Presentation of: Efficient Detection of All Pointer and Array Access Errors

Todd M. Austin, Scott E. Breach, Gurindar S. Sohi

Presented by Michael Shah
9/18/12
Duration ~ 15 minutes
Motivation

- Programming errors are costly.
- Memory access errors (pointer errors), we already know are troublesome.
- Some software error reports that 50% of all errors are pointer/array access errors!
Papers Main Contributions

- Extended safe pointer representation
- First technique to detect spatial and temporal errors
- Source level transformation: can apply to both compile- and run-time optimization
  - Overall overhead ranges from 130% to 540%
- Some things to keep in mind: Paper was published in 1994, have we done better?
  - (Where do we see "smart pointers" today?)
Memory Access Error Definition

- Two Types:
  a. 1.) Spatial Access Error - Indexing past the end of an array
  b. 2.) Temporal Access Error - Assigning to a heap allocation after it has been freed.

- Memory access errors are hard to detect
  a. Errors may not manifest except under exceptional conditions
  b. Exceptional conditions are hard to reproduce
  c. Even when we do get errors, it's hard to make the correlation between program and memory error!
Figure 1 - Basic Pointer Arithmetic

Danger!

int FindToken(int *data, int count, int token) {
    int i = 0;
    int *p = data; // If we do not find p == token, then we keep looping!
    while ((i < count) && (*p != token)) {
        // This will cause a memory access error or maybe something even more unpredictable!
        p++;
        i++;
    }
    return(*p==token);
}

● But would we find this error?
   a. Will our data always not contain the token?
   b. What if our data is dynamic, will we be able to recreate the exact scenario?
   c. Will we make the connection?
What protection do we have?

- Unix based systems
  - Attempting to store in `.text` section of program terminates execution of program (core.dump)
- Does Windows provide such protection? I'm not sure?
  - Okay, I did look it up, there is some Data Protection and a Data Protection API at the system level
The papers solution to memory access problems!

- "Transform programs at compile-time to use extended pointer representation which we call a safe pointer"
Safe Pointer Representation

typedef {
    // Value of any expressible address
    <type> *value;
    // Address of referent
    <type> *base;
    // Size of referent (base + size allows us to range check!)
    unsigned size;
    // Where we are storing (helps detect erroneous storage deallocations)
    enum {Heap=0, Local, Global} storageClass;
    // Represents all active dynamic storage (plus special capabilities FOREVER and NEVER)
    int capability;
} SafePtr<type>;
Safe Pointer Usage

// value is only attribute accessible from source
<type>*value;

// Base and Size used for Spatial Access Errors
<type>*base;
unsigned size;

// storageClass and capability used for Temporal Access Errors
enum {Heap=0, Local, Global} storageClass;
int capability;
Safe Pointer States

- **unsafe** - dereferencing may cause an undetected memory access error
- **invalid** - Error would occur if we try to dereference a pointer in this state
  - Note this is why error checking is done only when we dereference. It's okay to have invalid pointers if and only if we do not use them!
- **valid** - Pointer is okay to dereference.
Program Transformation to Support Smart Pointers

1. Pointer Conversion
   a. Extend pointers into Safe Pointer Representation
      i. If you generate a new ptr, make it a safe ptr

2. Check Insertion
   a. Detect memory access errors
      i. Check referent's capability
      ii. Perform bounds check (fit data into referent)

3. Operator Conversion
   a. Maintain (or drop) object attributes
      i. Ex: q = p +6 creates new safe pointer for q, where q has p's attributes.
      ii. Ex: Casting to non-pointer type drops object attributes.
      iii. may be direct (refers to name) or indirect (traversal)
Can we cover everything?

1. Storage management must be apparent to translator
   a. Some errors can be missed!

2. Referents of all pointer constants must have known location, size, and lifetime
   a. What about casting non-pointers to pointers?

3. Program must not manipulate the object attributes of any pointer value
   a. Imagine changing storage of a pointer from Global to Heap or some other value! What happens when we free pointer?

