Dynamic Purity Analysis for Java Programs

Haiying Xu Chris Pickett Clark Verbrugge
{hxu31,cpicke,clump}@sable.mcgill.ca

Sable Research Group, McGill University
Montréal, Québec, Canada H3A 2A7

PASTE 2007

June 14, 2007

Outline

What is Method Purity?

Roughly, a pure method has *no externally visible side effects*.

Different variations on purity are possible:

- Sălcianu and Rinard:
  can create, modify and return new objects
- Rountev:
  similar, but cannot return a new object

Why is Method Purity Important?

Artzi, Kiezun, Glasser, Ernst:

- program comprehension
- modelling
- formal verification
- compiler optimization
- memoization
- thread level speculation
- stack allocation
- refactoring
- test input generation
- regression oracle creation
- invariant detection
- specification mining
- program slicing

Why is Method Purity Important?

Artzi, Kiezun, Glasser, Ernst:

- program comprehension
- modelling
- formal verification
- compiler optimization
- **memoization**
- thread level speculation
- stack allocation
- refactoring
- test input generation
- regression oracle creation
- invariant detection
- specification mining
- program slicing

Contributions

In this work, we:

- Design and implement *dynamic purity analysis*.
- Investigate several different purity definitions.
- Introduce three different dynamic purity metrics.
- Implement memoization as a purity consumer.

