Static and Dynamic Program Analysis: Synergies and Applications

Mayur Naik
Intel Labs, Berkeley

CS 243, Stanford University
March 9, 2011
Today’s Computing Platforms

Trends:
- parallel
- cloud
- mobile

Traits:
- numerous
- diverse
- distributed

Unprecedented software engineering challenges in reliability, productivity, scalability, energy-efficiency
A Challenge in Mobile Computing

Rich apps are hindered by resource-constrained mobile devices (battery, CPU, memory, ...)

How can we seamlessly partition mobile apps and offload compute-intensive parts to the cloud?
A Challenge in Cloud Computing

How can we automatically predict performance metrics of programs?

- service level agreements
- data locality
- energy efficiency
- scheduling
A Challenge in Parallel Computing

“How can we automatically make concurrent programs more reliable?”

“Most Java programs are so rife with concurrency bugs that they work only by accident.”

– Brian Goetz

*Java Concurrency in Practice*
Terminology

- **Program Analysis**
  - Discovering facts about programs

- **Dynamic Analysis**
  - Program analysis using program executions

- **Static Analysis**
  - Program analysis without running programs
This Talk

Synergistically combine diverse techniques to solve modern software engineering challenges

Techniques
- static analysis
- dynamic analysis
- machine learning

Challenges
- program scalability
- program reliability
- program performance
Our Result: Mobile Computing

Seamless Program Partitioning:
Upto 20X decrease in energy used on phone

Techniques
- static analysis
- dynamic analysis
- machine learning

Challenges
- program scalability
- program reliability
- program performance
Our Result: Cloud Computing

Automatic Performance Prediction:
Prediction error < 7% at < 6% program runtime cost

Techniques

- static analysis
- dynamic analysis
- machine learning

Challenges

- program scalability
- program reliability
- program performance
Our Result: Parallel Computing

Scalable Program Verification:
400 concurrency bugs in 1.5 MLOC Java programs

Techniques
- static analysis
- dynamic analysis
- machine learning

Challenges
- program scalability
- program reliability
- program performance
Talk Outline

• Overview

• Seamless Program Partitioning

• Automatic Performance Prediction

• Scalable Program Verification

• Future Directions
How do we automatically find which function(s) to migrate?

Offline Optimization using ILP

**Static analysis yields constraints**
- dictates correct solutions
- uses program’s call graph to avoid nested migration

**Dynamic analysis yields objective function**
- dictates optimal solutions
- uses program’s profiles to minimize time or energy

Program Partitioning: CloneCloud [EuroSys’11]
CloneCloud on Face Detection App

- phone = HTC G1 running Dalvik VM on Android OS
- cloud = desktop running Android x86 VM on Linux

- Upto 20X decrease in energy used on phone
- Similar for total running time, and other apps
Talk Outline

• Overview

• Seamless Program Partitioning

• Automatic Performance Prediction

• Scalable Program Verification

• Future Directions
From Offline to Online Program Partitioning

- CloneCloud uses same partitioning regardless of input.

- But different partitionings optimal for different inputs:
  - 1 image: phone-only optimal
  - 100 images: (phone + cloud) optimal

- Challenge: automatically predict running time of a function on an input:
  - can be used to decide online whether or not to partition
  - but also has many other applications
The Problem: Predicting Program Performance

**Inputs:** Program P

```java
class Bldg {
    List events, floors;
    main() {
        Bldg b = new Bldg();
        for (...) {
            BP v = b.events.get(i);
            int t = v.time;
        }
    }
    Bldg() {
        floors = new List();
        for (...) {
            floors.add(new Floor());
        }
        for (...) {
            Elev e = new Elev(floors);
            e.start();
        }
        events = new List();
        for (...) {
            events.add(new BP(...));
        }
    }
}
```