If our program is "well-behaved" smart pointers will work quite well however.
Optimization - It must run fast

● Run-Time Check Optimization
  ○ Checks at runtime give us flexibility at cost of performance
  ○ Memoization to speed up spatial checks

● Compile-Time Check Optimization
  ○ Compile-time checks cause no overhead
  ○ 3 phase algorithm
    ■ Allows us to omit redundant temporal error checks
    ■ See Figure 8 for algorithm
Evaluation and Results

- Source-to-Source Translation to apply smart pointer transforms
- Analyzed six programs
  - Execution, Text, and Data overheads
  - Some checking done on external libraries through interlace checking (system library pointer arguments checked)
- Results
  - Still too slow for large scale programs to be used during deployment
  - Need to integrate with compilers for optimizations
  - .text and .data overhead overall is low (~330% each in the worst cases)
Related Work to Solve Memory Access Errors

- Reference Chaining
- Limit programming constructs
- Combined Techniques to solve other memory access errors
Conclusion of Paper

- A set of transformations are provided, and a set of pointer and array access errors are stopped.
Strengths and Weaknesses of technique (slide 1/2)

● **Strengths of technique**
  ○ The transformation overall seems pretty easy (make all pointers safe pointers)
  ○ Good emphasis talking about optimization
    ■ Optimizing dereference checks
    ■ Reducing redundant checks is good
    ■ Running only compile-time or run-time optimizations is an option
Strengths and Weaknesses of technique (slide 2/2)

- **Weaknesses of technique**
  - We blame the compiler for integration issues
    - Apparently some bug exists in frameworks so this is okay.
  - Should we go through all this trouble, or just use FORTRAN?
  - Do we have complete coverage? Object attributes need to be attached to all variables. However, assumption given that our program is "well-behaved"
Strengths and Weaknesses of paper

● Strengths of paper
  ○ Good build up - I think I could implement most of the basic program transformations
  ○ I liked the related work section could've even renamed to "future" work.

● Weaknesses of paper
  ○ Should programmers sacrifice expressiveness for less errors? Authors say no, would others say yes?
  ○ Was text size overhead (up to 300%) enough motivation for me to not use this technique?
  ○ Might have been interesting to see benchmarks used with compiler optimizations, just to see the best case scenario.
Should we accept this paper? (Remember, we're in 1994!)

- **Effectiveness**
  - I would argue most common pointer errors are captured. Overall good coverage

- **Performance**
  - As much as 540% slower, but potentially better on smaller applications

- **Usability**
  - Very useable, process is automated

- **Place in Software Lifecycle**
  - Can be used from the start to deployment (and possibly in deployment)
Congratulations this paper has been ACCEPTED(to PLDI 1994)!
Presentation of: Efficient Detection of All Pointer and Array Access Errors
Todd M. Austin, Scott E. Breach, Gurindar S. Sohi

Presented by Michael Shah
9/18/12
Duration ~ 15 minutes
Discussion Questions

1. Why not just use a language like FORTRAN that is safe? Do you buy the argument that "we felt that it was important not to restrict the expressiveness available to the programmer"

2. The authors technique is not portable across languages, but is portable across different platforms? How and why might this be important?

3. Your company is willing to invest in exclusively using either the presented smart pointer system or valgrind(memcheck tool). Which do you choose and why?

4. How can we speed up this smart pointer implementation?

5. How can this tool be combined with other techniques to solve other errors (hint, look in related-work section).

6. Should smart pointers refer to objects on the heap? What about the stack?

7. Should programmers sacrifice expressiveness for less errors?
Discussion Questions Suggested Answers

1. Certain tasks such as writing an operating system or expressing algorithms is important.
2. Maybe it doesn't matter, maybe everyone just uses C to program, it is 1994 afterall.
3. Smart Pointers for speed and possibility of being able to ship if product is small. Valgrind for detecting.
5. Combined with Conservative collector to detect storage leaks, many other tools listed as well.
6. Heap yes. Stack no, don't point to objects deleted when they go out of scope.
7. Open question.
Glossary and Helpful links

- Pointer -
- Dereference -
- Referent - Object to which a pointer points
- Associative Memory - Allows high speed searching