Self-Adjusting Stack Machines

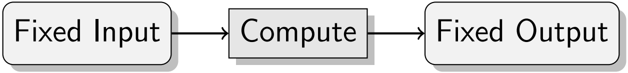
Matthew A. Hammer
Georg Neis Yan Chen Umut A. Acar

Max Planck Institute for Software Systems

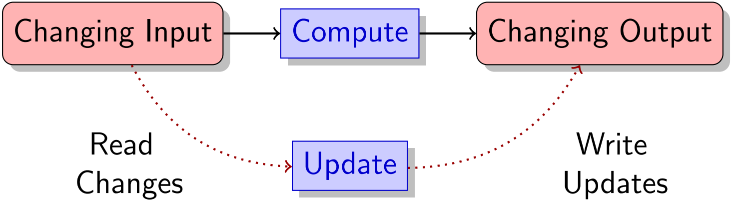
OOPSLA 2011 — October 27, 2011
Portland, Oregon, USA

Static Computation Versus Dynamic Computation

Static Computation:

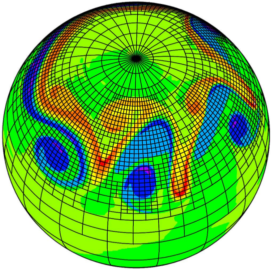


Dynamic Computation:

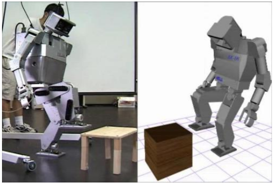


Dynamic Data is Everywhere

Software systems often consume/produce dynamic data



Scientific Simulation



Reactive Systems

Analysis of Internet data

Tractability Requires Dynamic Computations

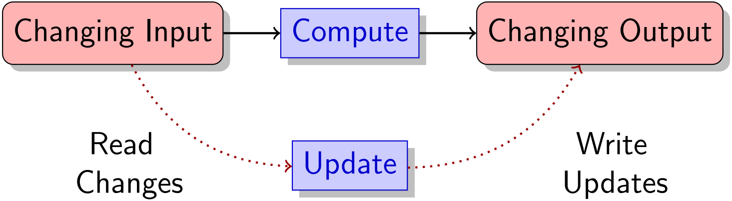
Changing Input → Compute → Changing Output

Static Case

(Re-evaluation "from scratch")

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

Tractability Requires Dynamic Computations



Changing Input

Compute

Changing Output

Read Changes

Update

Write Updates

Static Case

(Re-evaluation "from scratch")

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

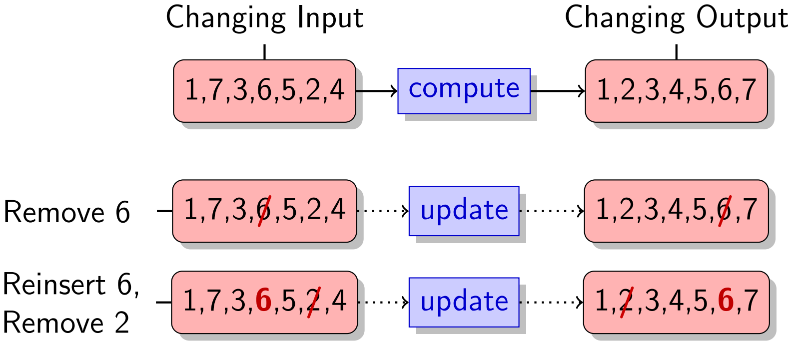
Dynamic Case

(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.



Changing Input

Changing Output

1,7,3,6,5,2,4 → compute → 1,2,3,4,5,6,7

Remove 6 — 1,7,3,~~6~~,5,2,4 ⋯→ update ⋯→ 1,2,3,4,5,~~6~~,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?

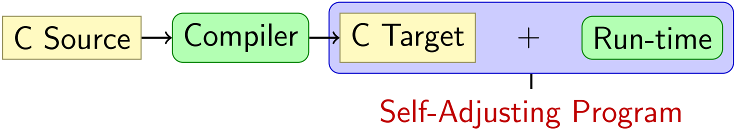
How to Program Dynamic Computations?

Can this programming be systematic?
What are the right abstractions?

1. How to describe dynamic computations?
   - **Usability**: Are these descriptions easy to write?
   - **Generality**: How much can they describe?

2. How to implement these descriptions?
   - **Efficiency**: Are updates faster than re-evaluation?
   - **Consistency**: Do updates provide the correct result?

