Efficient Detection of All Pointer and Array Access Errors

Todd M. Austin (presenting)
Scott E. Breach
Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin – Madison

Outline

- Memory Access Errors
- Motivation
- Basic Methodology
- Check Optimization
- Experimental Evaluation
- Summary and Future Work

Memory Access Errors

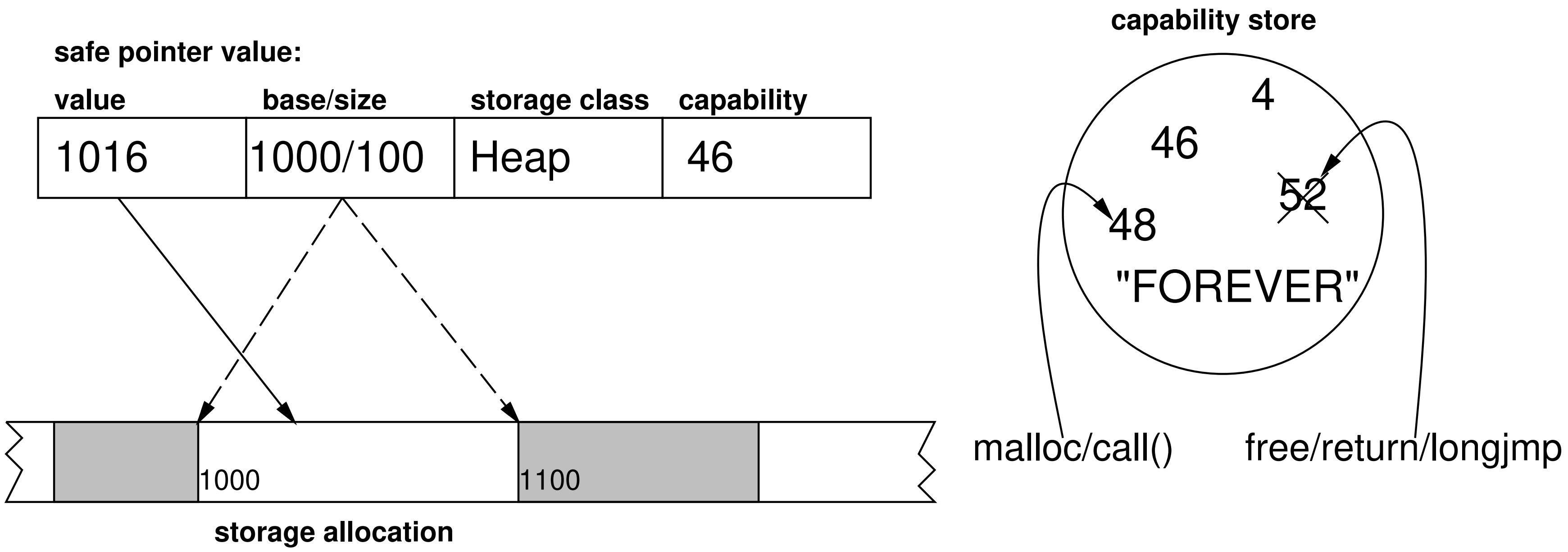
A *memory access error* is any dereference which accesses storage outside of the intended referent.

- *spatial access error*, outside of referent address bounds
- *temporal access error*, outside of referent lifetime
- *intended referent*, the memory object used to create the pointer value (in C, reference created with `malloc()` or '`&`')

Motivation

- programming errors are costly
- aggravated by abstraction, parallel programming, and programming in the large
- memory access errors are difficult to find and fix
  - arise under exceptional conditions
  - difficult to reproduce
  - difficult to correlate fault to error
- therefore: need both efficiency and complete coverage

Safe Pointer Abstraction



Valid Dereference:

$$(capability \in capStore) \wedge (base \leq value \leq base + size - sizeof(*value))$$

