static Case
(Re-evaluation "from scratch")

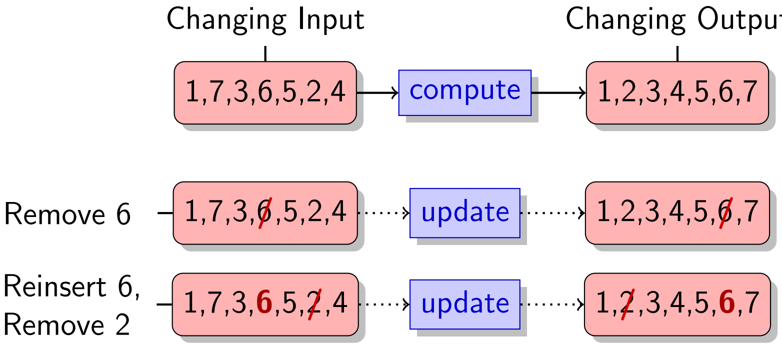
compute	1 sec
# of changes	1 million
Total time	11.6 days

Dynamic Case
(Uses update mechanism)

compute	10 sec
update	1×10^{-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.



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**

Domain knowledge + Library primitives

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**

Language feature	High-level help	Low-level gap
Type system	Indicates mutability	**Everything mutable**
Functions	Higher-order traces	**Closures are manual**
Stack space	Alters stack profile	**Bounded stack space**
Heap management	Automatic GC	**Explicit management**

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**;

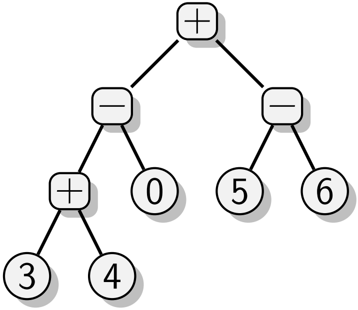
via practical **compilation** and **run-time techniques**, these machines are **programmable**, **sound** and **efficient**.

Contributions

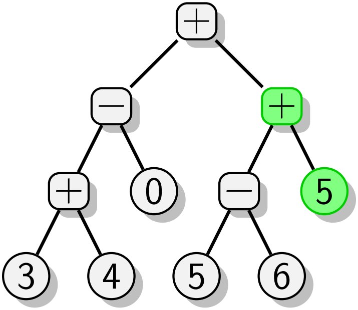
Surface language, C-based	**Programmable**
Abstact machine model	**Sound**
Compiler	Realizes static aspects
Run-time library	Realizes dynamic aspects
Empirical evaluation	**Efficient**

Example: Dynamic Expression Trees

Objective: As tree changes, maintain its valuation



$((3+4)-0)+(5-6)=6$



$((3+4)-0)+\color{red}{(}(5-6)\color{red}{+5)}\color{red}{=11}$

Consistency: Output is correct valuation
Efficiency: Update time is $O(\#\text{affected intermediate results})$

Expression Tree Evaluation in 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

```
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**:

The Stack "Shapes" the Computation