In **Self-Adjusting Computation**,
Ordinary programs describe dynamic computations.

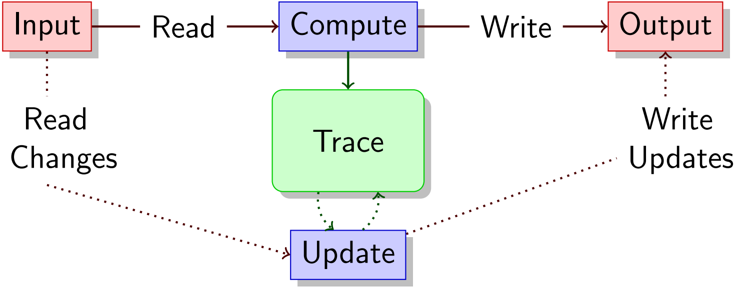

C Source → Compiler → C Target + Run-time

Self-Adjusting Program

The **self-adjusting program**:

1. Computes initial output from initial input
2. Automatically updates output when input changes

Self-Adjusting Programs



- **Self-adjusting program** maintains dynamic dependencies in an execution trace.
- **Key Idea**: **Reusing traces** ⇝ **efficient update**

Challenges

Existing work targets functional languages:

- Library support for SML and Haskell
- DeltaML extends MLton SML compiler

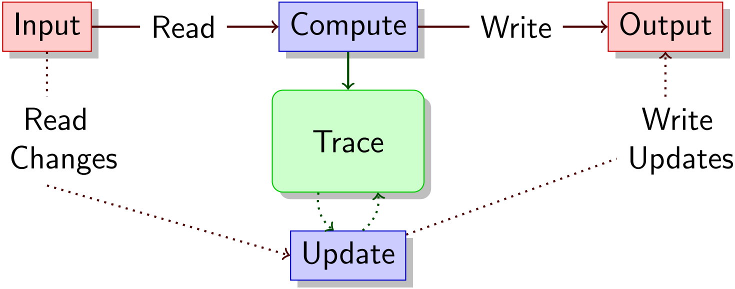
Our work targets low-level languages (e.g., C)

- stack-based
- imperative
- no strong type system
- no automatic memory management

Challenges Low-Level Self-Adj. Computation

Efficient **update** ⇝ complex **resource interactions**:

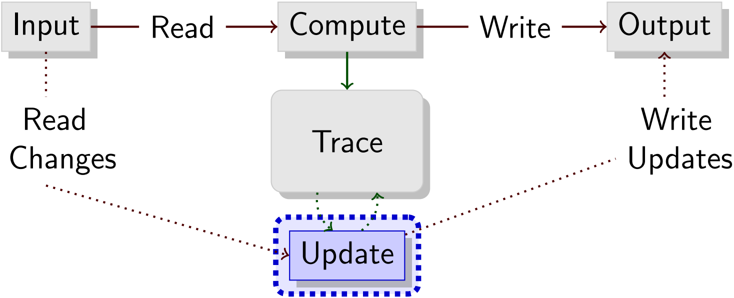
- execution trace, call stack, memory manager



Challenges Low-Level Self-Adj. Computation

Efficient **update** $\rightsquigarrow$ complex **resource interactions**:

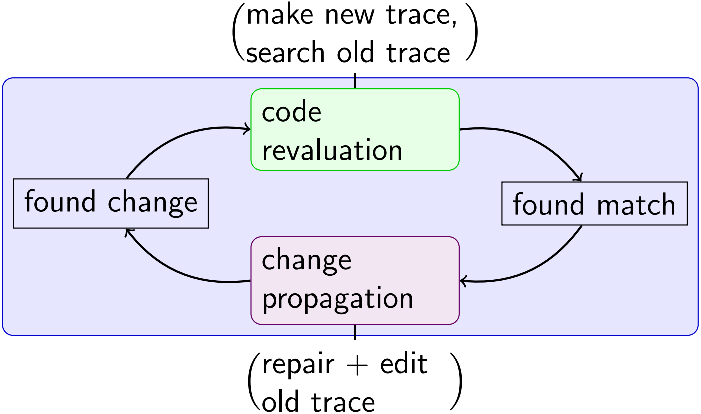
- execution trace, call stack, memory manager



Challenges Low-Level Self-Adj. Computation

Efficient **update** ⇝ complex **resource interactions**:

