Domain-Specific Analysis

Last time
– Context-sensitive pointer analysis

Today
– A break from pointer analysis
– Exploiting domain-specific information in analysis and optimization

Motivation

Two different views of software

Compiler’s view
– Abstractions: numbers, pointers, loops
– Operators: +, −, *, −>, [],

Programmer’s view
– Abstractions: files, matrices, locks, graphics
– Operators: read, factor, lock, draw

→ In high-level languages even language constructs can have this problem.

This discrepancy causes a problem...
Find the Error – Part 1

Example:
```
switch (var_83) {
    case 0: func_24();
    break;
    case 1: func_29();
    break;
    case 2: func_78();
```

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

Find the Error – Part 2

Example:
```
struct __sue_23 * var_72;
char var_81[100];
var_72 = libfunc_84(__str_14, __str_65);
libfunc_44(var_72);
libfunc_38(var_81, 100, 1, var_72);
```

Improper call to libfunc_38
- Syntax is correct – no compiler message
- Fails at run-time

Problem: what does libfunc_38 do?

*This is how compilers view reusables*
Find the Error – Part 3

Example:

```c
FILE * my_file;
char buffer[100];
my_file = fopen("my_data", "r");
fclose(my_file);
fread(buffer, 100, 1, my_file);
```

**Improper call to fread() after fclose()**

⇒ The names reveal the mistake

**No traditional compiler reports this error**

⇒ Run-time system: how does the code fail?

⇒ Code review: rarely this easy to spot

---

Missed Opportunities

**Libraries encapsulate domain-specific semantics**

⇒ This semantic information provides many opportunities for error checking

⇒ This semantic information provides many opportunities for optimization

⇒ This information is unavailable to conventional compilers

**Libraries are domain-specific languages**

⇒ Second-class languages

**The implications are significant . . .**
The Choice: Abstraction vs. Efficiency

Layered Software

Library

Clean

Efficient

Outline

The Problem

Domain-specific optimization
  – PLAPACK library example

Domain-specific program checking
  – Format String Vulnerability example

Results
The Challenge

Software libraries offer many benefits

The challenge facing libraries
- Different clients have different needs
- No single implementation is ideal for all situations

Real World Analogy: The Spork (really a Spofe or something like that)

```
spoon part
fork part
knife part
```

Interface Bloat

Common approach
- Create lots of specialized routines
- Let user choose the appropriate routine
Interface Bloat in MPI

**Short term problems:**
- Complex interface
- Specialized routines can be difficult to use

**12 ways (modes) to perform point-to-point communication:**

<table>
<thead>
<tr>
<th></th>
<th>Normal</th>
<th>Sync</th>
<th>Ready</th>
<th>Buffered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>MPI_Send</td>
<td>MPI_Ssend</td>
<td>MPI_Rsend</td>
<td>MPI_Bsend</td>
</tr>
<tr>
<td>Nonblock</td>
<td>MPI_Isend</td>
<td>MPI_Issend</td>
<td>MPI_Irsend</td>
<td>MPI_Ibsend</td>
</tr>
<tr>
<td>Persistent</td>
<td>MPI_Send_init</td>
<td>MPI_Ssend_init</td>
<td>MPI_Rsend_init</td>
<td>MPI_Bsend_init</td>
</tr>
</tbody>
</table>

March 23, 2015

**Long term problems:**
- No performance portability
- Application becomes less general

March 23, 2015
Problems with Interface Bloat

**Premature optimization**
- Requires manual changes to application source
- Embeds optimizations into application source

**Long Term Problems**
- Complicates maintenance
- Defeats portability

How Can We Avoid Interface Bloat?