```
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

The Stack "Shapes" the Computation

```
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

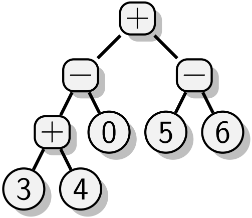
```
int eval (node_t root) {
  if (root->tag == LEAF)
    return root->u.leaf;
  else {
    int l = eval (root->u.binop.left);
    int r = eval (root->u.binop.right);
    if (root->u.binop.op == PLUS) return (l + r);
    else return (l - r);
  } }
```

Stack usage breaks computation into **three parts**:

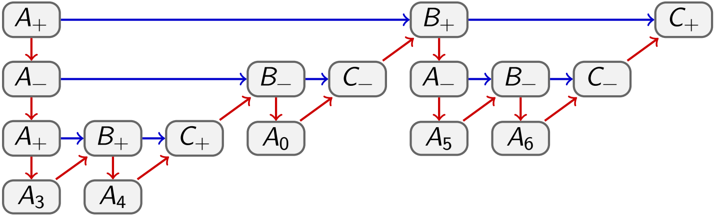
- **Part A**: Return value if `LEAF`
  Otherwise, evaluate `BINOP`, starting with left child
- **Part B**: Evaluate the right child
- **Part C**: Apply `BINOP` to intermediate results; return

Dynamic Execution Traces

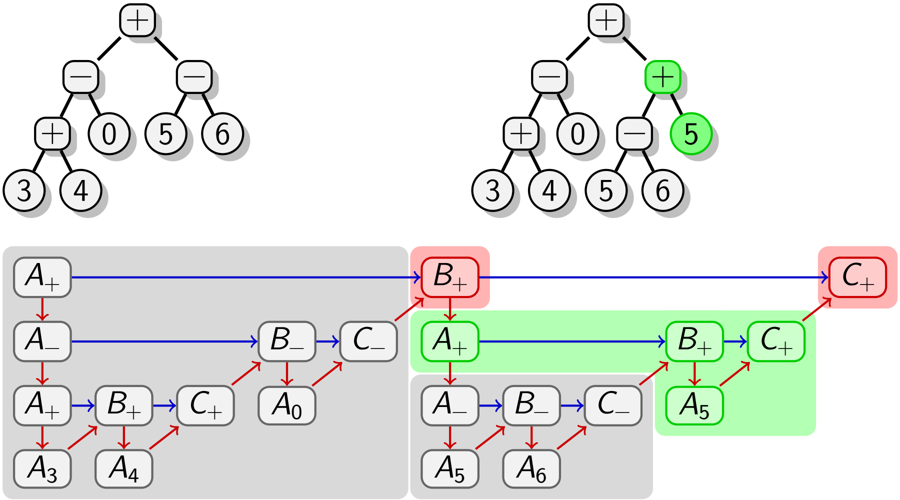
Input Tree



Execution Trace

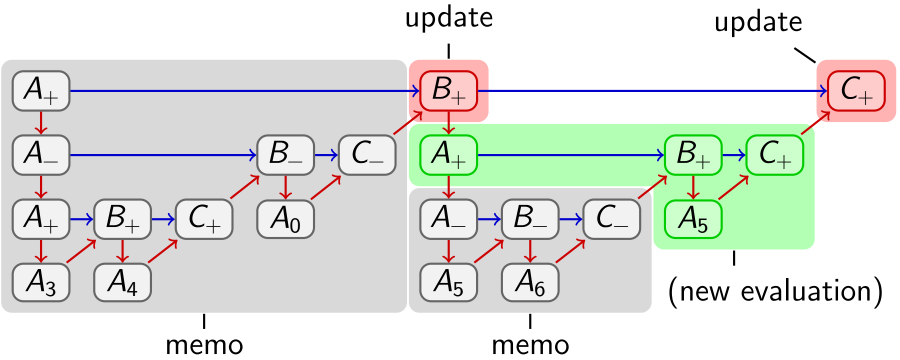


Updating inputs, traces and outputs



Core self-adjusting primitives

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



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)

