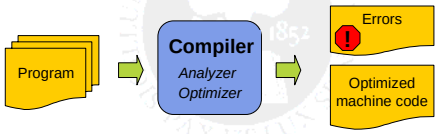
COMP 181

Lecture 20
My research

November 16, 2006

Motivation

- Building quality software is difficult
  - Modern software is complex
  - Need good performance and correctness
- Programmers rely on automation -- compilers



➡ Are compilers good enough?

Find the error – part 1

- Example:

```
switch (var) {
  case 0: ...
    break;
  case 1: ...
    break;
}
case 2: ...
```

- *Error*: **`case`** outside of **`switch`** statement
  - Part of the language definition
  - Reported at compile time
  - Compiler indicates the location and nature of error

Find the error – part 2

- Example:

```
int sock;
char buffer[100];
sock = socket(AF_INET, SOCK_STREAM, 0);
read(sock, buffer, 100);
execl(buffer);
```

- *Error*: executes any command from Internet
  - Security vulnerability
  - What if this code runs as root?

➡ No traditional compiler reports this error
Burden is on the programmer

Problem

- Compilers are too low level
  Fallen behind modern

> ***"Proebsting's Law"**: advances in compiler optimizations double performance every 18 years*

- Performance
  - ***Compilers***: constant folding, loop transforms, CSE
  - ***Opportunities***: domain-specific optimizations, cooperation with run-time systems
- Error checking
  - ***Compilers***: syntax, types, simple semantics
  - ***Problems***: API protocols, security vulnerabilities

My research

- **Solution**:
  Raise the level of abstraction in compilers
- Provide better information
  Higher level, domain-specific semantics
- More powerful analysis
  Collect deeper information about programs
- Cooperate with other system components
  Memory managers

Outline

- Motivation
- The Broadway compiler
  - Recognize and exploit software libraries
  - Scalable analysis for error detection
- Compiler-assisted garbage collection

Security vulnerabilities

Remote access vulnerability

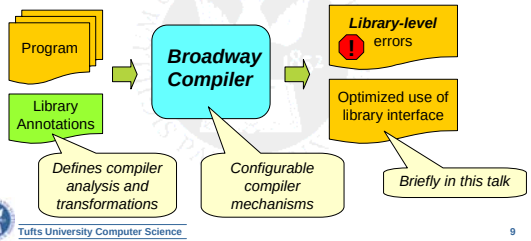
```
int sock;
char buffer[100];
sock = socket(AF_INET, SOCK_STREAM, 0);
read(sock, buffer, 100);
execl(buffer);
```

- Why don't compilers detect this error?
  - Don't know what library functions do
  - Don't know what constitutes an error

⇨ Idea: Tell the compiler about libraries

Broadway

- *Library-level* compiler
  - Leverage domain-specific semantics
  - Analyze and optimize at interface level



Annotation langua

Two kinds of data: "*Trusted*" and "*Untrusted*"

```
property Trust { Trusted, Untrusted }

procedure socket(domain, type, protocol)
{
  analyze Trust {
    if (domain == AF_UNIX) IOHandle <- Trusted
    if (domain == AF_INET) IOHandle <- Untrusted
  }
  on_exit { return --> new IOHandle }
}

procedure read(fd, buf, size)
{
  on_entry { buf --> buffer
             df --> IOHandle }
  analyze Trust {  buffer <- IOHandle }
}

procedure execl(path, args)
{
  on_entry { path --> path_string }
  report if (Trust : path_string could-be Untrusted)
    "Error at " ++ $callsite ++ ": remote access";
}
```

Internet sockets are *untrusted*

If we read from an untrusted socket, the resulting data is *untrusted*

If *untrusted* data is passed to exec, emit an error message

Automatic detection

- Read annotations
- Statically analyze program
  - Following untrusted values
  - See where untrusted data used
  - Emit error messages

⇨ Very difficult problem for compilers
  - Why is it hard?
  - New algorithm

Challenge 1: Pointers

- Example:

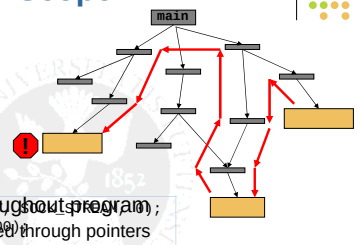
```
int sock;
char buffer[100];
char * ref = buffer;
sock = socket(AF_INET, SOCK_STREAM, 0);
read(sock, buffer, 100);
execl(ref);
```

- Still contains a vulnerability
  - Only one buffer
  - Variables `buffer` and `ref` are *aliases*

⇨ We need pointer analysis

Challenge 2: Scope

- Call graph:
  - Procedure
  - Calls




```
sock = socket(AF_INET,SOCK_PROTO);
read(sock, buffer, 100);
execl(ref);
```