Safe Pointer Creation, Manipulation, and Use

malloc(sizeof(struct Foo)) ==>

value	base/size	storage class	capability
2000	2000/32	Heap	52

&p−>next−>status ==>

value	base/size	storage class	capability
4032	4032/4	Local	36

value	base/size	storage class	capability
1010	1000/100	Heap	46

+ 6 ==>

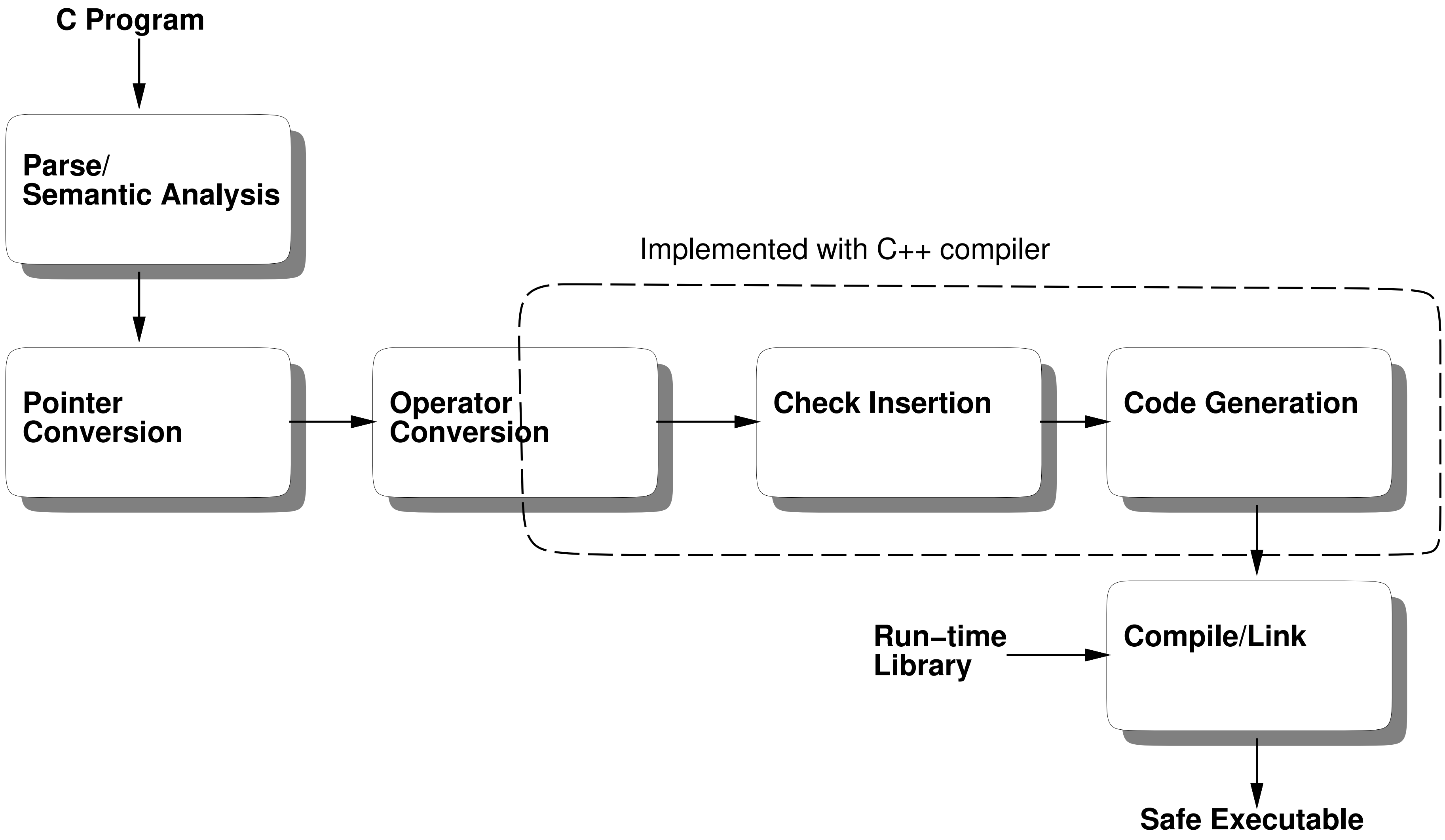
value	base/size	storage class	capability
1016	1000/100	Heap	46