**Input I**

```
java Elev /foo/input.txt
```

| 9 2 1 0 6 |
| 2 1 5 |
| 5 4 8 |
| 7 7 0 |

**Output:** Estimated running time of P(I)

**Goals:**
1. Accurate
2. Efficient
3. General-purpose
4. Automatic
Our Solution: Mantis [NIPS’10]

program P

Offline

performance model

Online

input I

training inputs $I_1, \ldots, I_N$

estimated running time of $P(I)$
Offline Stage of Mantis

- Instrument program with broad classes of features
- Collect feature values and time on training data
- Express time as function of few features using sparse regression
- Obtain evaluators for features using static slicing analysis

```java
class Bldg {
    List events, floors;
    main() {
        Bldg b = new Bldg();
        for (...) {
            BP v = b.events.get(i);
            int t = v.time;
            f4 += t;
        }
    }
    Bldg() {
        floors = new List();
        for (...) f1++;
        floors.add(new Floor());
        for (...) f2++;
        Elev e = new Elev(floors);
        e.start();
        events = new List();
        for (...) f3++;
        events.add(new BP(...));
    }
}

R = .3 + .5 f4 + .8 f4^2
```

<table>
<thead>
<tr>
<th></th>
<th>f1</th>
<th>...</th>
<th>fM</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Static Slicing Analysis

- static-slice(\(v\)) = \{ all actions that may affect value of \(v\) \}

- Computed using data and control dependencies
static-slice($f_4$) =

class Bldg {
    List events, floors;
    main() {
        Bldg b = new Bldg();
        for (...) {
            BP v = b.events.get(i);
            int t = v.time;
            $f_4 += t$;
        }
    }
    Bldg() {
        floors = new List();
        for (...) {
            floors.add(new Floor());
            Elev e = new Elev(floors);
            e.start();
            events = new List();
            for (...) {
                events.add(new BP(...));
            }
        }
    }
}
Offline Stage of Mantis

```java
class Bldg {
    List events, floors;
    main() {
        Bldg b = new Bldg();
        for (...) {
            BP v = b.events.get(i);
            int t = v.time;
            f4 += t;
        }
    }
    Bldg() {
        floors = new List();
        for (...) f1++;
        floors.add(new Floor());
        for (...) f2++;
        Elev e = new Elev(floors);
        e.start();
        events = new List();
        for (...) f3++;
        events.add(new BP(...));
    }
}
```

- Instrument program with broad classes of features
- Collect feature values and time on training data
- Express time as function of few features using sparse regression
- Obtain evaluators for features using static slicing analysis
- If feature is costly, discard it and repeat process

\[
R = 0.3 + 0.5 f_4 + 0.8 f_4^2
\]
Offline Stage of Mantis

- Instrument program with broad classes of features
- Collect feature values and time on training data
- Express time as function of few features using sparse regression
- Obtain evaluators for features using static slicing analysis
- If feature is costly, discard it and repeat process

**Dynamic Analysis**

- Training data
- Profile data
- Rejected features
- Static analysis
- Chosen features

**Static Analysis**

- Machine learning
Mantis on Apache Lucene

• Popular open-source indexing and search engine

• Datasets used: Shakespeare and King James Bible
  – 1000 inputs, 100 for training

• Feature counts:
  – instrumented = 6,900
  – considered = 410 (6%)
  – chosen = 2

• Similar results for other apps

prediction error = 4.8%

running time ratio = 28X (program/slices)
Talk Outline

- Overview
- Seamless Program Partitioning
- Automatic Performance Prediction
- Scalable Program Verification
- Future Directions
Static Analysis of Concurrent Programs

Data or control flow of one thread can be affected by actions of other threads!

```java
class Bldg {
    List events, floors;
    main() {
        Bldg b = new Bldg();
        for (...) {
            BP v = b.events.get(i);
            int t = v.time;
            f4 += t;
        }
    }
    Bldg() {
        floors = new List();
        for (...) {
            floors.add(new Floor());
            for (...) {
                Elev e = new Elev(floors);
                e.start();
            }
        }
        events = new List();
        for (...) {
            events.add(new BP(...));
        }
    }
}
```

Either prove these actions thread-local ...

