Future Challenges in Dynamic Interprocedural Analysis and Optimization

Vivek Sarkar

IBM T.J. Watson Research Center

vsarkar@us.ibm.com

Outline

1. *Motivation*
2. Dynamic Optimistic Interprocedural Type Analysis (DOIT)
3. Immutability Analysis Opportunities for Dynamic IPA
4. Future Challenges

Acknowledgments / References

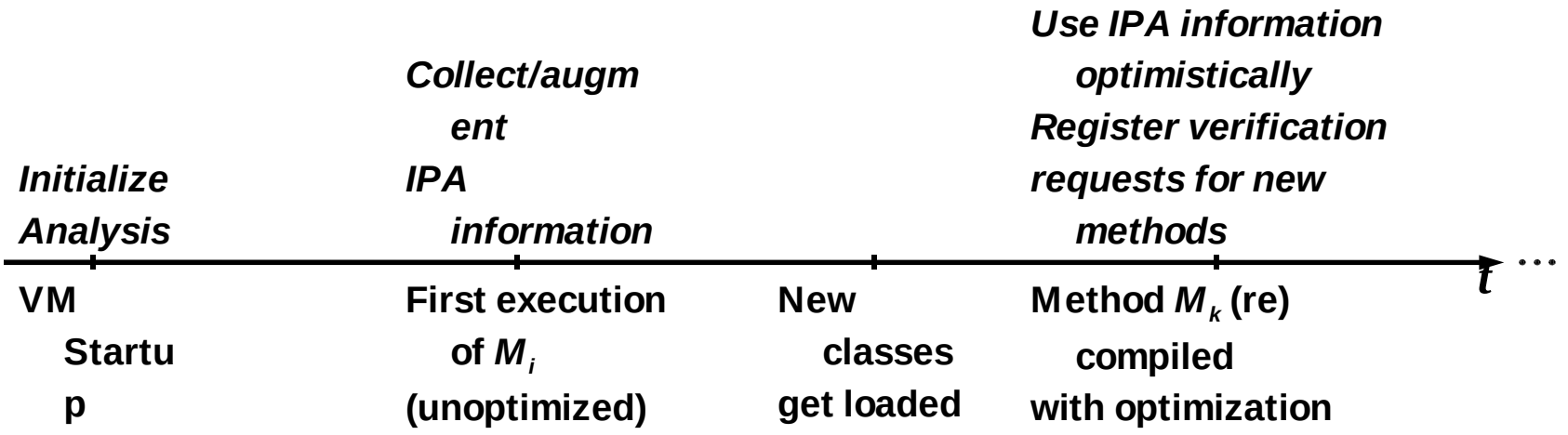
- “ Dynamic Optimistic Interprocedural Analysis: A Framework and an Application.” , OOPSLA 2001 conference
- “ Immutability Specification and its Applications” , I. Pechtchanski, V. Sarkar, JGI 2002 conference, CPE 2003 journal
- Discussions with Jikes RVM team members on interprocedural extensions to type analysis, load/store elimination, and register allocation

Motivation

- Interprocedural analysis (IPA) is essential for compiler-driven performance
  - especially when optimizing object-oriented languages
- Static IPA optimizations:
  - limited precision due to impact of methods that may not be executed
  - scalability limitations in analyzing static “ whole program”
- Dynamic intra-procedural optimizations:
  - Significant advances, with inlining, to address interprocedural optimization opportunities
  - reaching point of diminishing returns
- Dynamic IPA:
  - Opportunity to get best of both worlds

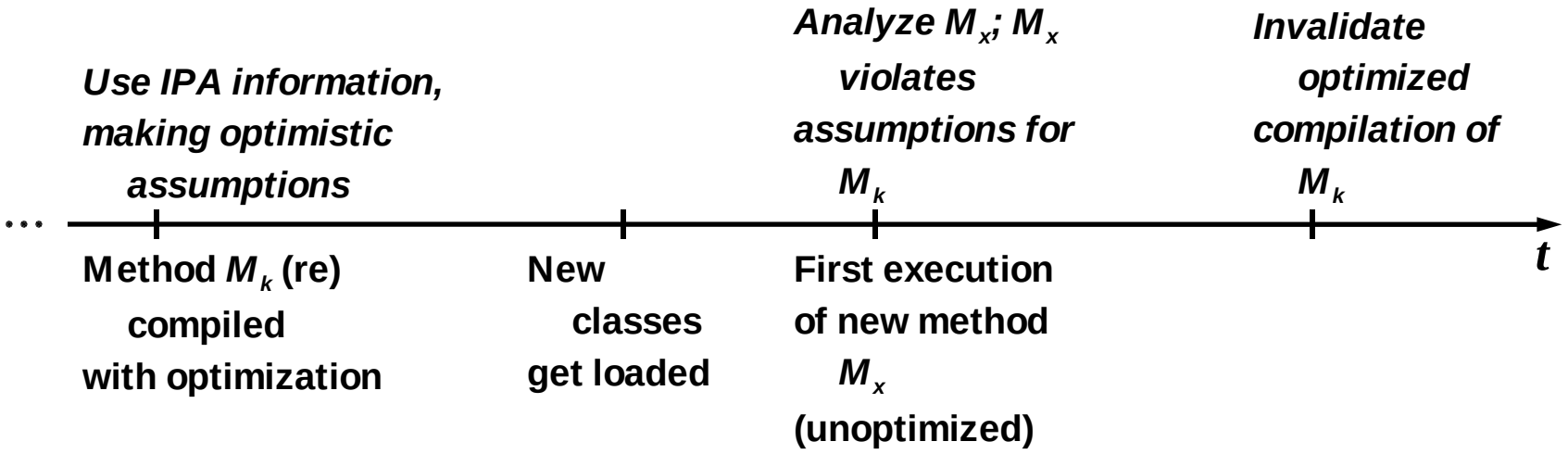
Dynamic Interprocedural Analysis Scenario

- supports dynamic class loading, adaptive optimization, optimistic assumptions about unanalyzed code