**Libraries are domain-specific languages**
- No new syntax
- No compiler support

\[
\begin{align*}
  c &= a \times b; & /* language primitive */ \\
  \text{bnMultiply}(&c,a,b) & /* library call */
\end{align*}
\]

**Libraries encapsulate domain-specific semantics**
- This semantic information provides many opportunities for analysis and optimization
- Make this information available to compilers
Our Solution: The Broadway Compiler [Guyer and Lin’99]

Extends power of compilers to library operations

Application
Library
Annotations: Domain-specific information

Broadway Compiler

Integrated App and Library
Customizes the library for the application

Separation of Concerns

Mortal programmers
Compiler writer
Domain Expert

Application
Library
Annotations: Domain-specific information

Broadway Compiler

Integrated App and Library
Separation of Concerns

The applications programmer does not see the annotations

- Application
- Library
- Broadway Compiler
- Integrated App and Library

Annotations

One compiler for all libraries
One set of annotations per library

Hard parts are reused many times
Hard parts are hidden from the mortals

Outline

The Problem

**Domain-specific optimization**
- PLAPACK library example

**Domain-specific program checking**
- Format String Vulnerability example

Results
Introduction to PLAPACK

PLAPACK
- Parallel Linear Algebra Package
- Developed by van de Geijn, et al. [van de Geijn'97]
- Developed for high performance
- ≈ 40,000 lines of C code

Applications: LU, QR, Cholesky, ...

Typical PLAPACK Application

```
while (True) {
    PLA_Obj_global_length(ABR, &length);
    if (length == 0) break;

    PLA_Obj_split_4(ABR, nb, nb, &A11, &A12, &A21, &A22);

    Cholesky(A11);

    PLA_Trsm(PLA_SIDE_RIGHT, PLA_LOW_TRIAN,
              PLA_TRANS, PLA_NONUNIT_DIAG,
              one, A11, A21);

    PLA_Syrk(PLA_LOW_TRANS, PLA_NO_TRANS,
              minus_one, A21, one, ABR);
}
```

"views" of the data
Views in PLAPACK

The notion of views can be used to perform optimizations

Views can have special properties

These properties can be reasoned about by programmers

These properties can be exploited by using special algorithms

PLA_Trsm (PLA_SIDE_RIGHT, PLA_LOW_TRIAN,
PLA_TRANS, PLA_NONUNIT_DIAG,
one, A11, A21);

PLA_Trsm_local (PLA_SIDE_RIGHT, PLA_LOW_TRIAN,
PLA_TRANS, PLA_NONUNIT_DIAG,
one, A11, A21);

These properties cannot be inferred by conventional compilers

March 23, 2015

View-Based Optimizations

Given the original program

– Compiler analyzes the flow of view information through the program
– Compiler determines when specialized routines can be used

PLA_Obj_view_all(A, &ABR)
while (True) {
    PLA_Obj_length(ABR, &b);
    b = min(b, nb);
    if (b==0) break;
    PLA_Obj_split_4 (ABR,b,b,&A11.. &A21, &ABR);
    Cholesky(A11);
    PLA_Trsm (PLA_SIDE_RIGHT, ...) PLA_Syrk (PLA_LOW_TRIAN, ...)
}

PLA_Trsm() and PLA_Syrk() are overly general– they work for any distribution

March 23, 2015
Specialization Has Many Benefits

If a View is empty

```c
PLA_Obj_view_all(A, &A11)
{ do {
    PLA_Trsm(A11);
    PLA_Obj_len(A11, &len);
    len = min(length, nb);
    PLA_Obj_split_4(A11,len,...);
} while (len>0)
```

- This becomes a no-op
- This becomes dead code
- This loop disappears

The interaction of optimizations produces many benefits

What Information is Needed?

**Define special properties**
- Views can be distributed or local

**Specify how library routines affect these properties**
- Which routines create views, shrink views, etc.?

**Specify when specialized routines can be used**
- How can view information be used to invoke specialized routines?
Library-specific Analysis

Annotations specify analysis problems
- Define a simple flow value – like an enumeration
- The lattice structure and meet function are implied

```plaintext
property Distribution : { Distributed, Local, Panel { Row, Col }}
```

- Define transfer functions for library routines

```plaintext
procedure Copy_matrix(src, dest)
{
    analyze Distribution {
        if (src is-exactly Local) { dest <- Local }
    }
}
```