Outline

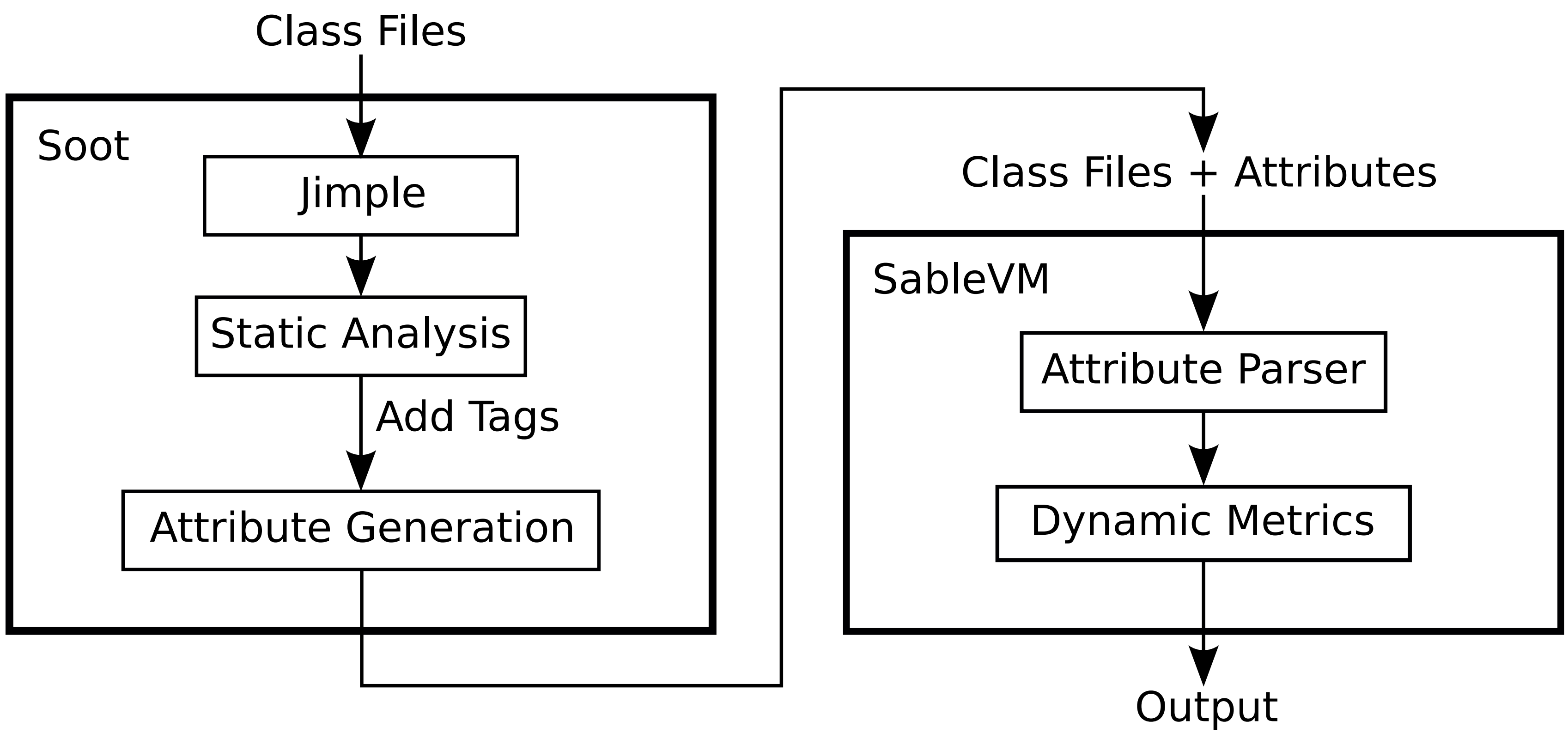
Static Purity Analysis

Consider the classic functional form of purity:

A method is strongly pure iff it

- Does not r/w the heap or static data
- Does not perform any synchronization
- Does not invoke any native method
- Does not invoke any impure method

Static Purity Analysis Framework



Static Analysis Results

metric	comp	db	jack	javac	jess	mpeg	rt
static method purity	14%	13%	13%	12%	13%	13%	13%
dynamic method purity	6%	6%	6%	5%	5%	6%	5%
dynamic invocation purity	≈0%	2%	10%	10%	6%	16%	3%
dynamic bytecode purity	≈0%	2%	1%	≈0%	≈0%	2%	≈0%

- Static/dynamic method purity
  % of reachable/reached methods that are pure
- Dynamic invocation purity
  % of all invocations that are pure
- Dynamic bytecode purity
  % of bytecode instruction stream contained in a pure method

Outline

Motivation for Dynamic Purity Analysis

Static purity analysis is hard:

- Implementation is complex
- Whole-program analysis is expensive

Dynamic evaluation tells a different story:

- Static vs. dynamic call graph
- Choice of metrics
- Input dependence

Motivation for Dynamic Purity Analysis

- Purity can also depend on method input:

```
int x;
void foo (boolean b) {
    if (b)
        x = 10;
}
```

If we only ever execute `foo (false)`, foo is pure!

Different Kinds of Dynamic Purity

Four different kinds of dynamic purity:

- **Strong**: the same as strong static purity
  – no heap or static r/w
  – no calls to impure methods
- **Moderate**:
  – allow object allocation, if the object does not escape
  – allow heap r/w to non-escaping objects
  – allow calls to certain impure methods
- **Weak**:
  – moderate, but no limitations on heap reads
- **Once-Impure**:
  – weak, but no restrictions on the first invocation

Moderate Purity

```
class Obj {
    int f;
    public Obj() {
        f = 10;
    }
    Obj bar() {
        Obj o = new Obj();
        return o;
    }
    int foo() { // moderately pure
        Obj o = bar();
        return o.f;
    }
    ...
```