- execution trace, call stack, memory manager



$\left(\begin{array}{l}\text{make new trace,}\\\text{search old trace}\end{array}\right)$

code revaluation

found match

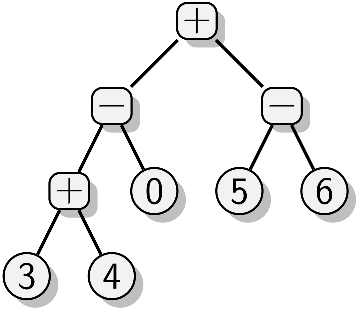
change propagation

found change

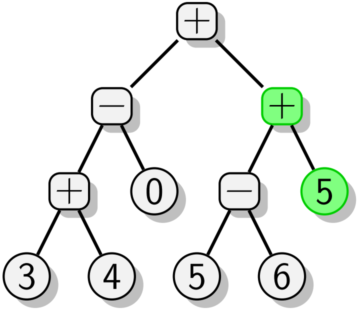
$\left(\begin{array}{l}\text{repair + edit}\\\text{old trace}\end{array}\right)$

Example: Dynamic Expression Trees

Objective: As tree changes, maintain its valuation



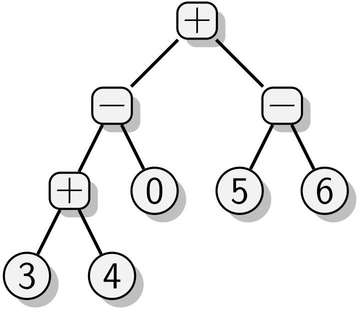
$((3+4)-0)+(5-6)=6$



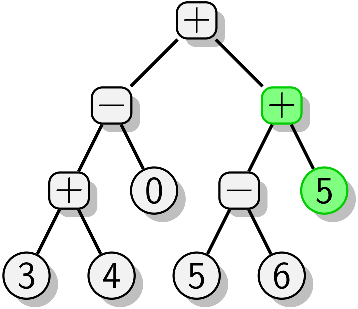
$((3+4)-0)+((5-6)+5)=11$

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 $O(\#\text{affected intermediate results})$

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

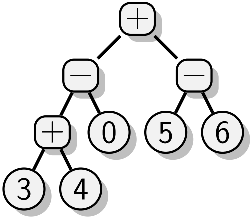
```
int eval (node_t root) {
  if (root->tag == LEAF)
    return root->u.leaf;
  else {
    int l = eval (root->u.binop.left);
    int r = eval (root->u.binop.right);
    if (root->u.binop.op == PLUS) return (l + r);
    else return (l - r);
  } }
```

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

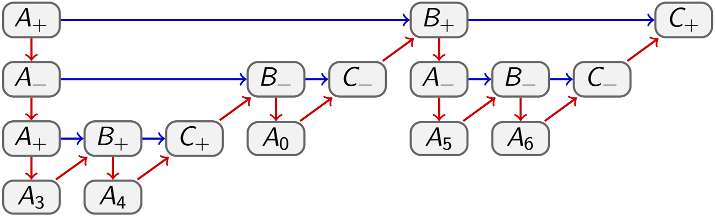
- **Part A**: Return value if `LEAF`
  Otherwise, evaluate `BINOP`, starting with left child
- **Part B**: Evaluate the right child
- **Part C**: Apply `BINOP` to intermediate results; return

Dynamic Execution Traces

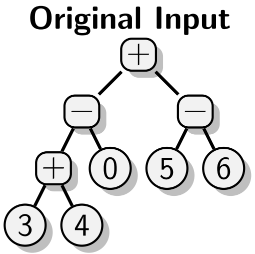
Input Tree

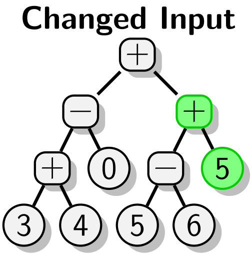


Execution Trace



How to Update the Output?

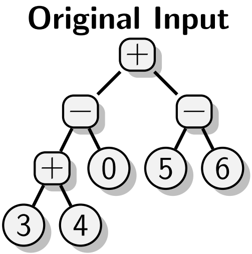


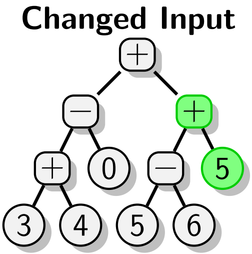


Goals:

- **Consistency**: Respect the (static) program's meaning
- **Efficiency**: Reuse original computation when possible

How to Update the Output?