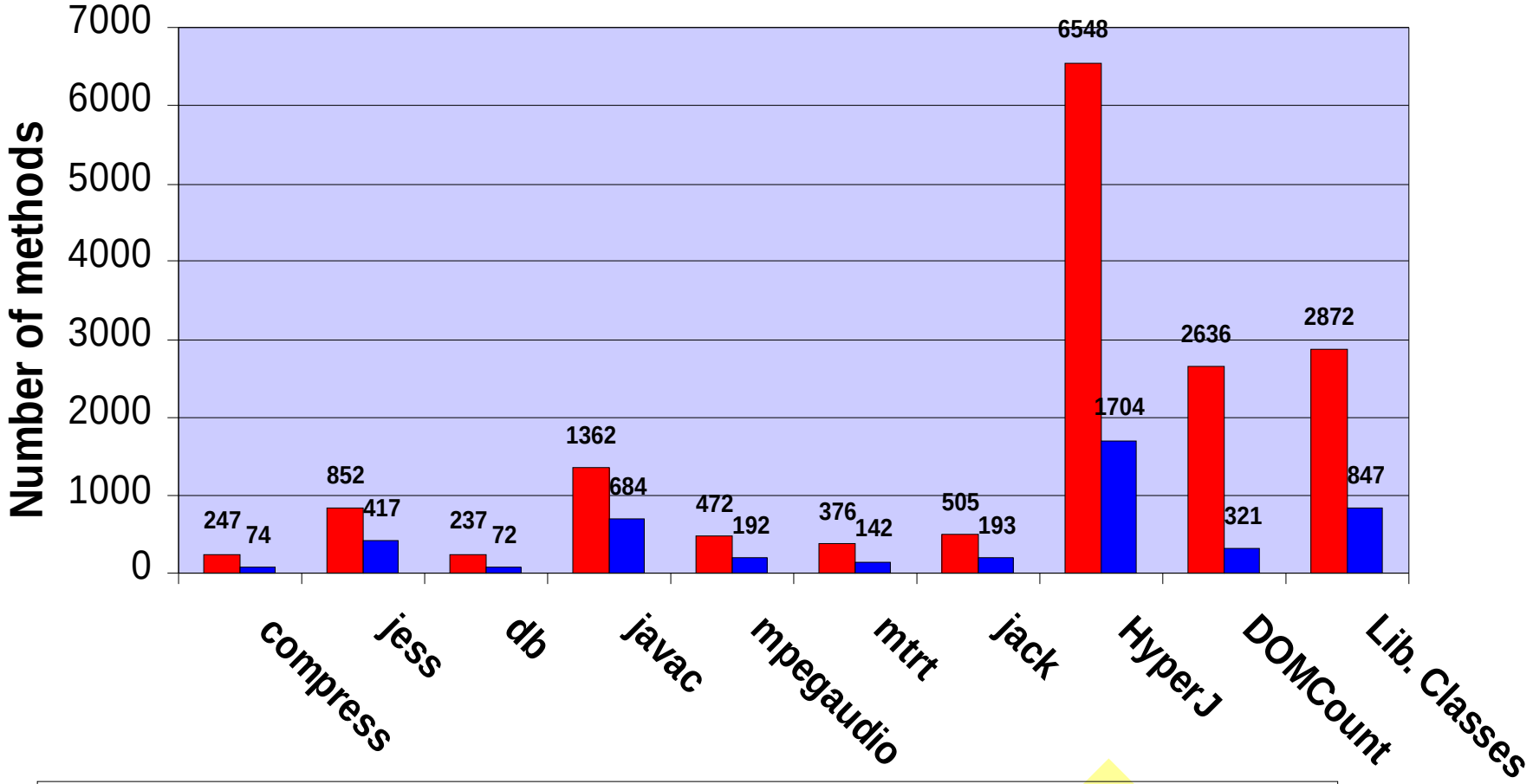


Invalidation Scenario

- Support for invalidation is necessary, to handle case when optimistic assumption proves to be incorrect

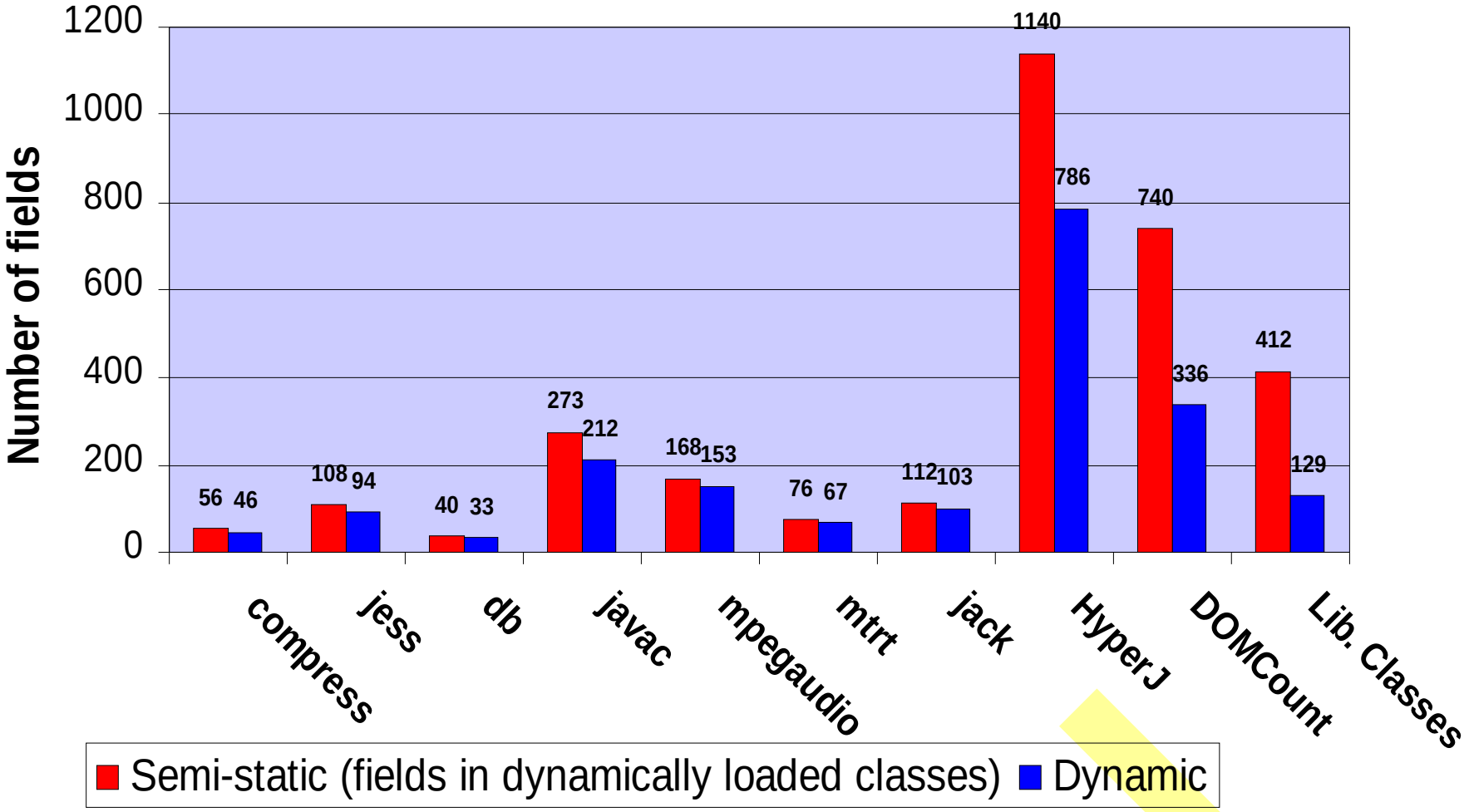


Static vs. Dynamic Application Characteristics: (Number of Methods)