Weak and Once-Impure Purity

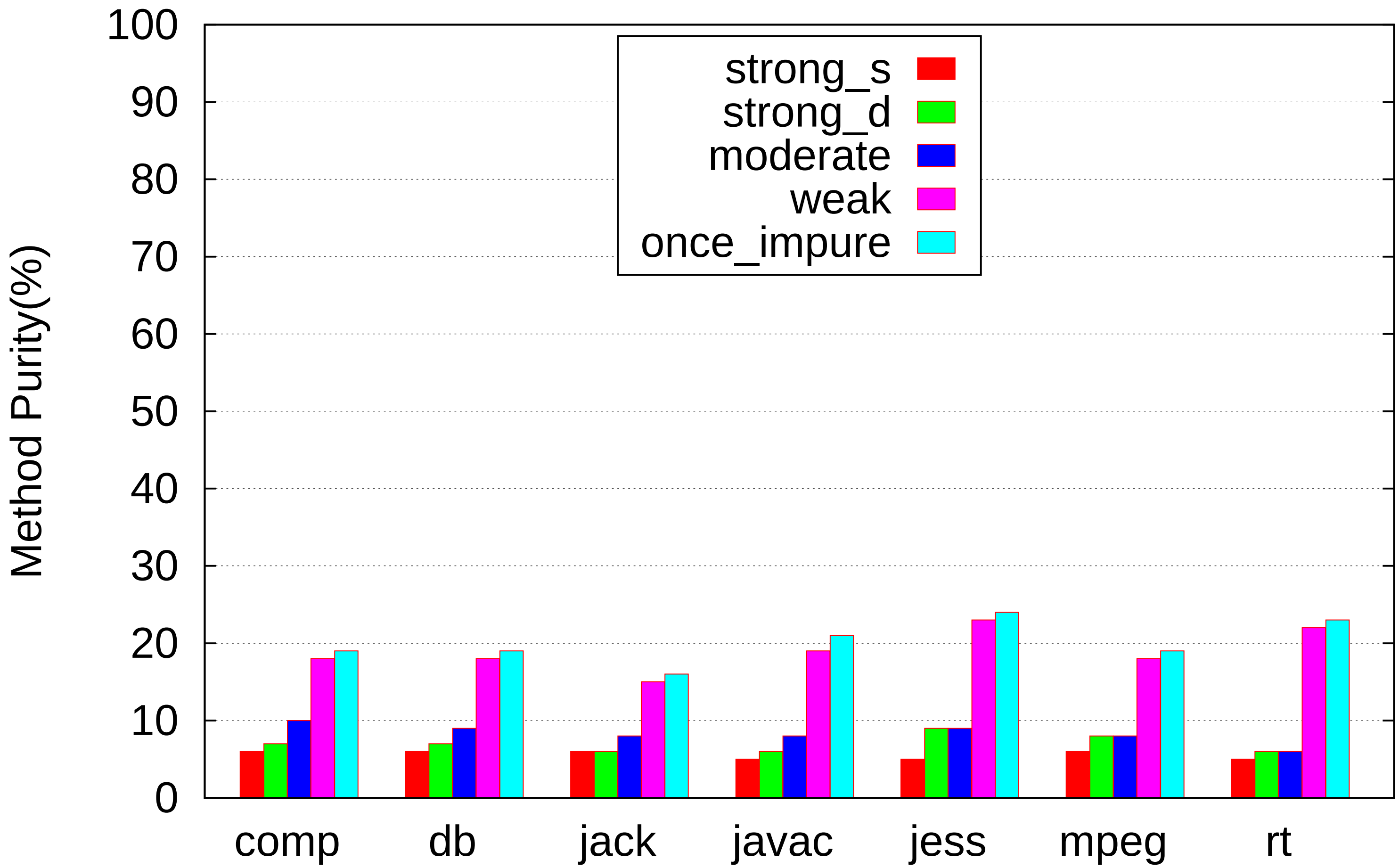
```
    ...
    static int x;

    int baz (Obj o) { // weakly pure
        return o.f;
    }

    int baf (boolean b) { // once-impure for TF+
        if (b) {
            Obj.x = 9 * 6; // write to static field
        }
        return 42;
    }
}
```

