Learning from Executions

Dynamic analysis for program understanding and software engineering

Michael D. Ernst and Jeff H. Perkins

November 7, 2005
Tutorial at ASE 2005

Outline

Goals

Interaction among participants
Dynamic analysis is profitable: effective and easy

- I didn't know you could use dynamic analysis for that
- I didn't know how to do that with dynamic analysis

Generate ideas for your work

Dynamic analysis

Examples: profiling, testing

Execute program (over some inputs)

- The compiler provides the semantics

Observe executions

- Requires instrumentation infrastructure

Analyze the results

- Use aggregation, inference, etc.

Must choose what to measure, and what test runs

Analysis challenge: What to measure?

Coverage or frequency

- Statements, branches, paths, procedure calls, types, method dispatch

Values computed

- Parameters, array indices

Run time, memory usage

Test oracle results

- Similarities among runs [Podgurski 99, Reps 97]

This choice determines what is reported

Analysis challenge: Choose good tests

The test suite determines the expense (in time and space)
The test suite determines the accuracy (what executions are never seen)

- Less accurate results are poor for applications that require correctness
- Many domains do not require correctness!

* What information is being collected also matters

Outline

What is dynamic analysis good for?

What is dynamic analysis good for?

Correctness

What is dynamic analysis good for?

Correctness
Performance

What is dynamic analysis good for?

Correctness

Performance (optimization)

- Profile-directed compilation: feed into a (static) compiler
  - control optimizations: fall-through branches, hot paths (trace scheduling), specialize for common case
  - data optimizations: lay out data structures, cache-aware algorithms, speculation

What is dynamic analysis good for?

Correctness

Performance (optimization)

- Profile-directed compilation: feed into a (static) compiler
  - control optimizations: fall-through branches, hot paths (trace scheduling), specialize for common case
  - data optimizations: lay out data structures, cache-aware algorithms, speculation

- Dynamic re-compilation: JIT
  - control optimizations: re-compile hot blocks/paths/procedures re-order dispatch table
  - data optimizations:
    - garbage collection: move objects for locality
    - specialization, partial evaluation (possibly better constant propagation, etc.)

Dynamic analysis for correctness

Finding bugs

- Violations of user-specified properties
- Deviations from past behavior (e.g., intrusion detection)

Testing

- Creating test suites
- Improving test suites

Enabling transformations

Program understanding

Dynamic analysis features

Can be as fast as execution (over a test suite, and allowing for data collection)

- Example: aliasing

Precise: no abstraction or approximation

Unsound: results may not generalize to future executions

- Describes execution environment or test suite

Outline

What is dynamic analysis?

Uses for dynamic analysis

Comparison with static analysis

Dynamic analysis techniques

Problem-solving and brainstorming

Implementing dynamic analysis: Executions

Implementing dynamic analysis: Monitoring

Implementing dynamic analysis: Data analysis/reporting

Combining static and dynamic analysis

Static analysis

Examples: compiler optimizations, type checkers, lint, program verifiers

Examine program text (no execution)

Build a model of program state

- An abstraction of the run-time state

Reason over possible behaviors

- E.g., "run" the program over the abstract state

Abstract interpretation

Typically implemented via dataflow analysis

Each program statement's transfer function indicates how it transforms the (abstract) state

Example: What is the transfer function for

```
y = x++;
```

?

Selecting an abstract domain

Abstract domain: { even, odd, either }

$\langle$x is odd; y is odd$\rangle$

```
y = x++;
```

$\langle$x is even; y is odd$\rangle$

Selecting an abstract domain

Abstract domain: { even, odd, either }

$\langle$x is odd; y is odd$\rangle$
```
y = x++;
```
$\langle$x is even; y is odd$\rangle$

Abstract domain: { prime, nonprime, anything }

$\langle$x is prime; y is prime$\rangle$
```
y = x++;
```
$\langle$x is anything; y is prime$\rangle$

Selecting an abstract domain

Abstract domain: set of numbers, one set per variable

$\langle \mathrm{x} = \{\ 3,\ 5,\ 7\ \};\ \mathrm{y} = \{\ 9,\ 11,\ 13\ \}\rangle$

```
y = x++;
```

$\langle \mathrm{x} = \{\ 4,\ 6,\ 8\ \};\ \mathrm{y} = \{\ 3,\ 5,\ 7\ \}\ \rangle$

Selecting an abstract domain

Abstract domain: set of numbers, one set per variable

$\langle$x $= \{$ 3, 5, 7 $\}$; y $= \{$ 9, 11, 13 $\}\rangle$

```
y = x++;
```

$\langle$x $= \{$ 4, 6, 8 $\}$; y $= \{$ 3, 5, 7 $\}$ $\rangle$

Abstract domain: set of environments

- environment assigns a variable to a number

$\langle$x=3, y=11$\rangle$, $\langle$x=5, y=9$\rangle$, $\langle$x=7, y=1$\rangle$

```
y = x++;
```

$\langle$x=4, y=3$\rangle$, $\langle$x=6, y=5$\rangle$, $\langle$x=8, y=7$\rangle$