Abstract machine semantics

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

Needed property: Store agnosticism

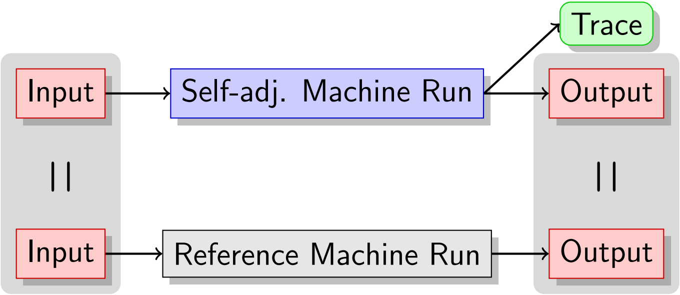
An IL program is **store agnostic** when each stack frame has a fixed return value; hence, not affected by **update** points

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

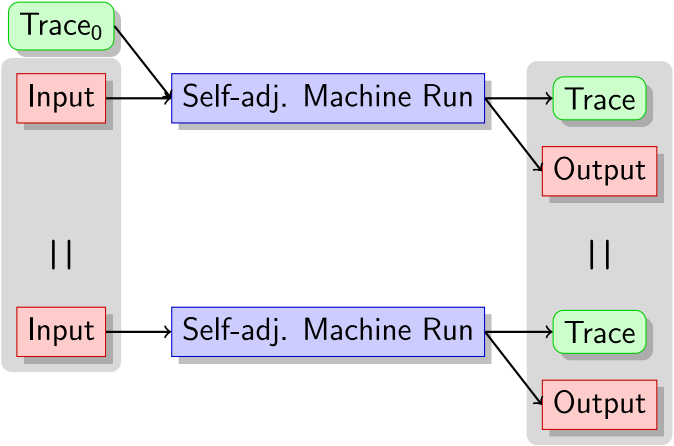
- Lemma: DPS-conversion preserves program meaning
- Lemma: DPS-conversion acheives store agnosticism

Consistency theorem, Part 1: No Reuse



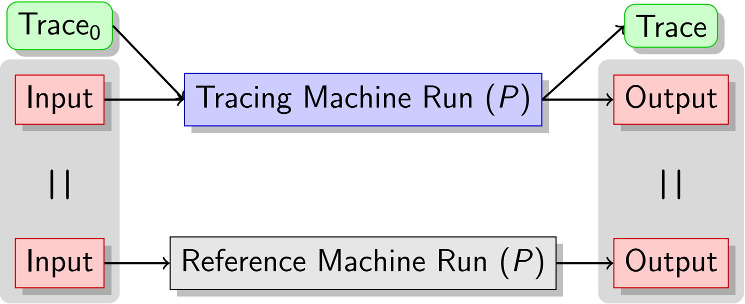
Self-adjusting machine is consistent with reference machine
when self-adjusting machine runs "from-scratch", with no reuse

Consistency theorem, Part 2: Reuse vs No Reuse



Self-adjusting 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:

Self-adjusting machine is consistent with reference machine

Concrete
Self-adjusting machines

From abstract to concrete 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

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

Sparser traces — avoid tracing when possible