Outline

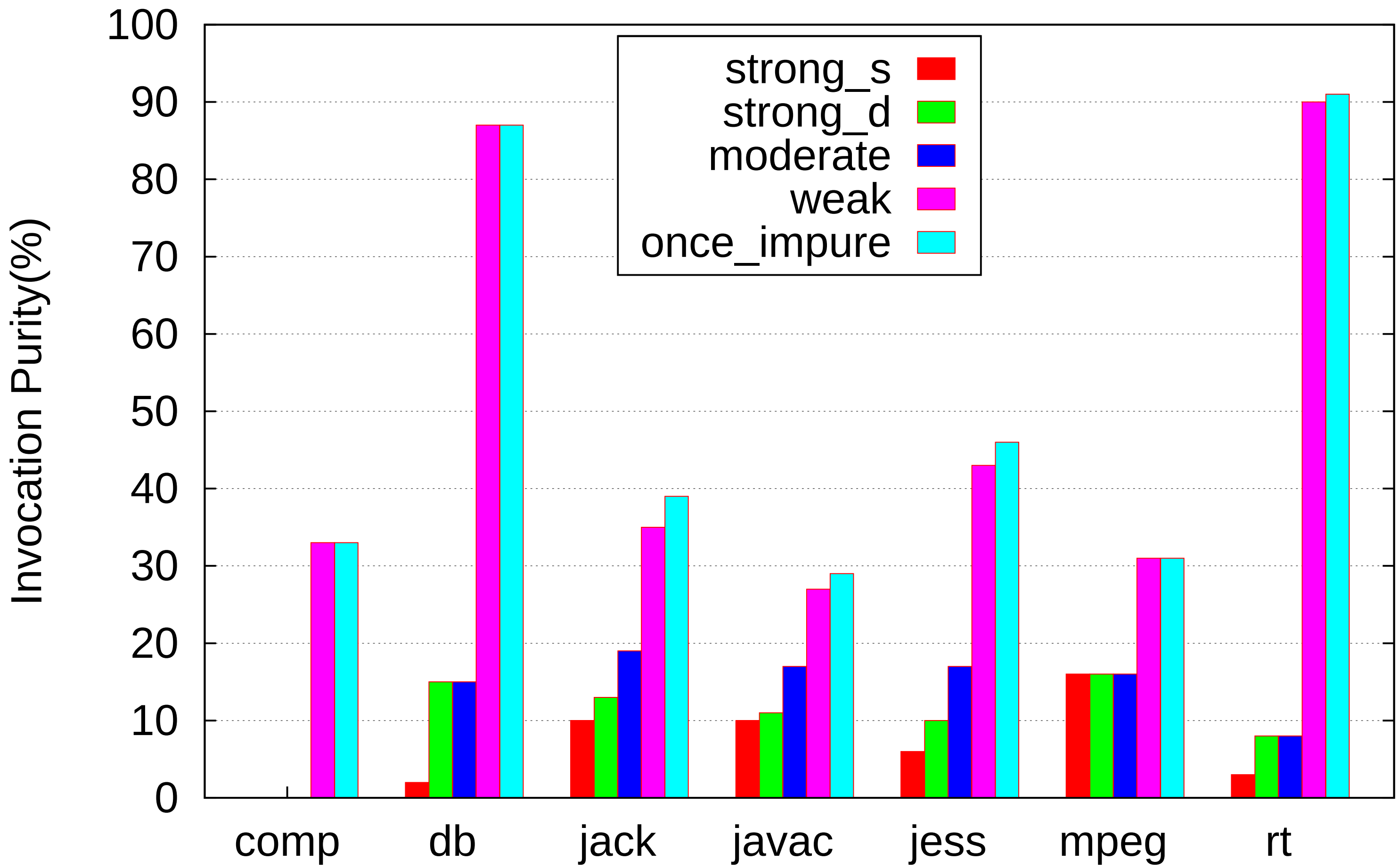
Method Purity



Fairly uniform across benchmarks.

Moderate purity does not improve much—cannot dereference input.