Selecting an abstract domain

Abstract domain: set of numbers, one set per variable

$\langle \mathrm{x} = \{\ 3,\ 5,\ 7\ \};\ \mathrm{y} = \{\ 9,\ 11,\ 13\ \} \rangle$

```
y = x++;
```

$\langle \mathrm{x} = \{\ 4,\ 6,\ 8\ \};\ \mathrm{y} = \{\ 3,\ 5,\ 7\ \}\ \rangle$

Abstract domain: set of environments

- environment assigns a variable to a number

$\langle \mathrm{x}{=}3,\ \mathrm{y}{=}11 \rangle,\ \langle \mathrm{x}{=}5,\ \mathrm{y}{=}9 \rangle,\ \langle \mathrm{x}{=}7,\ \mathrm{y}{=}1 \rangle$

```
y = x++;
```

$\langle \mathrm{x}{=}4,\ \mathrm{y}{=}3 \rangle,\ \langle \mathrm{x}{=}6,\ \mathrm{y}{=}5 \rangle,\ \langle \mathrm{x}{=}8,\ \mathrm{y}{=}7 \rangle$

Abstract domain: symbolic expression per variable

$\langle x_n = f_1(a_{n-1}, \ldots, z_{n-1}); y_n = f_2(a_{n-1}, \ldots, z_{n-1}) \rangle$

```
y = x++;
```

$\langle x_{n+1} = x_n + 1; y_{n+1} = x_n \rangle$

Analysis challenge: Choose good abstractions

The abstraction determines the expense (in time and space)
The abstraction determines the accuracy (what information is lost)

- Less accurate results are poor for applications that require precision
- Cannot conclude all true properties in the grammar

Static analysis features

Slow to analyze large models of state, so use abstraction
Conservative: account for abstracted-away state
Sound: (weak) properties are guaranteed to be true

- Some static analyses are not sound

Comparing static and dynamic analysis

Static analysis

Abstract domain
  slow if precise
Conservative
  due to abstraction
Sound
  due to conservatism

Dynamic analysis

Concrete execution
  slow if exhaustive
Precise
  no approximation
Unsound
  does not generalize

Outline

Coverage

Structural coverage: lexical structures in the source code

- statement, branch, path, other (def-use, condition, MCDC, etc.)
- measure the number of them (most interesting is 0 vs. non-zero)
- measure the time each one takes to execute (e.g., PC sampling)

Specification coverage

- Like structural coverage, but for expressions in the specification

Value coverage

Static analogs: dead code analysis; prediction of execution frequency

Type safety

No memory corruption or operations on wrong types of values

Dynamic type-checking

- Each value carries its type at run time
- When performing an operation, check the types of the argument
- The only type checking in Lisp, Perl, PHP, Python, Smalltalk, . . .
- Useful in statically typed languages
  - Java dynamically checks casts, has `instanceof`
  - C has unions
  - C++ has casts and RTTI (run-time type identification)
  - Checking type of data from external locations (files, remote calls)

Static analog: static type-checking

- Types are compile-time approximations to values
- Every type system prohibits some safe (correct) code

Slicing

What computations could affect a value
What lines of the program can affect the output?

```
if (a < 0)
  b = d;
else
  b = d
if (w < 0)
  x = y;
else
  x = z;
print(x);
```

Dynamic slicing: tracing run-time values

- Each value is tagged by
  - where it was computed
  - the values that computed it
- Follow the links backward to determine the full computation (reachability)

Static slicing: reachability over dependence graph

Memory checking

Goal: find array bound violations, uses of uninitialized memory

- Want to find them as soon as they happen

Purify [Hastings 92]: run-time instrumentation

- Tagged memory: 2 bits (allocated, initialized) per byte
- Each instruction checks/updates the tags
  - Allocate (`malloc`): set "A" bit, clear "I" bit
  - Write (`x = ...`): require "A" bit, set "I" bit
  - Read (`... = x`): require "I" bit
  - Deallocate (`free`): clear "A" bit

Static analog: LCLint [Evans 96] compile-time dataflow analysis

- Abstract state contains allocated and initialized bits
- Each transfer function checks/updates the state

Race detection

Data race: two threads simultaneously update a data structure

- Solution: locking ensures only one thread at a time has access

Eraser [Savage 1997]: check that all shared memory accesses follow a consistent locking discipline

- Each memory location is tagged with a state:
  - virgin
  - exclusive
  - shared
  - shared-modified (indicates a race condition)

Atomicity checking: uses identical analyses statically and dynamically [Flanagan 2003]

Alias analysis

To determine whether two pointers are the same; compare the addresses
Static analog: abstract representation of the heap

Specification checking

Verify that code satisfies its specification

Dynamic analysis: `assert` statement

- Language is familiar to the programmer, easy to write in
- May need to remember previous values to confirm updates (pre-conditions vs. post-conditions)

Static analysis: formal verification

- theorem-proving, either manually or with machine assistance
- an open research problem

Specification generation I

```
public class StackAr {
  Object[] theArray;
  int      topOfStack;
  ...
}
```

Object properties

```
theArray != null
theArray.getClass() == java.lang.Object[].class
topOfStack >= -1
topOfStack <= theArray.length - 1
theArray[0..topOfStack] elements != null
theArray[topOfStack+1..] elements == null
```

