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
- energy efficiency
- data locality
- scheduling
A Challenge in Parallel Computing

Most Java programs are so rife with concurrency bugs that they work only by accident.
– Brian Goetz
Java Concurrency in Practice

How can we automatically make concurrent programs more reliable?

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

Program Partitioning: CloneCloud [EuroSys’11]

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
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
```

Output: Estimated running time of P(I)

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

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;
            $f_4$ += t;
        }
        Bldg() {
            floors = new List();
            for (…) $f_j$++;
            floors.add(new Floor());
            for (…) $f_j$++;
            Elev e = new Elev(floors);
            e.start();
            events = new List();
            for (…) $f_j$++;
            events.add(new BP(…));
        }
    }
    R = .3 + .5 $f_4^2$ + .8 $f_6$
}
```
Static Slicing Analysis

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

- Computed using data and control dependencies

From Dependencies to Slices

\text{static-slice}(f_4) = \{ \text{all actions that may affect value of } 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());
      for (...) {
        Elev e = new Elev(floors);
        e.start();
        \text{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;
            f₁ += 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(...));
        }
    }
}

R = .3 + .5f₄ + .8f₄²
```

- 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

<table>
<thead>
<tr>
<th></th>
<th>f₁</th>
<th>...</th>
<th>fₙ</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>
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

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;
            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(...));
            }
        }
    }
}
```

... or include these actions in the slice as well

Either prove these actions thread-local ...

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;
            p:
        }
        Bldg() {
            floors = new List();
            for (...) {
                floors.add(new Floor());
                Elev e = new Elev(floors);
                e.start();
                events = new List();
                for (...) {
                    events.add(new BP(...));
                }
            }
        }
    }
}
```
The Thread-Escape Problem

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

- Needs to reason about \textit{all} inputs \implies \text{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);
        }
    }
    Bldg() {
        floors = new List();
        for (...) {
            floors.add(new Floor());
        }
    }
}
class List {
    List() {
        this.elems = new Object[...];
    }
}
```

Complexity of Static Analysis

<table>
<thead>
<tr>
<th>pointer abstraction</th>
<th>max abstract values (N)</th>
<th>control-flow abstraction</th>
<th>max abstract states</th>
</tr>
</thead>
<tbody>
<tr>
<td>trivial</td>
<td>1</td>
<td>flow and context insensitive</td>
<td>1</td>
</tr>
<tr>
<td>allocation sites</td>
<td>H</td>
<td>flow sensitive context insensitive</td>
<td>L</td>
</tr>
<tr>
<td>k-CFA</td>
<td>H . I^k</td>
<td>flow and context sensitive</td>
<td>L . 2^(N^2 . 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:

| 2-partition | 2 |
| flow and context sensitive | Q . L . 4^F |

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
  \[ \Rightarrow \text{existing analyses sacrifice precision and/or scalability} \]

```
P ⊢ Q_1?
P ⊢ Q_2?
```

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

```
P ⊢ Q_1?
P ⊢ Q_2?
```

```
P ⊢ Q_1?
P ⊢ Q_2?
```
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());
        for (…)
            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 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
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());
        }
        for (…) {
            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</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>hedc</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

Previous Approach

- Pointer abstraction:
  - Allocation sites

- Control abstraction:
  - Flow insensitive
  - Context insensitive

Our Approach

- Pointer abstraction:
  - 2-partition

- Control abstraction:
  - Flow sensitive
  - Context sensitive

• 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></th>
<th>baseline static analysis</th>
<th>our approach</th>
<th>static analysis</th>
<th>total</th>
<th>per query group</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>dynamic analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mean</td>
<td>max</td>
<td></td>
</tr>
<tr>
<td>hedc</td>
<td>24s</td>
<td>6s</td>
<td>38s</td>
<td>1s</td>
<td>2s</td>
</tr>
<tr>
<td>weblech</td>
<td>39s</td>
<td>8s</td>
<td>1m</td>
<td>2s</td>
<td>4s</td>
</tr>
<tr>
<td>lusearch</td>
<td>43s</td>
<td>31s</td>
<td>8m</td>
<td>3s</td>
<td>6s</td>
</tr>
<tr>
<td>hsqldb</td>
<td>1m08s</td>
<td>35s</td>
<td>86m</td>
<td>11s</td>
<td>21s</td>
</tr>
<tr>
<td>avrora</td>
<td>1m00s</td>
<td>32s</td>
<td>41m</td>
<td>5s</td>
<td>8s</td>
</tr>
<tr>
<td>sunflow</td>
<td>1m18s</td>
<td>3m</td>
<td>74m</td>
<td>9s</td>
<td>19s</td>
</tr>
</tbody>
</table>

### Sparsity of Our Abstraction

<table>
<thead>
<tr>
<th></th>
<th>total # sites</th>
<th>all queries</th>
<th>proven queries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>mean</td>
<td>max</td>
</tr>
<tr>
<td>hedc</td>
<td>1,914</td>
<td>3.2</td>
<td>12</td>
</tr>
<tr>
<td>weblech</td>
<td>2,958</td>
<td>2.2</td>
<td>8</td>
</tr>
<tr>
<td>lusearch</td>
<td>3,549</td>
<td>2.2</td>
<td>18</td>
</tr>
<tr>
<td>hsqldb</td>
<td>5,056</td>
<td>2.7</td>
<td>56</td>
</tr>
<tr>
<td>avrora</td>
<td>5,923</td>
<td>12.1</td>
<td>195</td>
</tr>
<tr>
<td>sunflow</td>
<td>6,053</td>
<td>2.2</td>
<td>18</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
Performance analysis
Slicing analysis
Pointer analysis
Datarace analysis
Thread-escape analysis
May-happen-in-parallel analysis

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