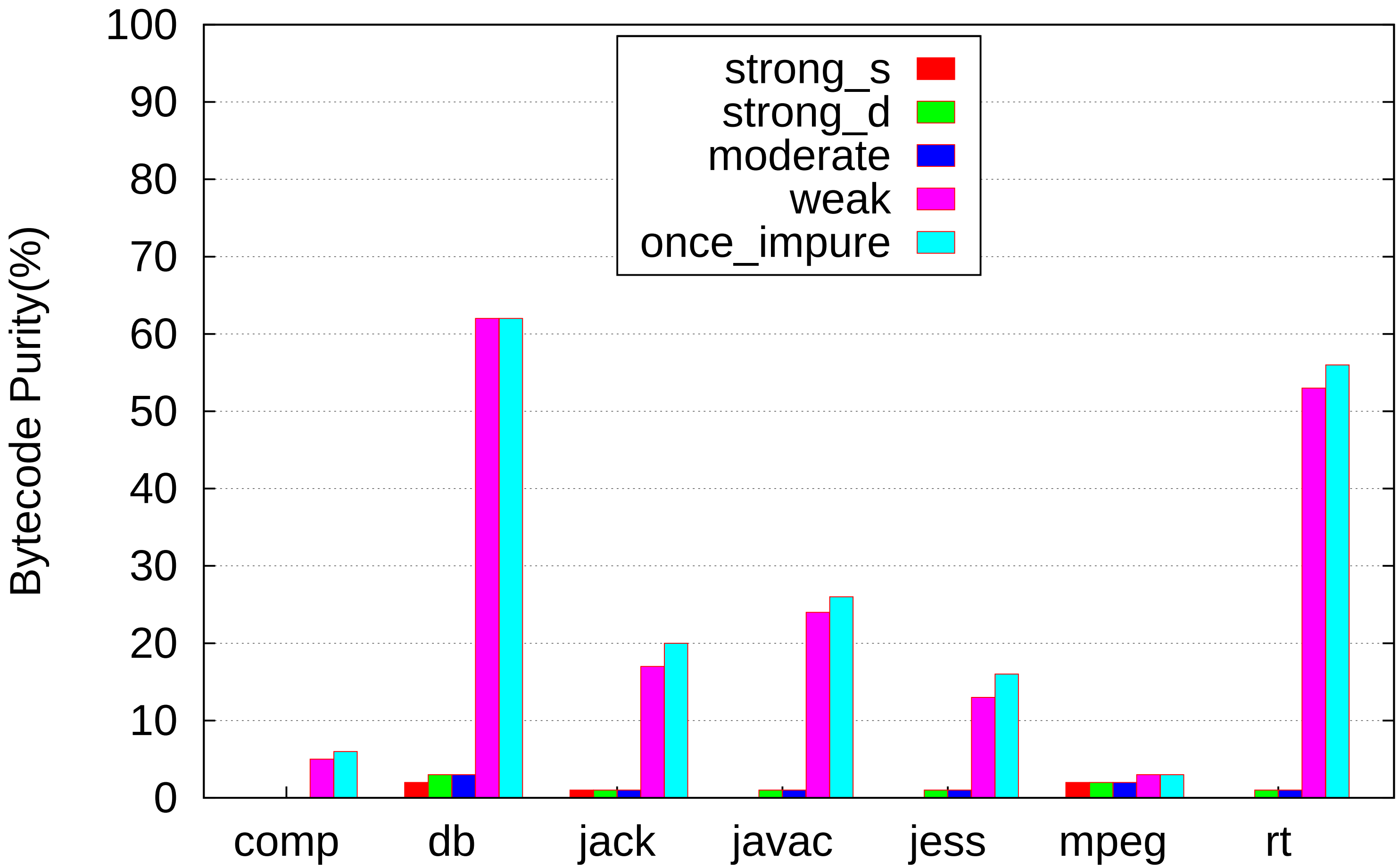
Invocation Purity



Unpredictable from method purity.

Two different groups appear.

Bytecode Purity



Somewhat predictable from invocation purity.
Three different groups appear.

Sources of Impurity

source	comp	db	jack	javac	jess	mpeg	rt
PUTFIELD	27%	29%	21%	21%	23%	24%	28%
PUTFIELD+	52%	52%	58%	66%	61%	60%	53%

method impurity

source	comp	db	jack	javac	jess	mpeg	rt
PUTFIELD	81%	82%	45%	25%	24%	40%	71%
PUTFIELD+	19%	17%	37%	58%	19%	60%	28%

invocation impurity

source	comp	db	jack	javac	jess	mpeg	rt
PUTFIELD	21%	85%	38%	25%	8%	11%	33%
PUTFIELD+	79%	13%	48%	66%	45%	89%	66%

bytecode impurity

PUTFIELD is the main reason for impurity.