Pre-conditions for the StackAr constructor

```
capacity >= 0
```

Specification generation II

Post-conditions for the StackAr constructor

```
orig(capacity) == theArray.length
theArray[] elements == null
topOfStack == -1
theArray[0..topOfStack] == []
```

Post-conditions for the isFull method

```
theArray == orig(theArray)
theArray[] == orig(theArray[])
topOfStack == orig(topOfStack)
(return == false)  <==>  (topOfStack < theArray.length - 1)
(return == true)   <==>  (topOfStack == theArray.length - 1)
```

Specification generation

Dynamic invariant detection [Ernst 99]

- Machine learning over values the program computes
  - Generate-and-test strategy
  - Statistical tests to combat overfitting
- Daikon implementation is publicly available: http://pag.csail.mit.edu/daikon/
- Many applications:
  - verifying safety properties [Vaziri 1998, Nimmer 2002]
  - automating theorem-proving [Ne Win 2003]
  - identifying refactoring opportunities [Kataoka 2001]
  - predicate abstraction [Dodoo 2002]
  - generating test cases [Xie 2003, Gupta 2003, Pacheco 2005]
  - selecting and prioritizing test cases [Harder 2003]
  - explaining test failures [Groce 2003]
  - predicting incompatibilities in component upgrades [McCamant 2003]
  - error detection [Raz 2002, Hangal 2002, Pytlik 2003, Mariani 2004, Brun 2004]
  - error isolation [Xie 2002, Liblit 2003]
  - choosing modalities [Lin 2004]
  - . . . and more

Statically: by hand or abstract interpretation [Cousot 77]

Outline

Problem-solving and brainstorming

Interactive discussion

Goal:

- return home with ideas you can use
- return home with a to-do item

Start from

- specific problems
  - try to find solutions
- specific static analyses
  - try to find analogs

Outline

What is dynamic analysis?

Uses for dynamic analysis

Comparison with static analysis

Dynamic analysis techniques

Problem-solving and brainstorming

Implementing dynamic analysis: Executions

Implementing dynamic analysis: Monitoring

Implementing dynamic analysis: Data analysis/reporting

Combining static and dynamic analysis

Where to get them

- Any old place tends to work pretty well.

How good are they in practice?

- Surprisingly good
- In other words: don't worry

Evaluating test suites

The output

- tells you about the program, *or*
- tells you about the test suite or execution environment

Compare suite with actual usage

- i.e., compare analysis results over suite with analysis results over actual usage
- overhead can be a problem for current usage

Compare suite with augmented suite

- Do the results change?
- Our results suggest that you achieve diminishing returns very fast

Improving test suite quality

Use a bigger suite; sample if necessary
Lots of research in test generation
The dynamic analysis output itself tells you what you need to know

Outline

What is dynamic analysis?

Uses for dynamic analysis

Comparison with static analysis

Dynamic analysis techniques

Problem-solving and brainstorming

Implementing dynamic analysis: Executions

Implementing dynamic analysis: Monitoring

Implementing dynamic analysis: Data analysis/reporting

Combining static and dynamic analysis

Outline

Outline

What is dynamic analysis?

Uses for dynamic analysis

Comparison with static analysis

Dynamic analysis techniques

Problem-solving and brainstorming

Implementing dynamic analysis: Executions

Implementing dynamic analysis: Monitoring

Implementing dynamic analysis: Data analysis/reporting

Combining static and dynamic analysis

Combining static and dynamic analysis

1. Aggregation:
   Pre- or post-processing
2. Inspiring analogous analyses:
   Same problem, different domain
3. Hybrid analyses:
   Blend both approaches

1. Aggregation: Pre- or post-processing

Use output of one analysis as input to another

Dynamic then static

- Profile-directed compilation: unroll loops, inline, reorder dispatch, ...
- Verify properties observed at run time

Static then dynamic

- Reduce instrumentation requirements
  - Efficient branch/path profiling
  - Discharge obligations statically (type/array checks)
- Type checking (e.g., Java, including generics)
- Indicate suspicious code to test more thoroughly

2. Analogous analyses: Same problem, different domain

Any analysis problem can be solved in either domain
Examples:

- Type safety
- Slicing
- Memory checking
- Atomicity
- Specification checking
- Specification generation

Your analogous analyses here

Look for gaps with no analogous analyses!
Try using the same analysis

- But be open to completely different approaches

There is still low-hanging fruit to be harvested

3. Hybrid analyses: Blending static and dynamic

Combine static and dynamic analyses

- Not mere aggregation, but a new analysis
- Disciplined trade-off between precision and soundness: find the sweet spot between them

Possible starting points

Analyses that trade off run-time and precision

- Different abstractions (at different program points) Switch between static and dynamic at analysis time

Ignore some available information

- Examine only some paths [Evans 94, Detlefs 98, Bush 00]

Merge based on observation that both examine only a subset of executions

- Problem: optimistic vs. pessimistic treatment

More examples: (bounded) model checking, security analyses, delta debugging [Zeller 99], etc.