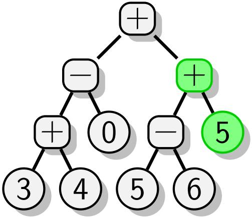


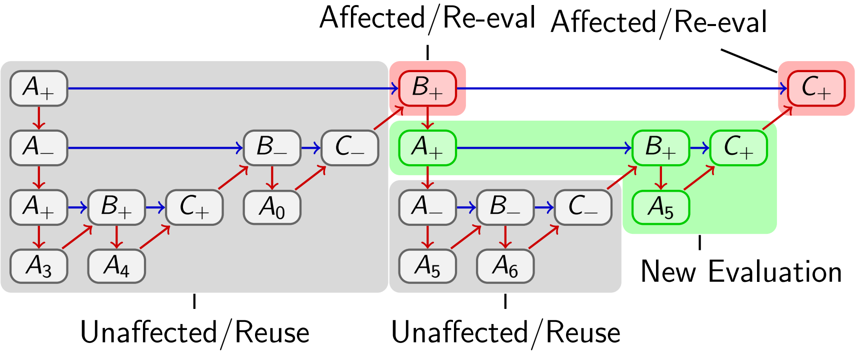


Goals:

- **Consistency**: Respect the (static) program's meaning
- **Efficiency**: Reuse original computation when possible

Idea: Transform the first trace into second trace




Affected/Re-eval

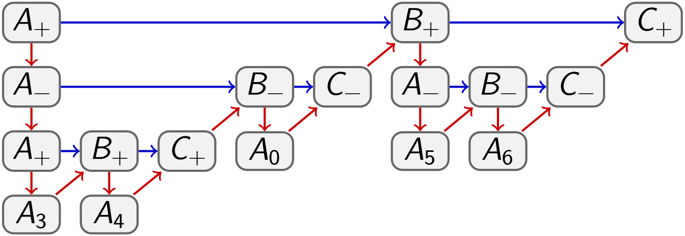
Affected/Re-eval

New Evaluation

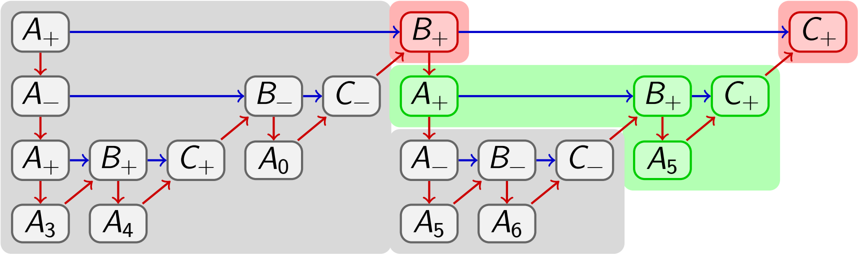
Unaffected/Reuse

Unaffected/Reuse

Before Update



After Update



How to Program Dynamic Computations?

1. How to describe dynamic computations?
    - ✓ **Usability**: Are these descriptions easy to write?
    - ✓ **Generality**: How much can they describe?

2. How to implement this description?
    - ? **Correctness**: Do updates provide the correct result?
    - ? **Efficiency**: Are updates faster than re-evaluation?

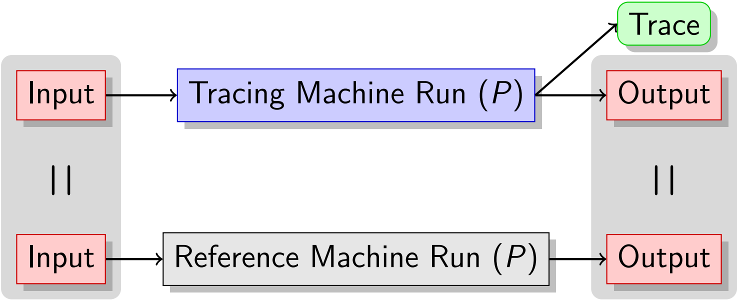
Overview of Formal Semantics

- IL: Intermediate language for C-like programs
- IL has instructions for:
  - Mutable memory: **alloc**, **read**, **write**
  - Managing local state via a stack: **push**, **pop**
  - Saving/restoring local state: **memo**, **update**