Outline

Using Purity for Memoization

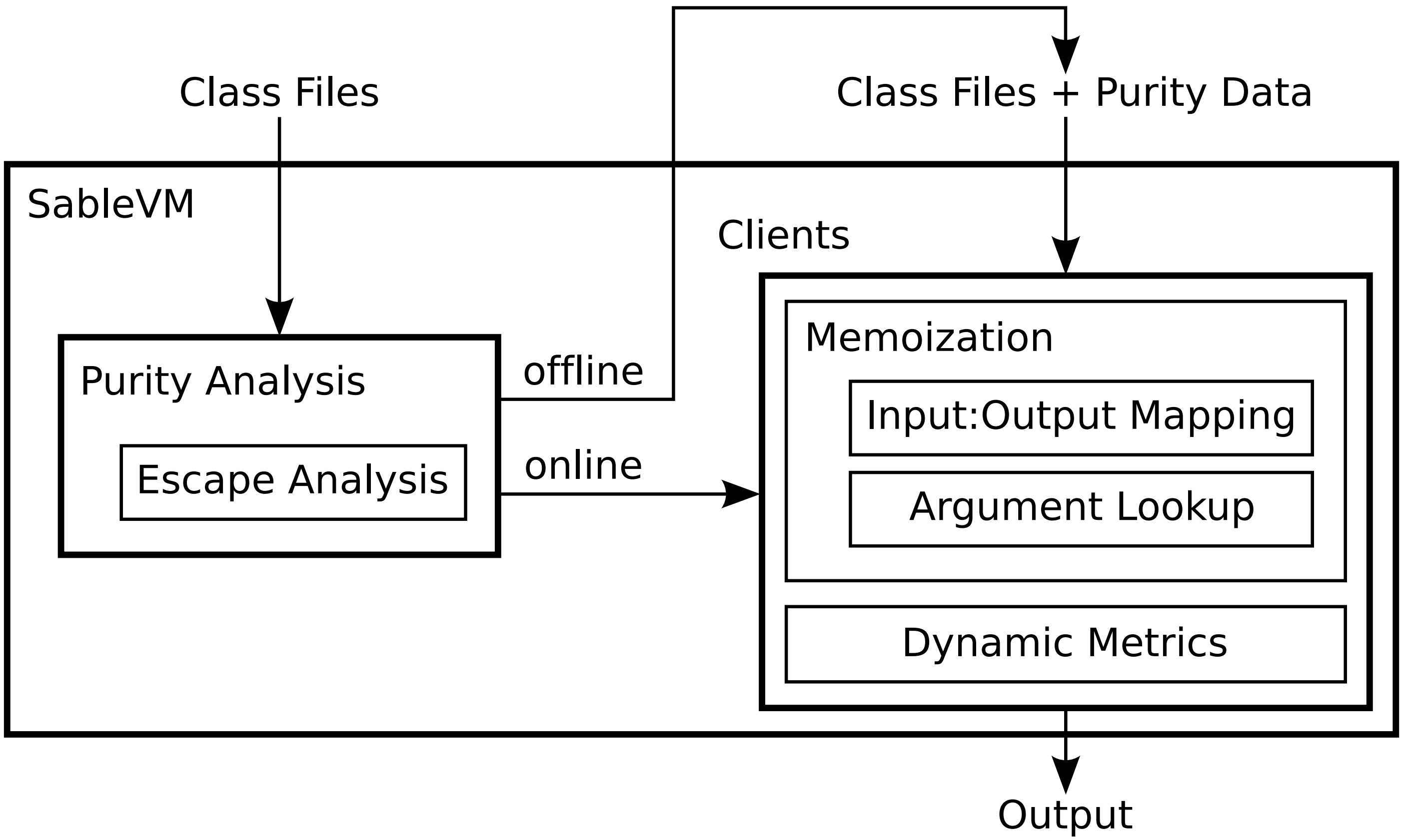
Overview of memoization:

- Maps method input to output
- Allows repeat invocations of pure methods to be skipped
- Once-impure purity is a natural fit

How can we use memoization?