... or include these actions in the slice as well
The Thread-Escape Problem

- thread-local(p,v): Is v reachable from single thread at p on all inputs?

```java
class Bldg {
    List events, floors;
    main() {
        Bldg b = new Bldg();
        for (...) {
            BP v = b.events.get(i);
            int t = v.time;
        }
    }
    Bldg() {
        floors = new List();
        for (...) {
            floors.add(new Floor());
        }
        for (...) {
            Elev e = new Elev(floors);
            e.start();
            events = new List();
            for (...) {
                events.add(new BP(...));
            }
        }
    }
}
```

Diagram:
- Nodes represent objects.
- Edges represent object relationships.
- Blue nodes are local.
- Red nodes are shared.
- A program state at p is shown with object relationships and object states.

26
The Thread-Escape Problem

• thread-local($p, v$): Is $v$ reachable from single thread at $p$ on all inputs?

• Needs to reason about all inputs ⇒ Use static analysis
The Need for Program Abstractions

• All static analyses need abstraction
  – represent sets of concrete entities as abstract entities

• Why?
  – Cannot reason directly about infinite concrete entities
  – For scalability

• Our static analysis:
  – How are pointer locations abstracted?
  – How is control flow abstracted?
Example: Trivial Pointer Abstraction

class Bldg {
    List events, floors;
    main() {
        Bldg b = new Bldg();
        for (...) {
            BP v = b.events.get(i);
            int t = v.time;
        }
        Bldg() {
            floors = new List();
            for (...) {
                floors.add(new Floor());
            }
            Elev e = new Elev(floors);
            e.start();
            events = new List();
            for (...) {
                events.add(new BP(...));
            }
        }
    }
}

class List {
    List() {
        this.elems = new Object[...];
    }
}
Example: Allocation Sites Pointer Abstraction

class Bldg {
    List events, floors;
    main() {
        Bldg b = new Bldg();
        for (...) {
            BP v = b.events.get(i);
        }
        int t = v.time;
    }
    Bldg() {
        floors = new List();
        for (...) {
            floors.add(new Floor());
        }
        Elev e = new Elev(floors);
        e.start();
        events = new List();
        for (...) {
            events.add(new BP(...));
        }
    }
}

class List {
    List() {
        this.elems = new Object[...];
    }
}
Example: k-CFA Pointer Abstraction

```java
class Bldg {
    List events, floors;
    main() {
        Bldg b = new Bldg();
        for (...) {
            BP v = b.events.get(i);
            int t = v.time;
        }
    }
    Bldg() {
        floors = new List();
        for (...) {
            floors.add(new Floor());
        }
        Elev e = new Elev(floors);
        e.start();
        events = new List();
        for (...) {
            events.add(new BP(...));
        }
    }
}

class List {
    List() {
        this.elems = new Object[...];
    }
}
```
Complexity of Static Analysis

<table>
<thead>
<tr>
<th>pointer abstraction</th>
<th>max abstract values (N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>trivial</td>
<td>1</td>
</tr>
<tr>
<td>allocation sites</td>
<td>H</td>
</tr>
<tr>
<td>k-CFA</td>
<td>H \cdot I^k</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>control-flow abstraction</th>
<th>max abstract states</th>
</tr>
</thead>
<tbody>
<tr>
<td>flow and context insensitive</td>
<td>1</td>
</tr>
<tr>
<td>flow sensitive context insensitive</td>
<td>L</td>
</tr>
<tr>
<td>flow and context sensitive</td>
<td>L \cdot 2^{(N^2 \cdot F)}</td>
</tr>
</tbody>
</table>

H = allocation sites, I = call sites
L = program points, F = fields

Challenge: an abstraction that is both precise and scalable

Our Static Analysis:

<table>
<thead>
<tr>
<th>2-partition</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>flow and context sensitive</td>
<td>Q \cdot L \cdot 4^F</td>
</tr>
</tbody>
</table>

Q = queries
Drawback of Existing Static Analyses

- Different queries require different parts of the program to be abstracted precisely
- But existing analyses use the same abstraction to prove all queries simultaneously

⇒ existing analyses sacrifice precision and/or scalability
Insight 1: Client-Driven Static Analysis