Ratio of Dynamic methods to Semi-static methods ~ 12% - 50%

Static vs. Dynamic Application Characteristics: (Number of Fields containing object references)



Ratio of Dynamic fields to Semi-static fields ~ 31% - 92%

Outline

1. Motivation
2. *Dynamic Optimistic Interprocedural Type Analysis (DOIT)*
3. Immutability Analysis Opportunities for Dynamic IPA
4. Future Challenges

DOIT Phases

- Initialization
- Analysis
  - analyzes each method on first invocation
  - incorporates method summary into *Value Graph*
- Optimization
  - traverses *Value Graph* to identify types
  - uses type information in optimization
  - registers verification actions for type info used
  - registers invalidations for optimized method

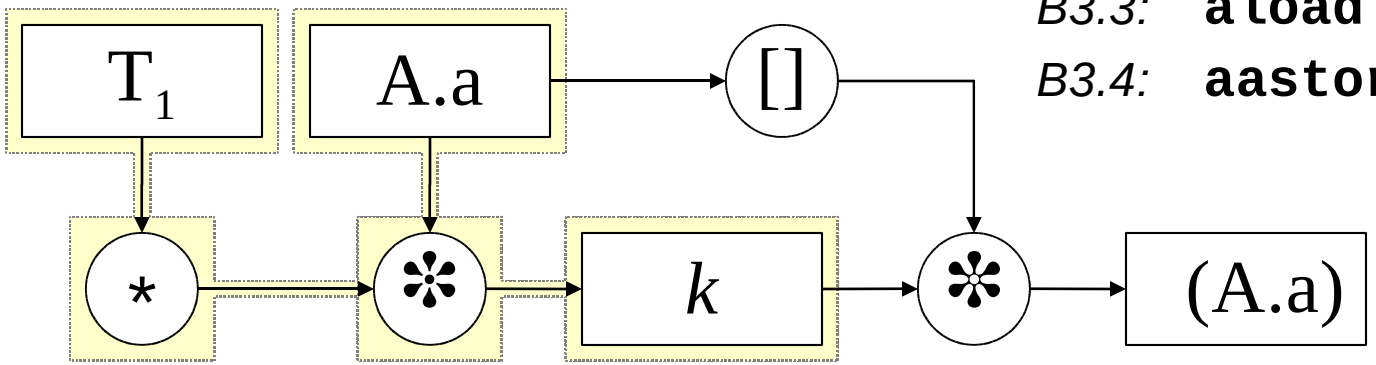
Value Graph

- Node *n* denotes a set of types, (*n*)
- *Location nodes*
  - *Local variable*
  - *Field*
  - *Array element*
  - *Constant type e.g.*, T_1
- *Operator nodes*
  - *Closure:* (*)
  - *Subscript:* ([])
  - *Union:* (✻)
  - *Intersection:* (✻)
- Edges represent flow of types
  - graph may be cyclic

Local Value Graph Example

```
     T1 M();
     ...
S1:  k=A.a;
     ...
S2:  k=M();
     ...
S3:  A.a[0]=k;
```

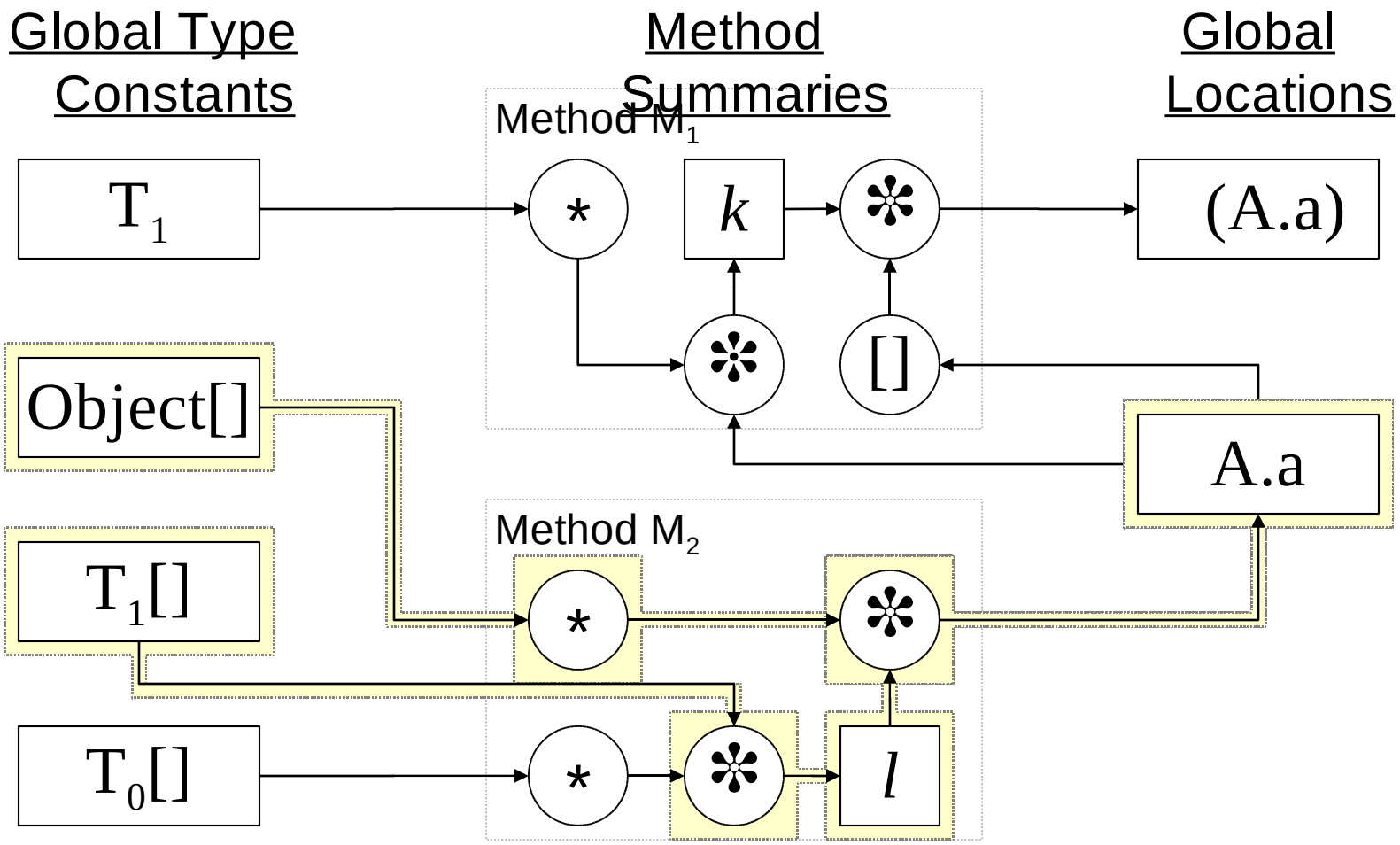
```
B1.1:  getstatic A.a
B1.2:  astore k
       ...
B2.1:  invokestatic M
B2.2:  astore k
       ...
B3.1:  getstatic A.a
B3.2:  iconst_0
B3.3:  aload k
B3.4:  aastore
```