- Objects flow throughout program
  - Objects referenced through pointers
  - No scoping constraints

➡ We need whole-program analysis

Challenge 3: Precision

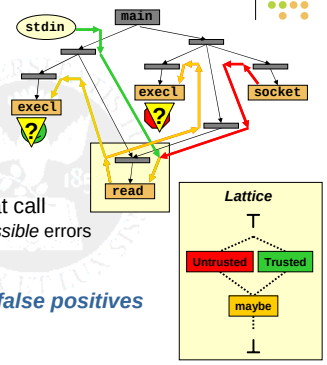
- Static analysis is always an approximation
  Precision: level of detail or sensitivity

Context Sensitivity (Consider each procedure invocation separately?)	Flow Sensitivity (Track state of variables over time?): no	Flow Sensitivity: yes
no	CI-FI	CI-FS
yes	CS-FI	CS-FS

➡ How do we choose?

Low precision

- Example:
  Context-insensitivity

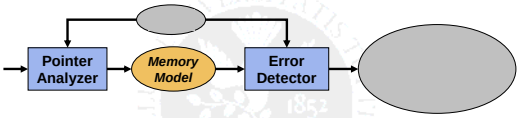


- Information merged at call
  - Analyzer reports 2 *possible* errors
  - Only 1 real error

➡ Imprecision leads to ***false positives***

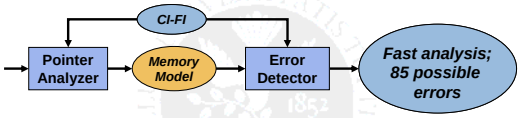
High precision

- Real-life scenario:
  Check for security vulnerabilities in BlackHole mail filter



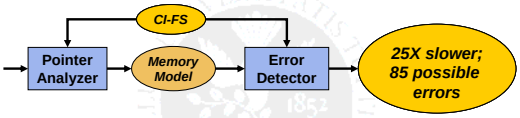
High precision

- Real-life scenario:
  Check for security vulnerabilities in BlackHole mail filter



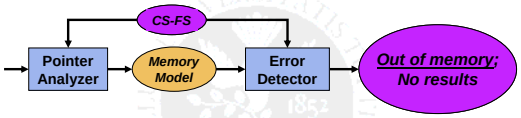
High precision

- Real-life scenario:
  Check for security vulnerabilities in BlackHole mail filter



High precision

- Real-life scenario:
  Check for security vulnerabilities in BlackHole mail filter



- Manually inspect reported errors
  - One thing in common: a string processing routine
  - Clone procedure = ad hoc context sensitivity
  - Using CI-FI, all 85 false positives go away!

➡ *Can we automate this process?*

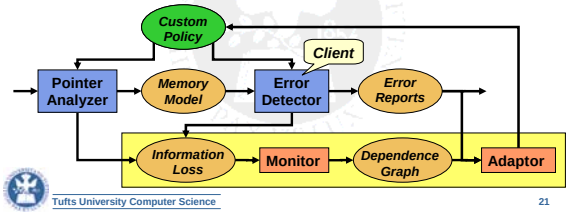
Our solution

- Causes of the problem:
  - Cost-benefit tradeoff – severe for pointer analysis
  - Precision choices are too coarse
  - Choice is made *a priori* by the compiler writer
- **Solution**: Mixed precision analysis
  - Apply higher precision where it's needed
  - Use cheap analysis elsewhere

➡ **Key**: Error checking problem dictates precision
"Client" of the pointer analyzer

Client-driven analysis

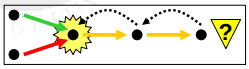
- Algorithm:
  - Start with fast cheap analysis: FI and CI
  - ***Monitor***: how imprecision causes information loss
  - ***Adapt***: reanalyze with a customized precision policy



Algorithm components

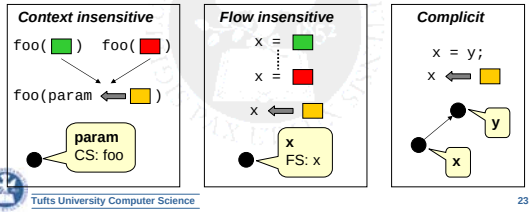
- Monitor
  - Runs alongside main analysis
  - Records imprecision in a graph

- Adaptor
  - Start at the locations of possible errors
  - Trace back to the cause and diagnose

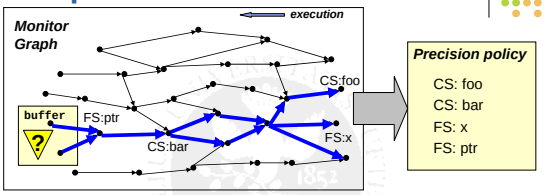


Monitor builds a graph

- How does maybe happen?
  - Polluting assignment
    Add a node for the variable – annotate with a diagnosis
  - Complicit assignment
    Add an edge from left side back to right side



Adaptor

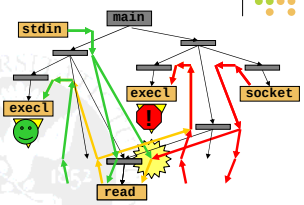


- After analysis...
  - Start at the variables causing "maybe" errors
  - Visit contributing variables – collect the diagnoses

➡ Often a small subset of all imprecision

In action...