Overview of Formal Semantics

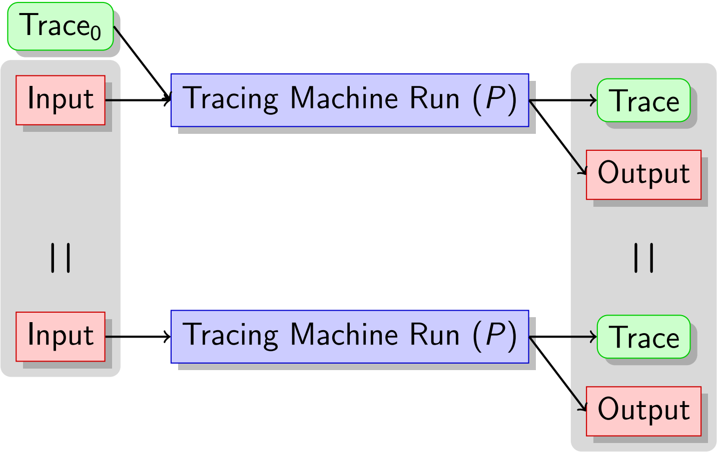
- IL: Intermediate language for C-like programs
- IL has instructions for:
  - Mutable memory: **alloc**, **read**, **write**
  - Managing local state via a stack: **push**, **pop**
  - Saving/restoring local state: **memo**, **update**
- Transition semantics: two abstract **stack machines**:
  - **Reference machine**: defines "normal" semantics
  - **Tracing 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 **stack machines** are **consistent**

Consistency theorem, Part 1: No Reuse



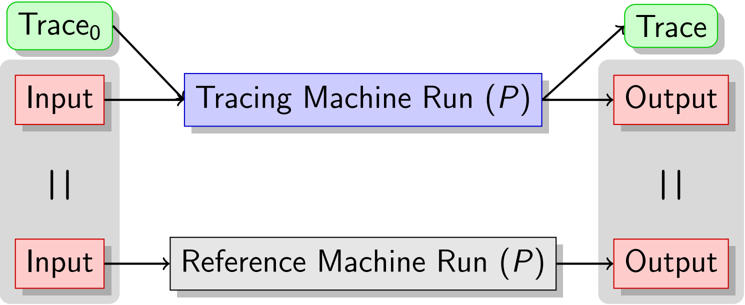
Tracing machine is consistent with reference machine
(when tracing machine runs "from-scratch", with no reuse)

Consistency theorem, Part 2: Reuse vs No Reuse



Tracing 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:

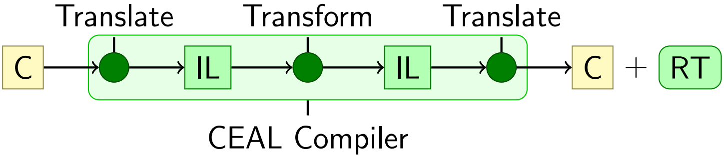
Tracing machine is consistent with reference machine

How to Program Dynamic Computations?

1. How to describe dynamic computations?
   - ✓ **Usability**: Are these descriptions easy to write?
   - ✓ **Generality**: How much can they describe?

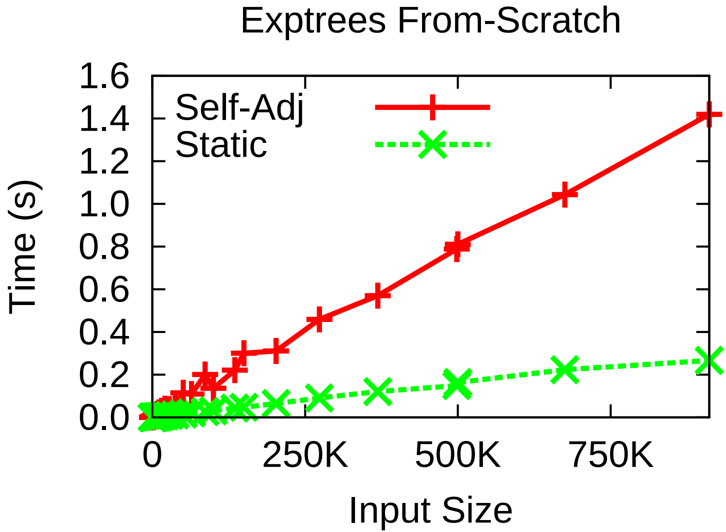
2. How to implement this description?
   - ✓ **Correctness**: Do updates provide the correct result?
   - ? **Efficiency: Are updates faster than re-evaluation?**