Computing Local and Global Value Graphs

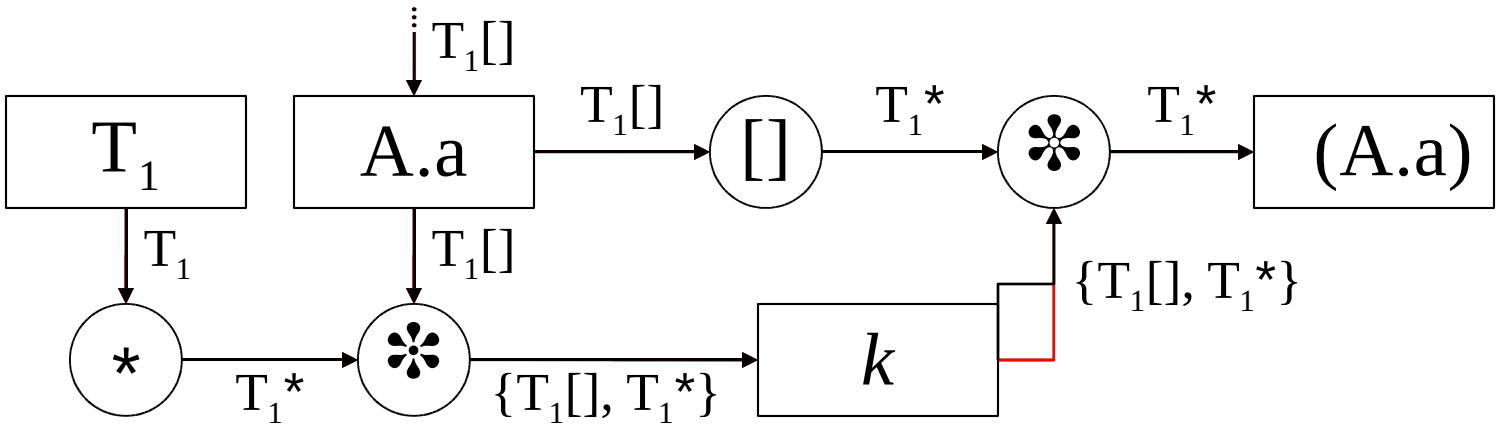
- Local Value Graph
  - Abstract interpretation of bytecodes
    - propagates types symbolically through stack
  - Represents type flow in method
- Global Value Graph
  - Local Value Graph is compressed after method is analyzed
    - Local variable nodes can be bypassed and removed
  - Local Value Graph is spliced into Global Value Graph

Global Value Graph



Computing Type Information

- For use in optimization
- Determine the type of a given location
  - on-demand traversal of the Value Graph
    - *reverse-DFS* starting at location
    - types are propagated along the edges

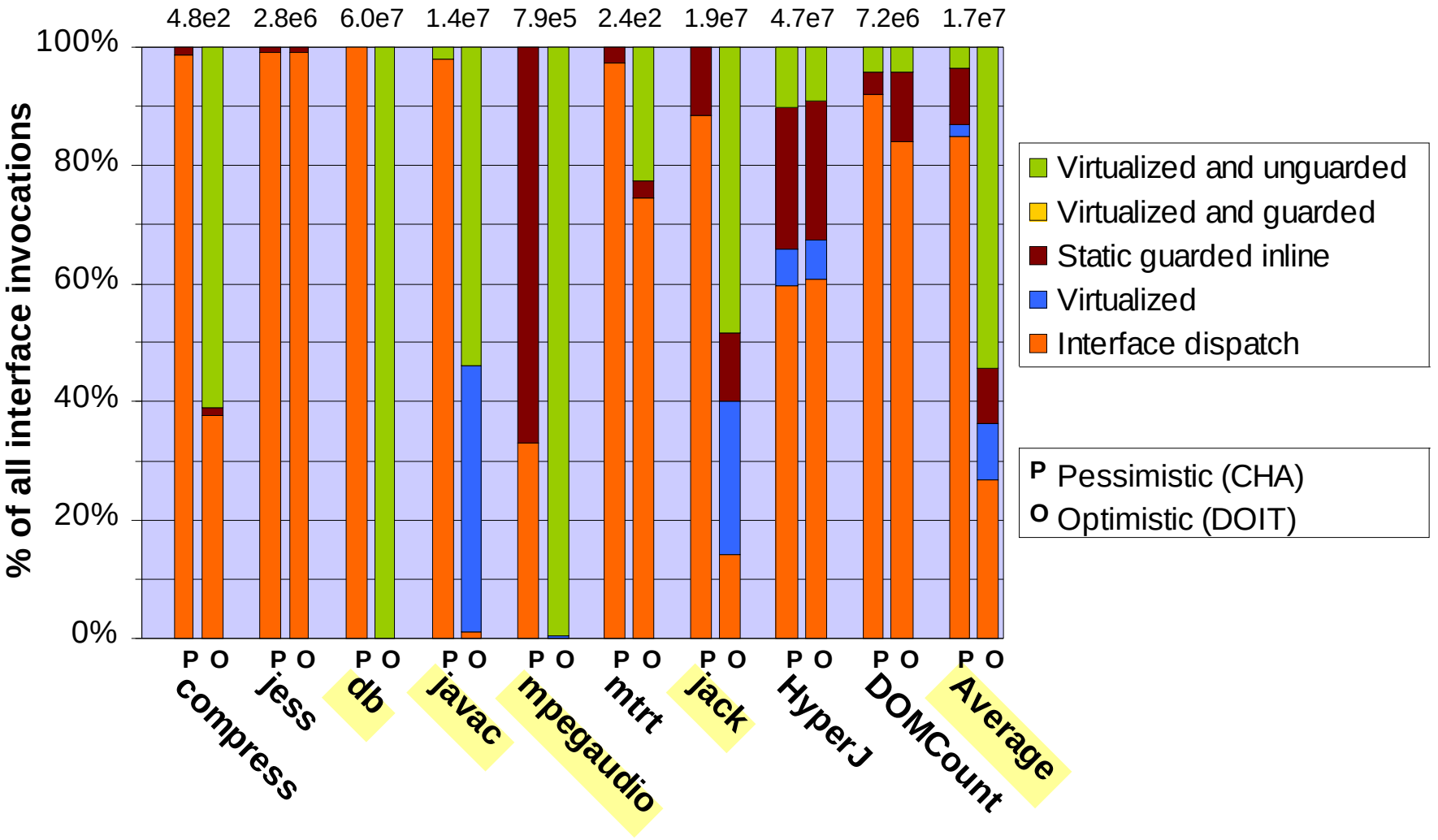


Experimental Setup

- Prototyped using *Jikes RVM*
  - type-based optimizations of calls
  - recompile after first run at highest opt level
- Benchmarks:
  - SPECjvm98, Hyper/J, Xerces (DOMCount)
- Measurements
  - Dynamic counts of virtual and interface calls
  - Execution times
  - Value Graph sizes and traversal statistics
  - Value Graph construction times