- Monitor analysis



- Polluting assignments
- Diagnose and apply "fix"
  In this case: one procedure context-sensitive
- Reanalyze

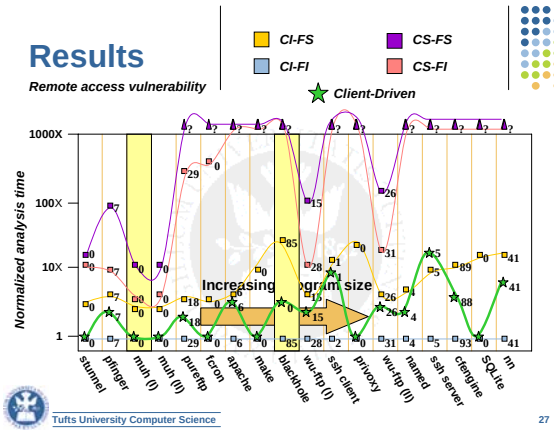
Experiments

- 18 open-source C programs
  Unmodified source – many are system tools
- 5 error checking problems
- Compare client-driven with fixed-precision
  1. Report all real errors – all algorithms are ***sound***
  2. Reduce number of false positives reported
  3. Reduce analysis time

Actual:	True errors	True non-errors		
Reported:	"Error"		False positives	"No error"

Results

Remote access vulnerability



Summary

Library-level compilation with Broadway

- Annotation language [DSL 99]
  Simple language for defining new compiler passes
- Error checking results [SCP 05]
  First thorough evaluation of effect of precision
- Adaptive program analysis [SAS 03]
  Roadmap for scalable analysis
- Optimization experiments [LCPC 00, IEEE 05]
  High-performance computing

Outline

- Motivation
- The Broadway compiler
  - Recognize and exploit libraries
  - Scalable analysis for error detection
- Compiler-assisted garbage collection
  - Customized pointer analysis
  - Cooperation with memory manager

PLDI 2006

Compiler-assisted garbage collection for Java

- Run-time systems
  - Integral part of modern languages
  - High impact on performance
- Garbage collection
  - Significant software engineering benefits
  - *Common complaint*: GC makes programs slow
  - Space/time tradeoff

⇨ Compilers can help
Know about future program behavior

Motivation

- Automatic memory reclamation (GC)
  - No need for explicit "free"
  - Garbage collector reclaims memory
  - Eliminates many programming errors
- **Problem**: when do we get memory back?
  - Frequent GCs:
    Reclaim memory quickly, with high overhead
  - Infrequent GCs:
    Lower overhead, but lots of garbage in memory

Example

```
void parse(InputStream stream) {
  while (not_done) {
    String idName = stream.readToken();
    Identifier id = symbolTable.lookup(idName);
    if (id == null) {
      id = new Identifier(idName);
      symbolTable.add(idName, id);
    }
    computeOn(id);
}}
```

Read a token (new String)

Look up in symbol table

If not there, create new identifier, add to symbol table

Compute on identifier

- ***Notice***: String `idName` is often garbage

  Memory:

Solution

```
void parse(InputStream stream) {
  while (not_done) {
    String idName = stream.readToken();
    Identifier id = symbolTable.lookup(idName)
    if (id == null) {
      id = new Identifier(idName);
      symbolTable.add(idName, id);
    }
    else free(idName);
    computeOn(id);
}}
```

String idName is garbage, free immediately

- Garbage does not accumulate

  Memory:

Our approach

- Adds **`free()`** automatically
  - ***FreeMe*** compiler pass inserts calls to **`free()`**
  - Preserve software engineering benefits
- Can't determine lifetir
  - Works *with* the garbag
  - Implementation of **`fre`**
- **Goal:**
  - Incremental, "eager" memory reclamation
  - ⇨ Results: reduce GC load, improve performance

Potential: 1.7X performance **`malloc`**/**`free`** vs GC in tight heaps
(Hertz & Berger, OOPSLA 2005)

FreeMe Analysis

- **Goal**:
  - Determine *when* an object becomes *unreachable*

Within a method,
*for allocation site "**`p = new A`**"*
*where can we place a call to "**`free(p)`**"?*

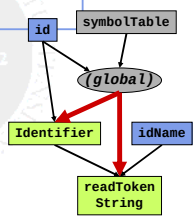
⇨ Not a whole-program analysis

- **Idea**: pointer analysis + liveness
  - Pointer analysis for *reachability*
  - Liveness analysis for *when*