deref

value	base/size	storage class	capability
1010	1000/100	Heap	46

==> **perform access check**

Program Transformations



Requisites of Complete Coverage

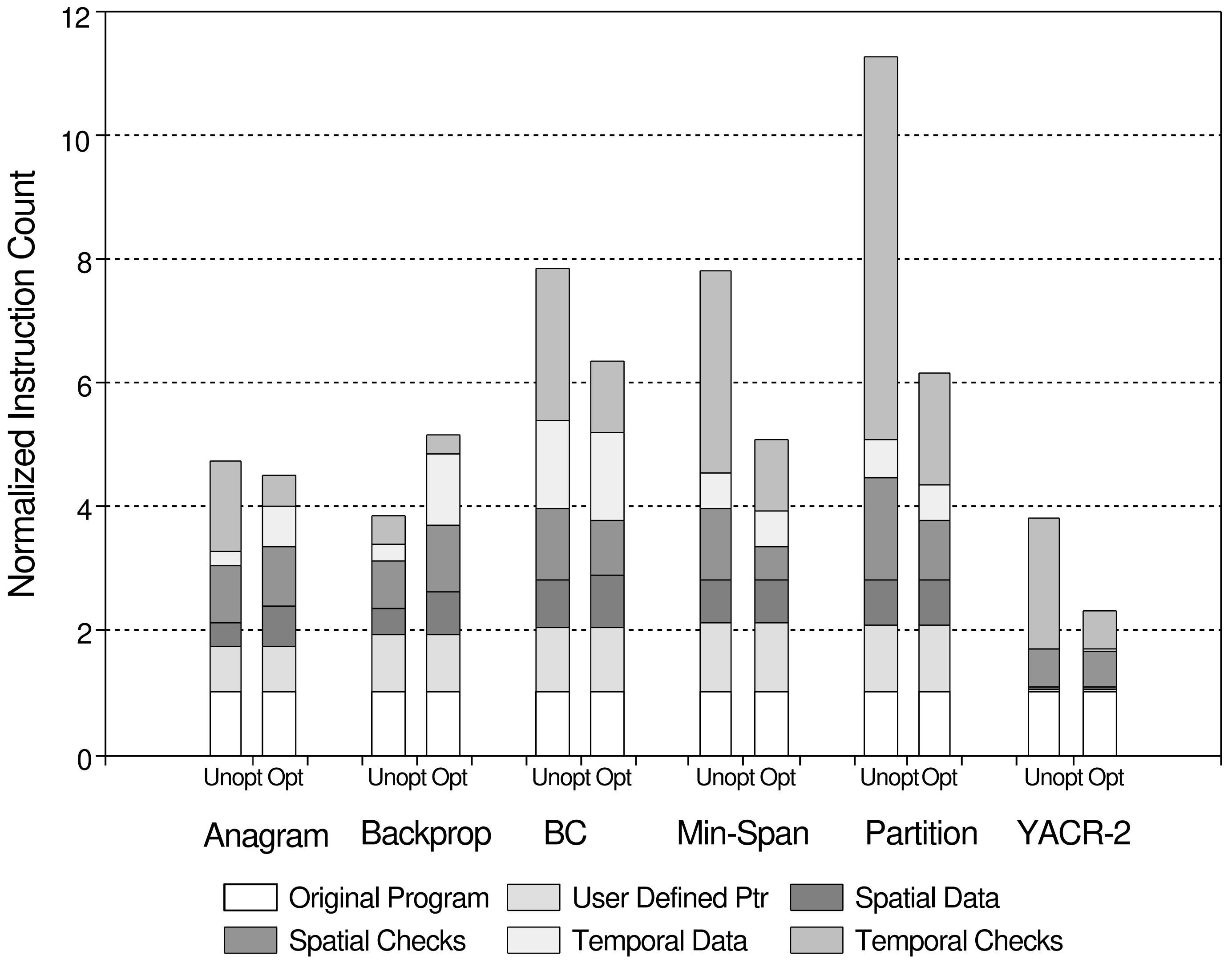
- storage management must be apparent
  - systems programmer can assist with API
- pointer constants must be well defined
  - NULL, functions, strings are ok, others use API
- object attributes must be preserved
  - cannot be manipulated by program
  - cannot be lost
  - “well behaved” programs can be checked efficiently