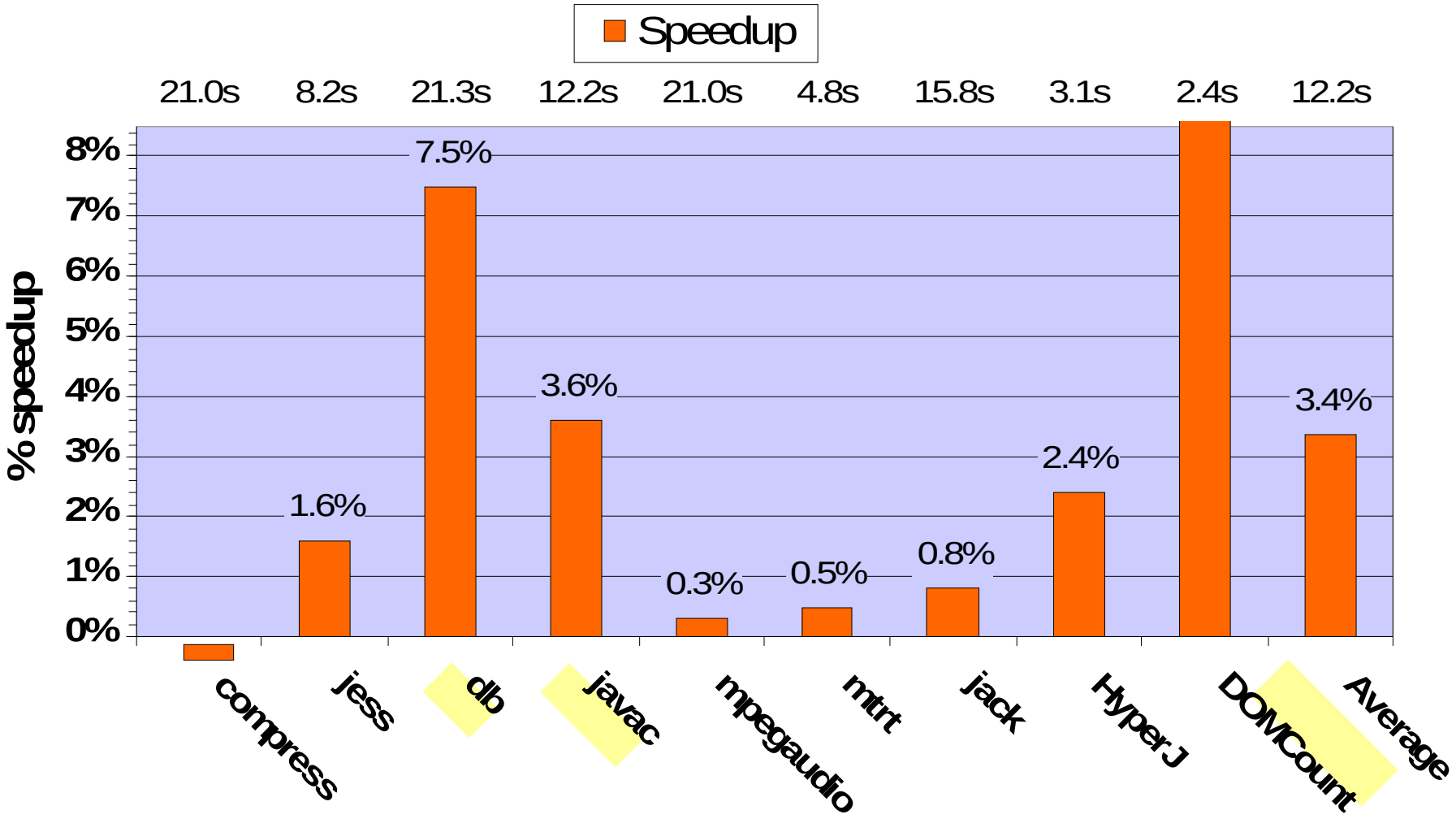
Experimental Results

Impact of DOIT Analysis on Interface Calls



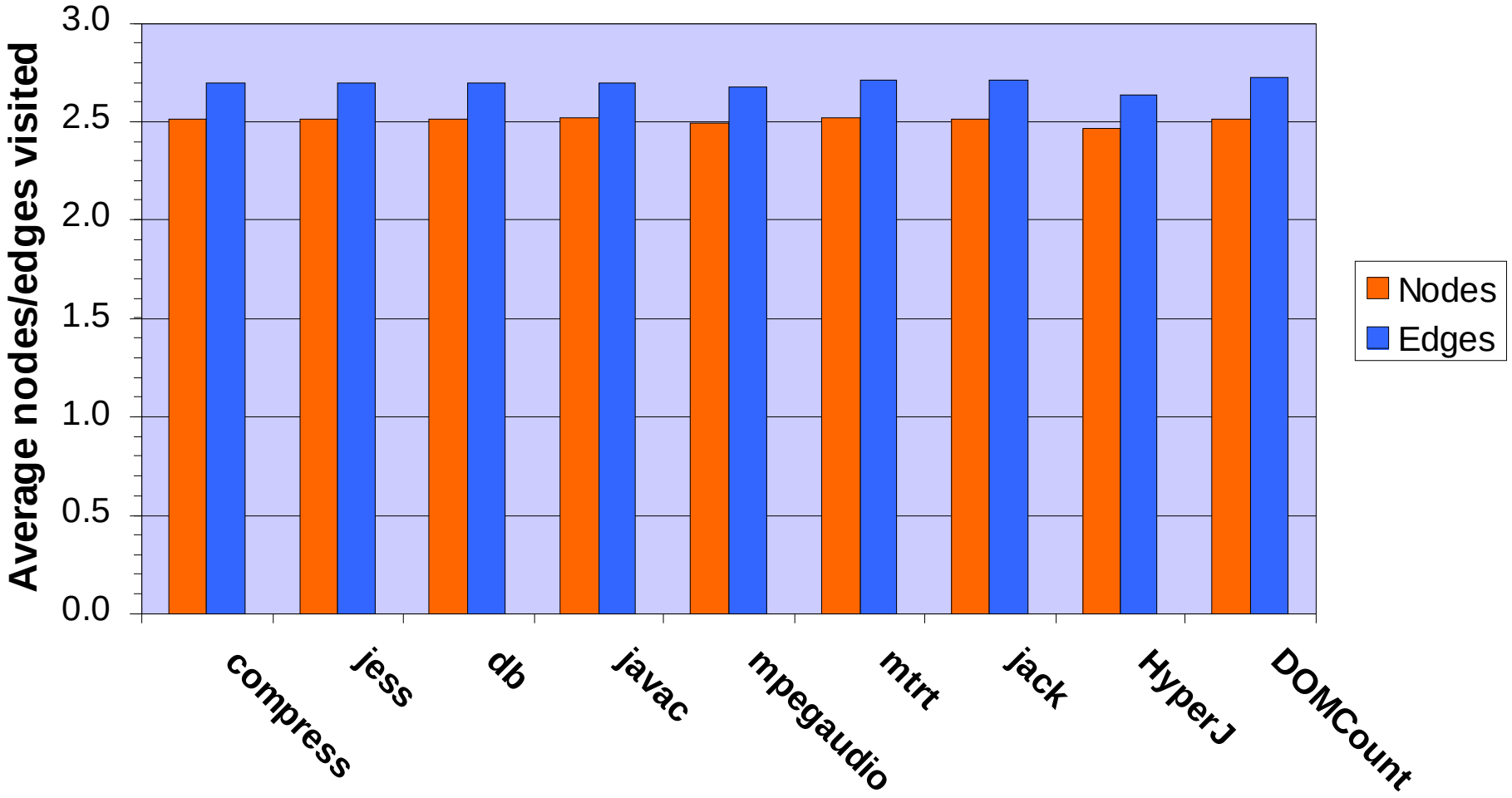
Experimental Results

Speedup from using interprocedural type info



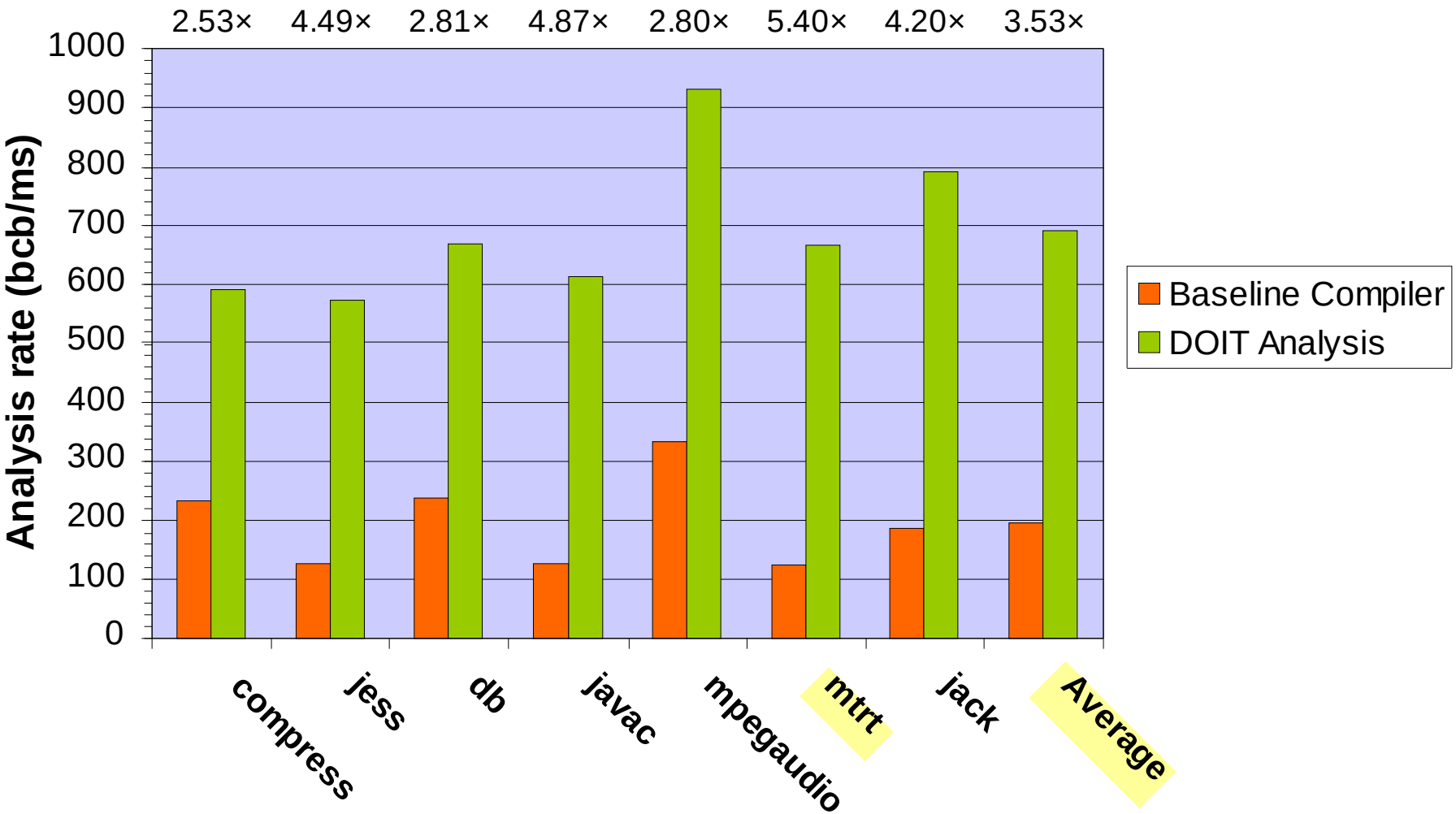
Experimental Results

Value Graph Traversal Statistics



Experimental Results

Analysis Rates (bytecode bytes/ms)



Outline

1. Motivation
2. Dynamic Optimistic Interprocedural Type Analysis (DOIT)
3. *Immutability Analysis Opportunities for Dynamic IPA*
4. Future Challenges

Immutability Analysis: Motivation

- Immutability information can be used interprocedurally to enhance:
  - Load elimination and register allocation
    - Load of immutable value cannot be changed across a procedure call
  - Array dependence analysis, pointer alias analysis
    - Target of a store instruction cannot be aliased with target of a load instruction
  - Value Numbering / CSE / PRE
    - Load of an immutable value can be treated similarly to read of an unmodified local variable to enable optimization of derived expressions (including null pointer, type checks, array bounds checks)
  - Data transformations
    - object inlining, splitting, replication, caching
  - Parallelization
    - Immutable locations cannot interfere with parallelization

Immutability Properties

- Dimensions of Immutability:
  - Lifetime
    - *e.g.*, whole program, after a certain point, in method call
  - Reachability
    - *e.g.*, reference, object, full reachability, arbitrary shape
  - Context
    - *e.g.*, all instances, instances within a method, *etc*.