• Query-driven: allows using separate abstractions for proving different queries

• Parametrized: parameter dictates how much precision to use for each program part for a given query
Example: Client-Driven Static Analysis

class Bldg {
    List events, floors;
    main() {
        Bldg b = new Bldg();
        for (...) {
            BP v = b.events.get(i);
            int t = v.time;
        }
    }
    Bldg() {
        floors = new List();
        for (...) {
            floors.add(new Floor());
        }
        Elev e = new Elev(floors);
        e.start();
        events = new List();
        for (...) {
            events.add(new BP(...));
        }
    }
}

class List {
    List() {
        this.elems = new Object[...];
    }
}
Insight 2: Leveraging Dynamic Dynamic Analysis

- Challenge: Efficiently find cheap parameter to prove query
  - $2^H$ choices, most choices imprecise or unscalable

- Our solution: Use dynamic analysis
  - Parameter is inferred efficiently (linear in $H$)
  - It can fail to prove query, but it is precise in practice and no cheaper parameter can prove query

```
inputs
I_1 \ldots I_n

\text{dynamic analysis}

Q
P

\text{static analysis}

abstraction \text{A}

H

P \vdash Q?
```
Example: Leveraging Dynamic Analysis

class Bldg {
    List events, floors;
    main() {
        Bldg b = new Bldg();
        for (...) {
            BP v = b.events.get(i);
            int t = v.time;
        }
    }
    Bldg() {
        floors = new List();
        for (...) {
            floors.add(new Floor());
        }
        Elev e = new Elev(floors);
        e.start();
        events = new List();
        for (...) {
            events.add(new BP(...));
        }
    }
}

class List {
    List() {
        this.elems = new Object[...];
    }
}
## Benchmark Characteristics

<table>
<thead>
<tr>
<th></th>
<th>classes (x 1000)</th>
<th>methods (x 1000)</th>
<th>bytecodes (x 1000)</th>
<th>allocation sites (x 1000)</th>
<th>queries (x 1000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hced</td>
<td>309</td>
<td>1.9</td>
<td>151</td>
<td>1.9</td>
<td>0.6</td>
</tr>
<tr>
<td>weblech</td>
<td>532</td>
<td>3.1</td>
<td>230</td>
<td>3.0</td>
<td>0.7</td>
</tr>
<tr>
<td>lusearch</td>
<td>611</td>
<td>3.8</td>
<td>267</td>
<td>3.5</td>
<td>7.2</td>
</tr>
<tr>
<td>hsqldb</td>
<td>771</td>
<td>6.4</td>
<td>472</td>
<td>5.1</td>
<td>14.4</td>
</tr>
<tr>
<td>avrora</td>
<td>1498</td>
<td>5.9</td>
<td>312</td>
<td>5.9</td>
<td>14.4</td>
</tr>
<tr>
<td>sunflow</td>
<td>992</td>
<td>6.6</td>
<td>478</td>
<td>6.1</td>
<td>10.0</td>
</tr>
</tbody>
</table>
Precision Comparison

- Pointer abstraction:
  - Allocation sites
- Control abstraction:
  - Flow insensitive
  - Context insensitive

- Pointer abstraction:
  - 2-partition
- Control abstraction:
  - Flow sensitive
  - Context sensitive
Precision Comparison

- Previous scalable approach resolves 27% of queries
- Our approach resolves 82% of queries
  - 55% of queries are proven thread-local
  - 27% of queries are observed thread-shared
## Running Time Breakdown

<table>
<thead>
<tr>
<th>Database</th>
<th>Baseline Static Analysis</th>
<th>Baseline Dynamic Analysis</th>
<th>Our Approach Static Analysis</th>
<th>Per Query Group Total</th>
<th>Per Query Group Mean</th>
<th>Per Query Group Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>hedc</td>
<td>24s</td>
<td>6s</td>
<td>38s</td>
<td>1s</td>
<td>2s</td>
<td></td>
</tr>
<tr>
<td>weblech</td>
<td>39s</td>
<td>8s</td>
<td>1m</td>
<td>2s</td>
<td>4s</td>
<td></td>
</tr>
<tr>
<td>lusearch</td>
<td>43s</td>
<td>31s</td>
<td>8m</td>
<td>3s</td>
<td>6s</td>
<td></td>
</tr>
<tr>
<td>hsqldb</td>
<td>1m08s</td>
<td>35s</td>
<td>86m</td>
<td>11s</td>
<td>21s</td>
<td></td>
</tr>
<tr>
<td>avrora</td>
<td>1m00s</td>
<td>32s</td>
<td>41m</td>
<td>5s</td>
<td>8s</td>
<td></td>
</tr>
<tr>
<td>sunflow</td>
<td>1m18s</td>
<td>3m</td>
<td>74m</td>
<td>9s</td>
<td>19s</td>
<td></td>
</tr>
</tbody>
</table>
## Sparsity of Our Abstraction