Check Optimization

A check at a dereference of pointer value v may be elided at program pointer p if the previous, equivalent check executed on v has not been invalidated by some program action.

- run-time check optimization
  - more flexible, but more run-time overhead
  - eliding capability store searches: free counter
  - eliding range checks: memoization
- compile-time check optimization
  - less flexible, no run-time overhead
  - similar to common subexpression elimination

Execution Overheads



Results

- execution overheads are low enough for development environments (130-540%), not low enough for in-field releases
- greatest slowdown factors:
  - check insertion breaks traditional optimizations
  - C++ templates are problematic
  - safe pointers are not register allocated
- run-time optimization of spatial checks is ineffective
- run-time optimization of temporal checks works well
- text and data size overheads are quite low

Comparison to Other Checking Techniques

- How are intended referents tracked?
  - "fat" pointers vs. no tracking
- How is the state of active memory represented?
  - capability store vs. memory state map
- How is the program instrumented?
  - object- vs. source-level
- What optimizations are applied?
  - spatial and temporal check optimization

Summary

- technique capable of detecting **ALL** memory access errors
- source level program transformations
- works on existing codes, *e.g.*, those written in C/C++
- employs safe pointers → pointer value + referent details
- compile- and run-time access check optimization possible
- 540% overhead for *Partition* (3.7 insts/deref)

Current and Future Work

- re-target compiler from C++ to C
- compile-time check optimization
- run-time check improvements
- back-end integration
- parallel checking
- storage leak detection
- interface issues (libraries, system calls, unsafe code)

- availability: e-mail `austin@cs.wisc.edu`

Temporal Access Error Example

	p	q	capability store
`char *p, *q;`	`[x,x,x,x,NEVER]`	`[x,x,x,x,NEVER]`	`{ }`
`p = malloc(10);`	`[2000,2000,10,Heap,1]`	`[x,x,x,x,NEVER]`	`{ 1 }`
`q = p+6;`	"	`[2006,2000,10,Heap,1]`	"
`*q; /* no error */`	"	"	"
`free(p);`	"	"	`{ }`
`p = malloc(10);`	`[2000,2000,10,Heap,2]`	"	`{ 2 }`
`*q; /* error!!! */`	"	"	"

[value, base, size, storageClass, capability]

Analyzed Programs

Program	Instructions		Insts per Dereference		Description
	Static	Dynamic	Static	Dynamic	
Anagram	10.0K	19.4M	106.3	7.6	anagram generator
Backprop	10.8K	122.4M	148.5	8.9	neural net trainer
GNU BC	19.5K	12.2M	15.5	7.6	arbitrary precision calc
Min-Span	11.9K	13.3M	48.7	5.9	min spanning tree comp
Partition	13.5K	21.1M	62.4	3.7	graph partitioning tool
YACR-2	18.5K	546.2M	37.1	14.0	VLSI channel router

Comparison to Other Checking Techniques

- How are intended referents tracked?
  - "fat" pointers (Safe-C, CodeCenter, RTCC, Bcc, UW-Pascal)
  - intended referents are not tracked (Purify)
- How is the state of active memory represented?
  - capability store (Safe-C)
  - memory state map (Purify, CodeCenter)
  - keys stored in heap node headers (UW-Pascal)
  - memory state is not tracked (RTCC, Bcc)
- How is the program instrumented?
  - object-level: (Purify)
  - source-level/compile-time: (Safe-C, RTCC, Bcc, UW-Pascal)
  - run-time: (CodeCenter)