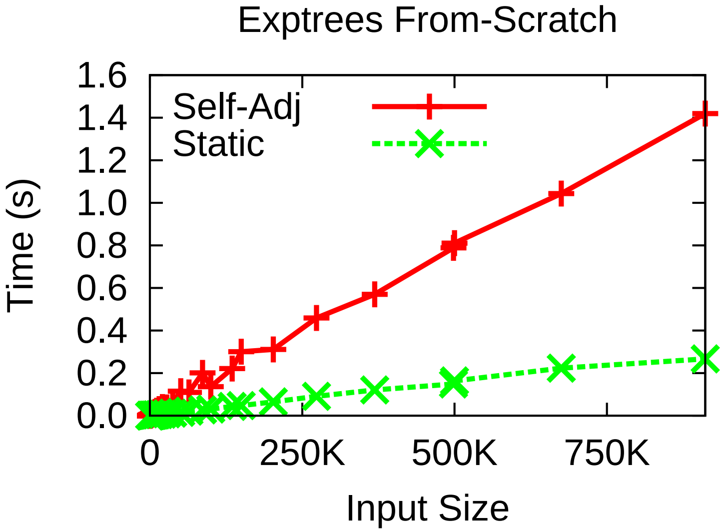
1. **Stable references**	Programmer uses type qualifier
2. **Selective DPS**	Compiler analysis of update points

Cheaper traces — more efficient representation

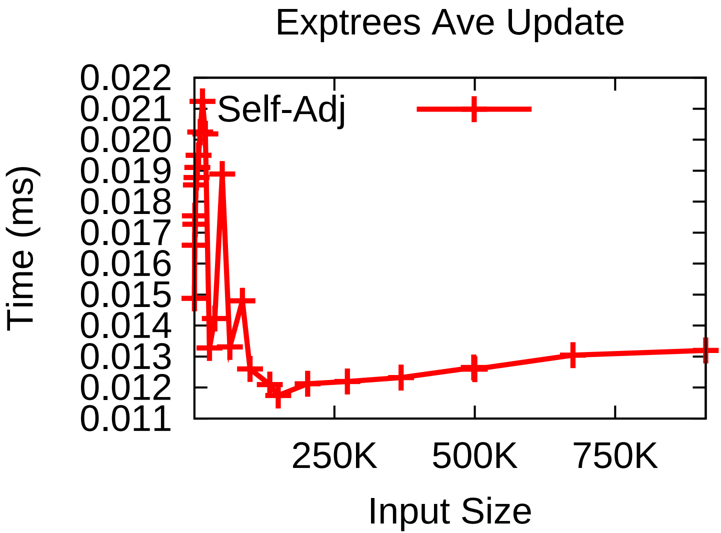
3. **Write-once memory**	Programmer uses type qualifier
4. **Trace node sharing**	Compiler analysis coalesces traced ops

Evaluation

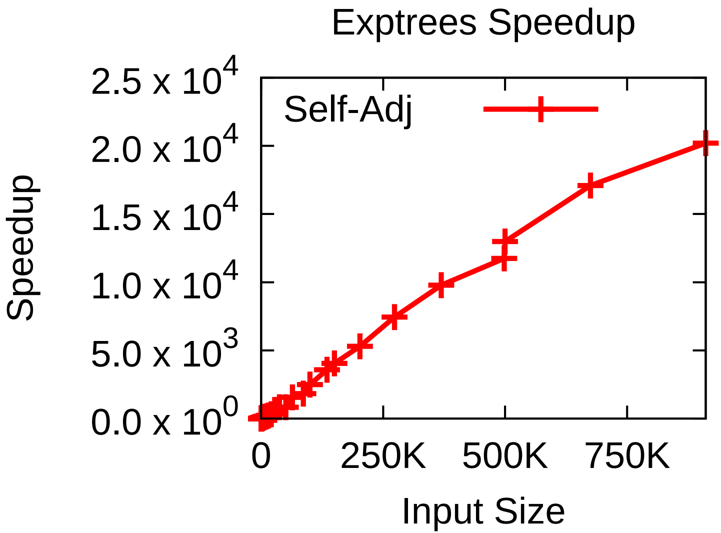
From-scratch time: Constant overhead



Average update time: Constant time



Speed up = From-scratch / Update



Evolution of our approach

Stage 1: **First run-time library**

- \+ Change propagation & memory management
- – Very high programmer burden

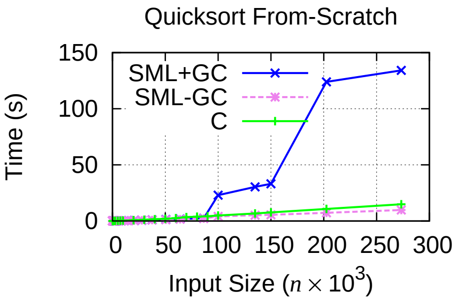
Stage 2: **First compiler**

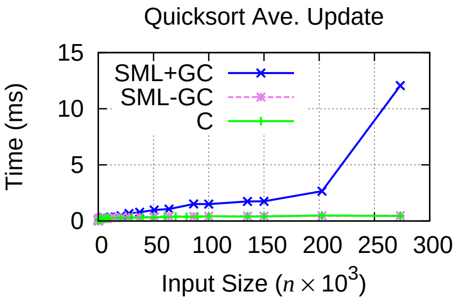
- \+ Lower programmer burden
- – No return values
- – Memo points are non-orthogonal (conflated with read and alloc primitives)
- – No model for consistency or optimizations

Stage 3: **New compiler & run-time library**

- \+ Self-adjuting 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