<table>
<thead>
<tr>
<th></th>
<th>total # sites</th>
<th># sites set to all queries</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>mean</td>
<td>max</td>
<td>mean</td>
<td>max</td>
</tr>
<tr>
<td>heduc</td>
<td>1,914</td>
<td>3.2</td>
<td>12</td>
<td>1.4</td>
<td>5</td>
</tr>
<tr>
<td>weblech</td>
<td>2,958</td>
<td>2.2</td>
<td>8</td>
<td>1.5</td>
<td>5</td>
</tr>
<tr>
<td>lusearch</td>
<td>3,549</td>
<td>2.2</td>
<td>18</td>
<td>1.5</td>
<td>18</td>
</tr>
<tr>
<td>hsqldb</td>
<td>5,056</td>
<td>2.7</td>
<td>56</td>
<td>1.3</td>
<td>5</td>
</tr>
<tr>
<td>avrora</td>
<td>5,923</td>
<td>12.1</td>
<td>195</td>
<td>2.3</td>
<td>31</td>
</tr>
<tr>
<td>sunflow</td>
<td>6,053</td>
<td>2.2</td>
<td>18</td>
<td>1.3</td>
<td>15</td>
</tr>
</tbody>
</table>
Feedback from Real-World Deployments

• 16 bugs in jTDS
  Before: “As far as we know, there are no concurrency issues in jTDS”
  After: “Probably the whole synchronization approach in jTDS should be revised from scratch”

• 17 bugs in Apache Commons Pool
  “Thanks to an audit by Mayur Naik many potential synchronization issues have been fixed” — *Release notes for Commons Pool 1.3*

• 319 bugs in Apache Derby
  “This looks like very valuable information ... could this tool be run on a regular basis? It is likely that new races could get introduced as new code is submitted”
Talk Outline

• Overview
• Seamless Program Partitioning
• Automatic Performance Prediction
• Scalable Program Verification
• Future Directions
Program Analyses as Building Blocks

- **partitioning analysis**
- **call-graph analysis**
- **performance analysis**
- **slicing analysis**
- **pointer analysis**
- **datarace analysis**
- **thread-escape analysis**
- **may-happen-in-parallel analysis**

- Applied to partitioning, but also applicable for better scheduling, etc.
- Applied to datarace analysis, but also used for deadlock detection, etc.
- Applied to slicing, but can also yield simpler memory models, etc.

**CHORD**: a versatile program analysis platform [PLDI’11 tutorial]
Dependency Graphs of Analyses in Chord

A pointer analysis in Chord: 37 tasks, 49 targets, 154 edges

Core analyses in Chord: 147 tasks, 246 targets, 1050 edges
Applications Built Using Chord

• Systems:
  – CloneCloud: Program partitioning [EuroSys’11]
  – Mantis: Performance prediction [NIPS’10]

• Tools:
  – CheckMate: Dynamic deadlock detection [FSE’10]
  – Static datarace detection [PLDI’06, POPL’07]
  – Static deadlock detection [ICSE’09]

• Frameworks:
  – Evaluating heap abstractions [OOPSLA’10]
  – Learning minimal abstractions [POPL’11]
  – Abstraction refinement [PLDI’11]
Example Application: Configuration Debugging

- Program configuration options often badly documented
- Chord has been used to automatically produce list of options, along with types
- Can be more accurate than human-produced

“Static Extraction of Program Configuration Options”
Ariel Rabkin and Randy Katz, ICSE’11
Conclusion

- Modern computing platforms pose exciting and unprecedented software engineering problems

- Static analysis, dynamic analysis, and machine learning can be combined to solve these problems effectively

- Program analyses can serve as reusable components in solving diverse software engineering problems

Download Chord at:
http://jchord.googlecode.com