- Candidate for optimization
- Good functional sanity test

Memoization Framework



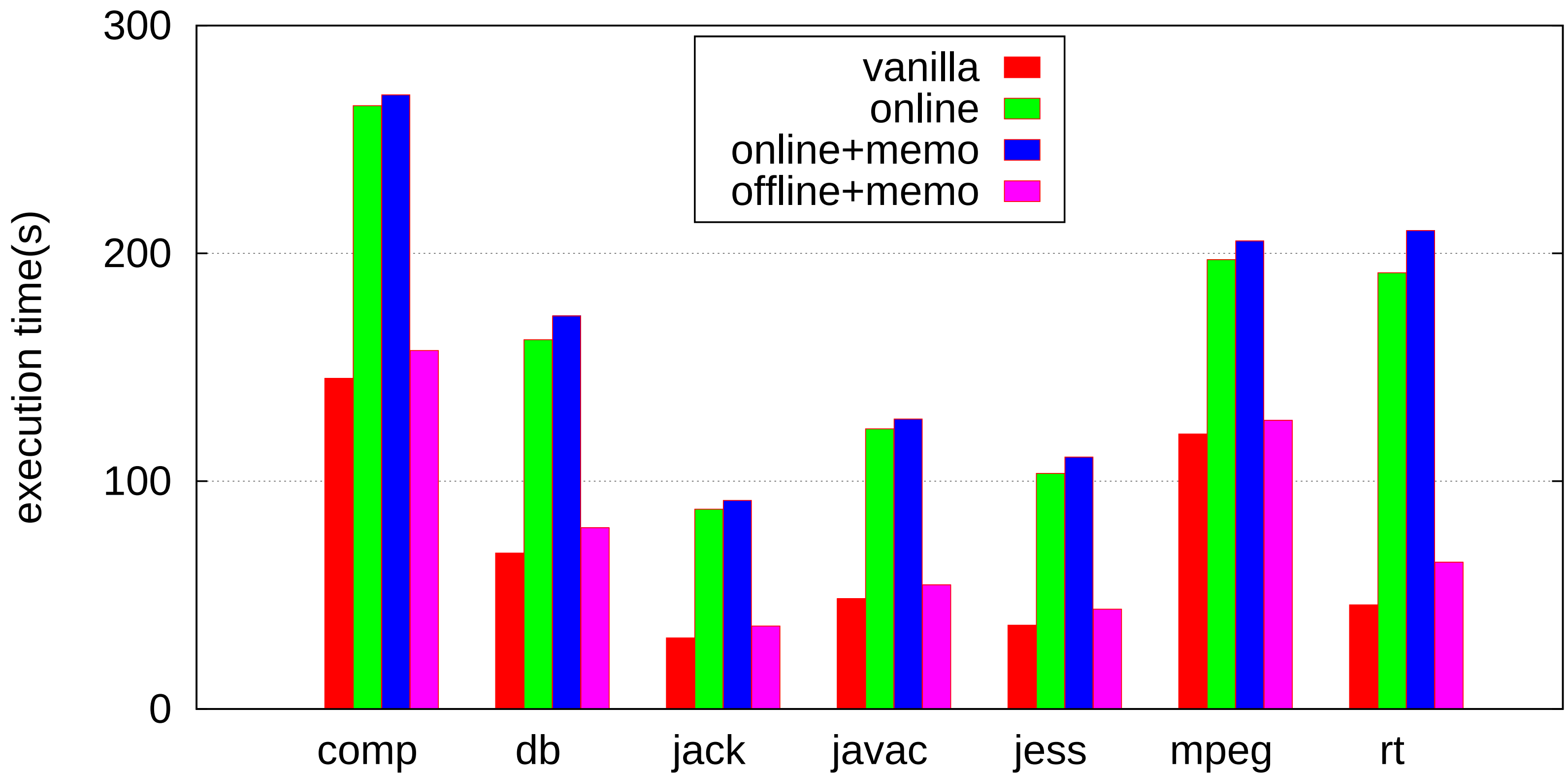
Applying Memoization

Factors influencing memoization decisions:

- Method size (50 instructions)
- Input size (100 KB—otherwise potentially the whole heap!)
- Hashtable warm up period (1000 cold start misses)
- Hit ratio (better than 1 in 10)
- Global memory consumption (1 GB)

These are fairly generous limits...

Execution Times



Memoization Improvements

Why doesn't memoization achieve speedup?

- Small number of memoized methods
- Most memoized methods are short
- Usually, less than 1% of bytecode is skipped (best case 9%)
- Implementation limitations

Potential improvements:

- Consider purity on a per-input basis
- Track only those fields read by the method
- Adaptively turn off memoization if no benefit
- Allow for cycles in input data structures

Outline

Conclusions

Static results correlate weakly with dynamic behaviour

We considered three different metrics:

- method purity varies only slightly
- invocation purity separates benchmarks into two groups
- bytecode purity separates benchmarks into three groups

Consumer applications can impose strong constraints

Future Work

Future work:

- Consider purity at different granularities
- Visualize purity evolution over time
- Support arbitrary kinds of dynamic purity
- Memoization improvements
- Other applications besides memoization (lots!)
  – e.g., speculate past nearly pure methods