Specifying Optimizations

Annotations define specializations
- Replace a library call based on analysis results

```plaintext
procedure ParallelMatrixMultiply(A)
{
    when (Distribution : A is-exactly Local)
        replace-with %{ SeqMatrixMultiply($A); }%
}
```

Traditional optimizations extended to libraries
- Constant propagation
- Dead-code elimination
**Dependence Annotations**

**Basic annotations** convey data dependence information
- Defs and uses of procedure parameters
  ```
  modify {};
  access {view};
  ```
- Pointer relationships
  ```
  on_entry {obj->view};
  on_exit {A11->view11, A12->view12,
           A21->view21, A22->view22};
  ```
- These annotations are important for describing complex data structures

---

**How Do We Use This Information?**

The Broadway Compiler has configurable phases
- Each phase draws information from the appropriate annotation
Does It Work?

**Improvement over clean, high quality PLAPACK programs**

PLA_Trsm() and PLA_Gemm() are specialized for their specific calling contexts.

![Graph showing improvement over problem size]

---

A Closer Look at Gemm

- MPI_Send is specialized
- Broadcast is specialized
- The Gemm algorithm is specialized

![Graph showing scalability improvement]

Scalability is improved
Optimizing at Multiple Levels

Levels of Abstraction in PLAPACK

<table>
<thead>
<tr>
<th>Level</th>
<th>Operations</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>Global matrix operations</td>
<td>PLA_Gemm()</td>
</tr>
<tr>
<td>Explicitly parallel</td>
<td>Matrices + high level communication</td>
<td>Rank K algorithm</td>
</tr>
<tr>
<td>Local</td>
<td>MPI + local BLAS</td>
<td>MPI_Send()</td>
</tr>
<tr>
<td>C language</td>
<td>C primitives</td>
<td></td>
</tr>
</tbody>
</table>

There is great benefit to optimizing at multiple levels of abstraction

Ultimate Performance Comparison

Gold standard

Comparison against hand-optimized version written by PLAPACK development team

![Graph showing performance comparison](image)
Optimization Summary

How did we improve performance?

- We specialized the library code for the specific context of the Cholesky application

Traditional compilers cannot exploit these opportunities

- They are unaware of domain-specific semantics

The specification can be simple

- The power is in the compiler mechanisms
- Data-flow analysis is a powerful tool

Outline

The Problem

Domain-specific optimization
- PLAPACK library example

Domain-specific program checking
- Format String Vulnerability example

Results
Error Detection

Error detection is a significant problem
– Code review and testing are tedious and unreliable
– Can the compiler help us?

Problem
– Errors are often domain-specific
– Not errors in the base programming language

Solution
– Use Broadway configurable analysis capabilities

Motivating Example

Format string vulnerability
– Well-known error – many CERT advisories
– Improper use of printf() family
– Example:
  
  ```c
  fgets(buffer, size, file);
  printf(buffer);
  ```
– What if the buffer contains “%s”? 
– What if the buffer is passed to sprintf()?

General solution
– Taintedness analysis
– Data from untrusted sources is “tainted”
– Tainted data may not end up in format string

How do we track this property?
Previous Solutions– Type Qualifiers [Shankar, et al ’01]

**Idea**
- Add tainted and untainted types to library function signatures
  
  ```c
  fgets(tainted char *buffer, int size, FILE *f);
  printf(untainted char *format, . . .);
  ```

- Use type constraint solver to find errors
  - Errors are type mismatches

**Issues**
- What is the type of `strdup()`?
- What happens when the value of strings change?

Results

**Type qualifier approach**
- Requires manual intervention to identify context-sensitivity

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines of C</th>
<th>Procedures</th>
<th>Known Errors</th>
<th>Errors Found</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>bftpd</td>
<td>1,017</td>
<td>180</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>muh</td>
<td>5,002</td>
<td>228</td>
<td>1</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>cfengine</td>
<td>45,102</td>
<td>700</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

**Problem**
- Type-based constraints are not context-sensitive
- Even a few false positives can be a problem
The Broadway Solution