Overview of Our Implementation

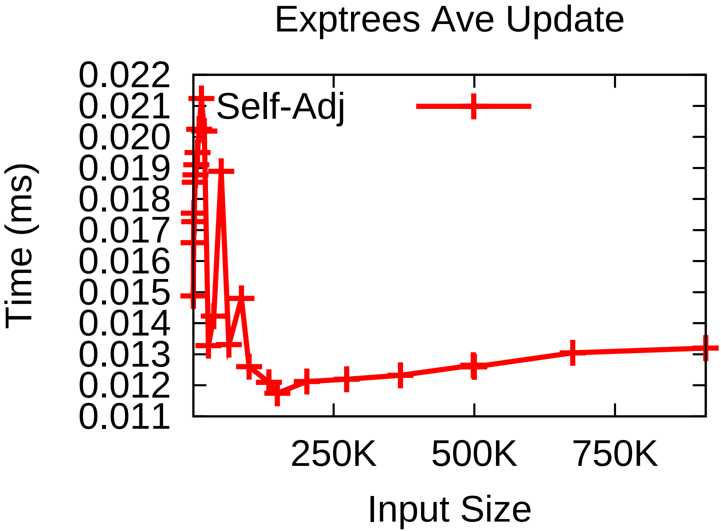


- **Compiler**: produces C targets from C-like source code
- **Run-time**: maintains traces, performs efficient updates

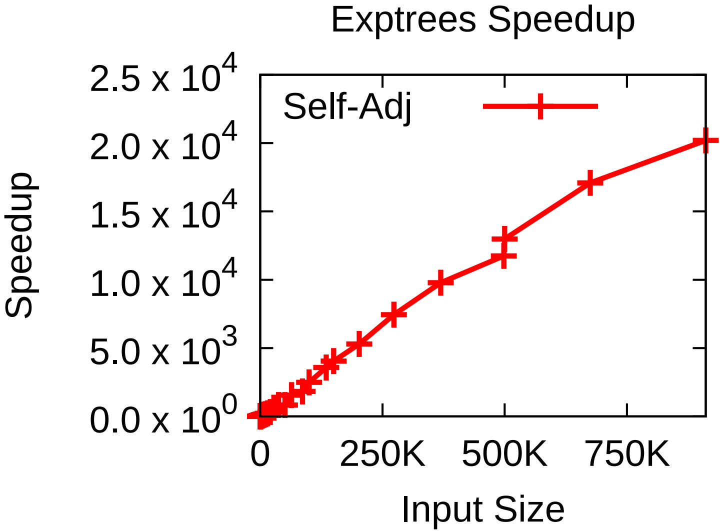
Dynamic Expression Trees: From-Scratch Time



Dynamic Expression Trees: Ave Update Time



Dynamic Expression Trees: Speed up



Summary of Empirical Results

Benchmark	N	Initial Overhead (Compute / Static)	Speed-up (Static / Update)
exptrees	10^6	8.5	1.4×10^4
map	10^6	18.4	3.0×10^4
reverse	10^6	18.4	3.8×10^4
filter	10^6	10.7	4.9×10^4
sum	10^6	9.6	1.5×10^3
minimum	10^6	7.7	1.4×10^4
quicksort	10^5	8.2	6.9×10^2
mergesort	10^5	7.2	7.8×10^2
quickhull	10^5	3.7	2.2×10^3
diameter	10^5	3.4	1.8×10^3
distance	10^5	3.4	7.9×10^2

Our Contributions

A consistent **self-adjusting semantics** for low-level programs

Our Contributions

A consistent **self-adjusting semantics** for low-level programs

Our **abstract machine semantics**

- Describes **trace editing** & **memory management**
  $\rightsquigarrow$ implementation of run-time system
- But requires programs with a **particular structure**
  $\rightsquigarrow$ implementation of compiler transformations

Our Contributions

A consistent **self-adjusting semantics** for low-level programs

Our **abstract machine semantics**

- Describes **trace editing** & **memory management**
  $\rightsquigarrow$ implementation of run-time system
- But requires programs with a **particular structure**
  $\rightsquigarrow$ implementation of compiler transformations

Our **intermediate language** is low-level, yet abstract

- orthogonal annotations for self-adjusting behavior
- no type system needed

$\rightsquigarrow$ implementation of C front end

Thank You! Questions?

Self-adjusting computation is a language-based technique to derive dynamic programs from static programs.

Summary of contributions:

- A self-adjusting semantics for **low-level** programs.
  This semantics defines **self-adjusting stack machines**.
- A compiler and run-time that implement the semantics.
- A front end that embeds much of C.