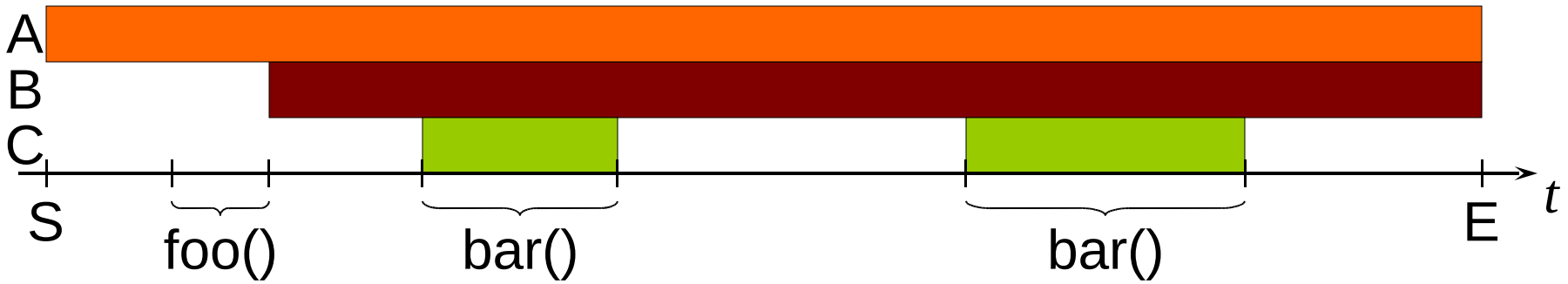
- Existing language mechanisms provide limited support for these dimensions
  - *e.g.*, Java `final`, C++ `const`

- How can immutability properties be obtained?
  1. Specified by user
  2. Inferred (optimistically) by dynamic optimization system

     Opportunity for Dynamic IPA

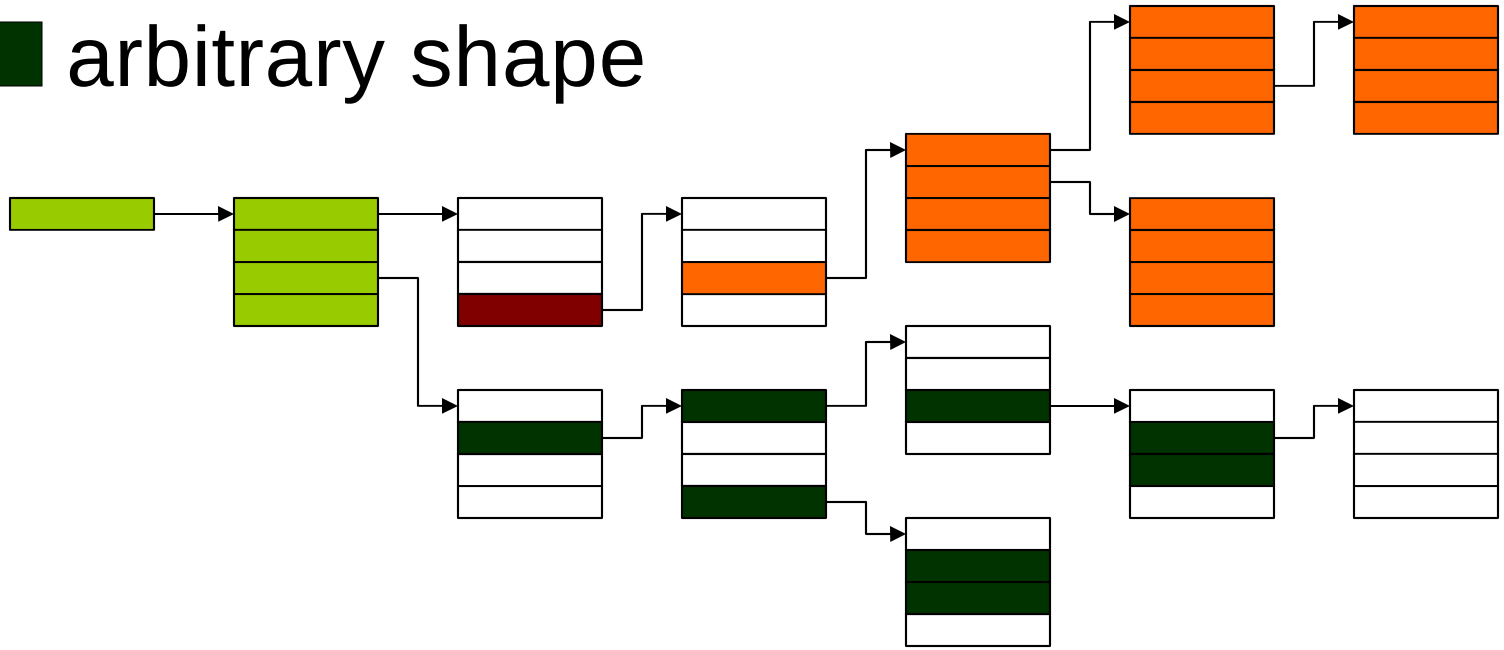
Dimensions: Lifetime

- whole program
- after a certain program point
  - *e.g.*, after an object has been initialized
- in a method call
- *etc.*



Dimensions: Reachability

- reference (`=final`)
- object
- full reachability
- arbitrary shape



Simple Example

```
class MyString {
  /* assume deep immutability for S*/
  final char[] S;
  final int count;
  . . .
  int foo( ) {
    int c1 = S[0];
    bar();
    int c2 = S[0]; // c2 must be same as c1
    return c1 + c2;
  }
}
```