Pointer Analysis

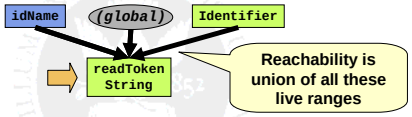
```
String idName = stream.readToken();
Identifier id = symbolTable.lookup(idName);
if (id == null) {
  id = new Identifier(idName);
  symbolTable.add(idName, id);
}
computeOn(id);
```

- Connectivity graph
  - *Variables*
  - *Allocation sites*
  - *Globals (statics)*
- Analysis algorithm
  Flow-insensitive, field-insensitive



Adding liveness

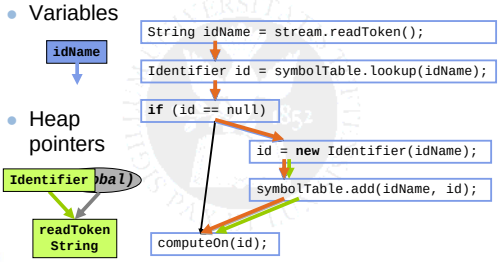
- ***Key***: An object is reachable only when all incoming pointers are live



- From a variable: Live range of the variable
- From a global: Live from the pointer store onward
- From other object: Live from the pointer store until source object becomes unreachable

Liveness Analysis

- Computed as sets of edges
  - Variables
  - Heap pointers

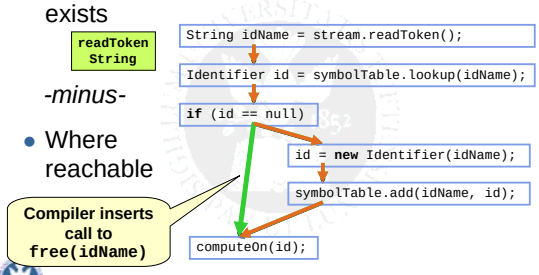


Where can we free it?

- Where object exists

-minus-

- Where reachable

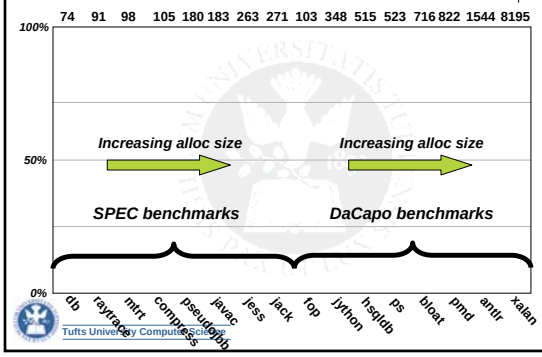


Implementation in JikesRVM

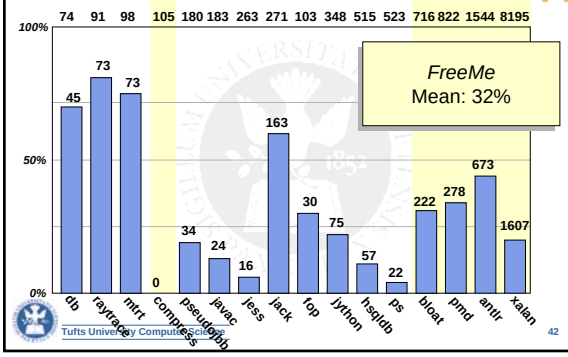
- FreeMe added to OPT compiler
- Run-time: depends on collector
  - Mark/sweep
    Free-list: `free()` operation
  - Generational mark/sweep
    Unbump: move nursery "bump pointer" backward

    Unreserve: reduce copy reserve
    - Very low overhead
    - Run longer without collecting

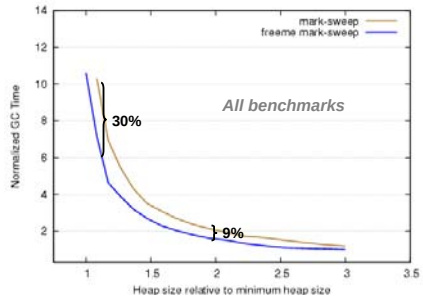
Volume freed – in MB



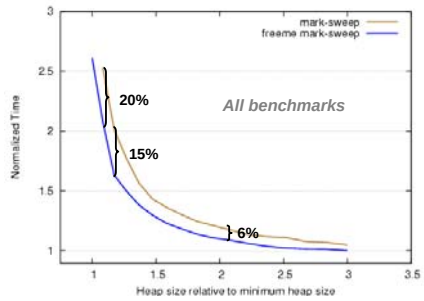
Volume freed – in MB



Mark/sweep – GC time



Mark/sweep – time



Next time

- New project is almost ready
  - Some semantic checking
  - Convert AST to x86 assembly
- More on optimization