Idea
– Track the taintedness of strings using dataflow analysis problem

Modeling format string vulnerabilities
– Define a taintedness lattice
– Determine the objects that carry the property
– Describe how library routines affect the property
– Identify the error conditions

Taintedness Lattice

<table>
<thead>
<tr>
<th>property Taint : { Tainted { Untainted } }</th>
</tr>
</thead>
<tbody>
<tr>
<td>Untainted</td>
</tr>
<tr>
<td>Tainted</td>
</tr>
<tr>
<td>(⊥)</td>
</tr>
</tbody>
</table>

Any external input is tainted
– User input: scanf(), gets()
– External values: getenv()
– File input: read(), fscanf(), readdir()

Note:
– Taintedness is a property of the buffer, not the surface variable
Transmitting Taintedness

String manipulation can transmit taintedness
- Examples: `strdup()`, `strcpy()`, `strcat()`, `sprintf()`

```c
procedure strcpy(dest, src)
{
  on_entry { src \rightarrow src_string
              dest \rightarrow dest_string }
  access { src_string }
  modify { dest_string }

  analyze Taint {
    if (src_string is-exactly Tainted)
      dest_string <- Tainted
    if (src_string is-exactly Untainted)
      dest_string <- Untainted
  }
}
```

Reporting Errors

Test the flow values
- Tainted strings are not allowed to be format strings
  - Examples: `printf()` family, `syslog()`

```c
procedure syslog(priority, format_ptr, args)
{
  on_entry { format_ptr \rightarrow format
              args \rightarrow arg_contents }
  access { format, arg_contents }

  report if (Taint : format is-exactly Tainted)
    "Error at " ++ @context ++ ": Argument " ++ [ format_ptr ] ++ " is tainted.\n"
}
```

Report the exact location of the problem
- The @context token gives the full call stack
- Brackets are special tokens that represent the name of the actual argument
Program Checking with Broadway

**Benchmarks**

– Actual programs that were distributed with the bug

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines of C</th>
<th>Procedures</th>
<th>Analysis Time (min:sec)</th>
<th>Known Errors</th>
<th>Errors Found</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>bftpd</td>
<td>1,017</td>
<td>180</td>
<td>0:01</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>muh</td>
<td>5,002</td>
<td>228</td>
<td>0:06</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>cfengine</td>
<td>45,102</td>
<td>700</td>
<td>6:38</td>
<td>6</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>named</td>
<td>25,820</td>
<td>444</td>
<td>1:11</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lpd</td>
<td>38,174</td>
<td>726</td>
<td>23:57</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Run on 2Ghz Pentium 4 with 512 MB RAM

March 23, 2015

Open Questions

**How else can domain-specific information be useful?**

– Scheduling and resource management optimizations
– Algebraic properties of operations
– Optimization of sequences of operations
– Machine-specific customization

**Can we apply these ideas to object-oriented languages?**

– Higher cost of encapsulation
– Extensible code is an issue

**Can we apply these ideas to dynamic optimizations?**
Conclusions: Compiler Perspective

**Compilers typically do better with more information**
- Increased scope
  - Peephole $\rightarrow$ Local $\rightarrow$ Global $\rightarrow$ Interprocedural analysis
- Dynamic information
  - Profiling
  - Dynamic feedback
- We’ve introduced a third, orthogonal dimension
  - Domain-specific information

Conclusions: Software Design

**Generality vs. Performance Tradeoff**
- Weak assumptions $\rightarrow$ generality, code reuse
- Strong assumptions $\rightarrow$ good performance

**Breaking the tradeoff**
- Programmers should create software that is general
- Let compilers specialize the software for specific contexts
Conclusions: Software Quality

**Improving software quality**
- To improve the quality software, raise the level of programming abstraction
- To help programmers reason at high levels, provide tools to reason at these high levels
- This idea applies to compilers, debuggers, performance analysis tools, *etc.*

Next Time

**Lecture**
- Adaptive pointer analysis

**Projects**
- Pre-proposals due tonight