Stage 2, Basic compiler: CEAL *vs* Delta-ML

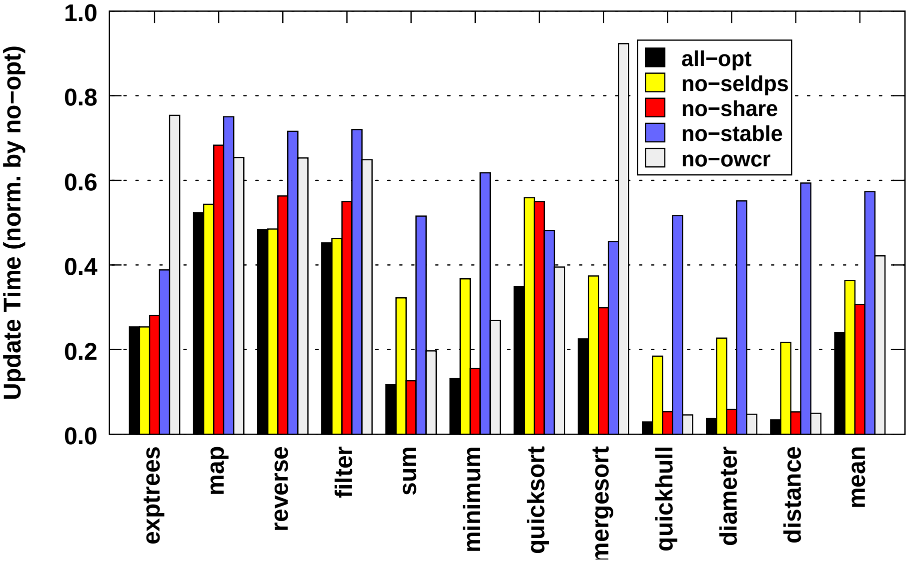
Normalized Measurements [(CEAL / DeltaML) $\times$ 100]

App.	From-Scratch	Ave. Update	Max Live
`filter`	11%	16%	23%
`map`	11%	14%	23%
`reverse`	13%	17%	24%
`minimum`	22%	11%	38%
`sum`	22%	29%	34%
`quicksort`	4%	6%	21%
`quickhull`	20%	30%	91%
`diameter`	17%	23%	67%
Averages	15%	18%	40%

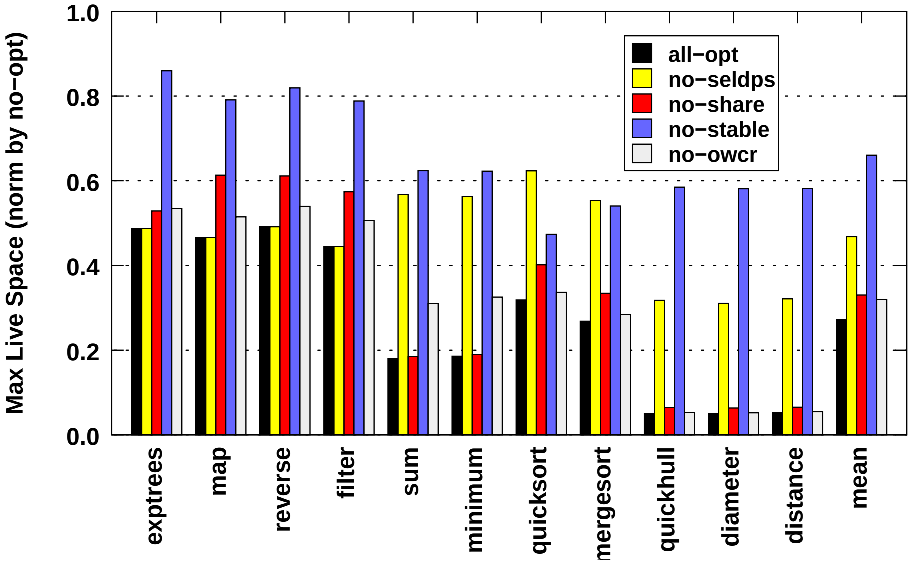
Stage 3, Machine model: Multiple targets

1. **Stable references**	Programmer uses type qualifier
2. **Selective DPS**	Compiler analysis of update points
3. **Write-once memory**	Programmer uses type qualifier
4. **Trace node sharing**	Compiler analysis coalesces traced ops

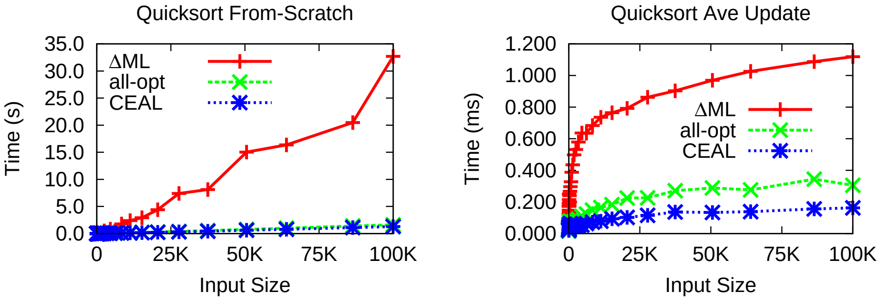
Stage 3, Machine model: Average update times



Stage 3, Machine model: Maximum live space



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	**Programmable**
Abstact machine model	**Sound**
Compiler	Realizes static aspects
Run-time library	Realizes dynamic aspects
Empirical evaluation	**Efficient**