Limit Study: Immutability Ratio

- Define *Immutability Ratio* as

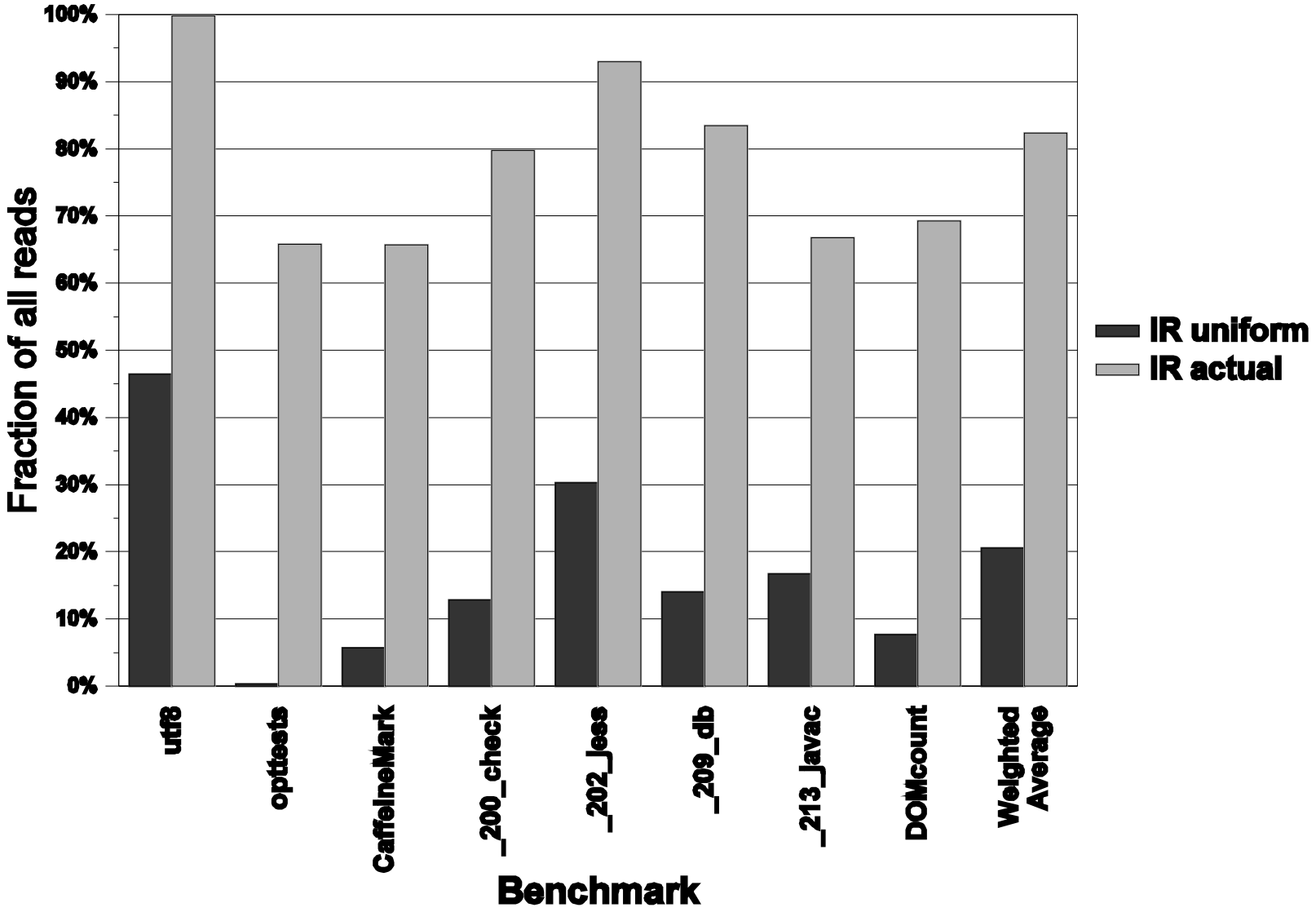
$$IR = \frac{\text{\# of read operations after last write}}{\text{total \# of read operations}}$$

- IR actual
  - Obtained by counting last write separately for each dynamic object instance
- IR uniform
  - Obtained by assuming that writes are uniformly distributed among reads
    - Hypothetical “ expected” value of IR

Limit Study: Experimental Setup

- Instrument *Jikes RVM* to generate traces
  - all read and write accesses
- Benchmarks
  - *Jikes RVM* regression tests
    - bytecodeTests, reflect, threads, utf8, opttests
  - CaffeineMark
  - SPECjvm98 (input size = 10%)
    - _200_check, _202_jess, _209_db, _213_javac
  - Xerces (DomCount)
- Goal: measure *Immutability Ratio* for benchmarks

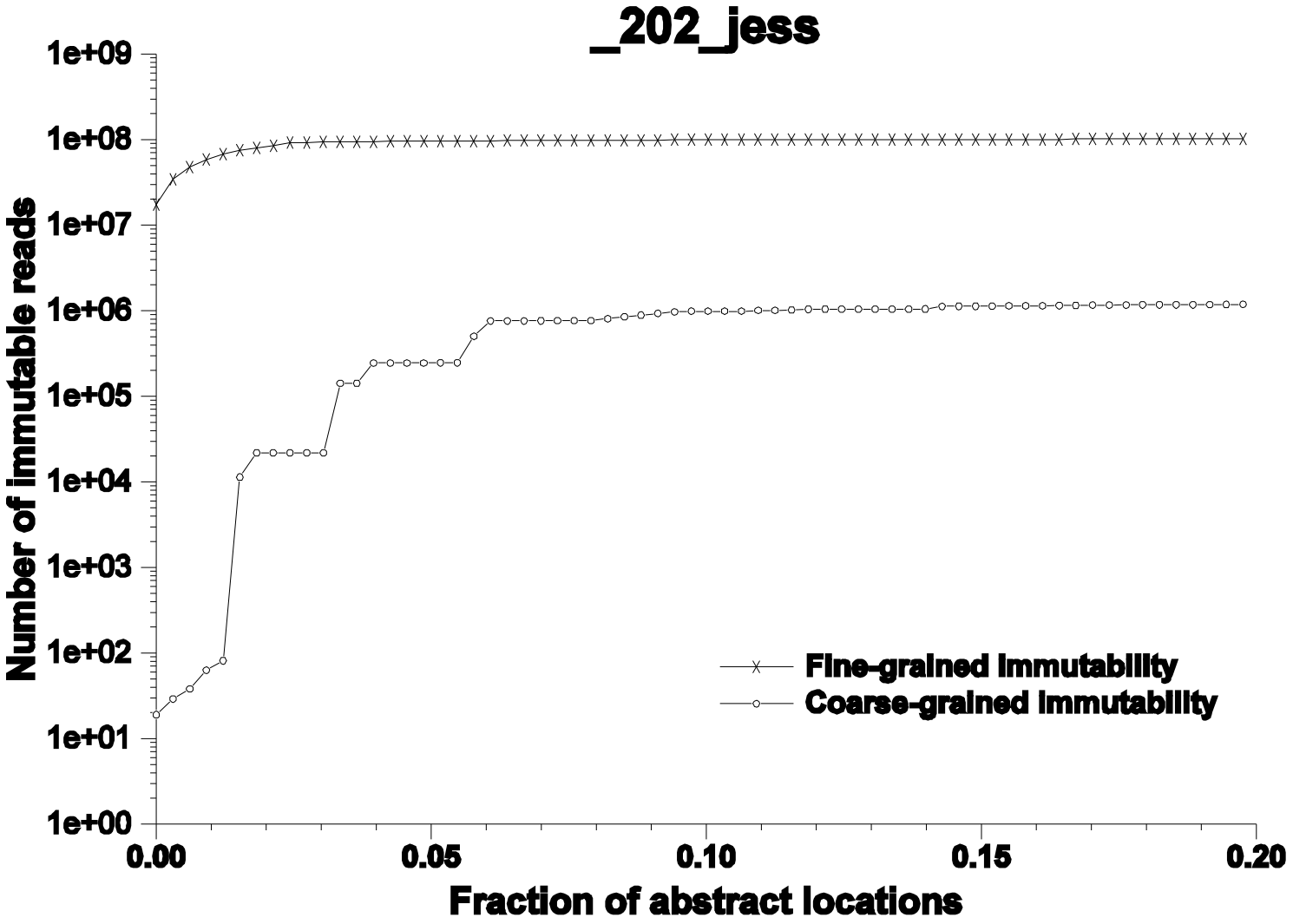
Immutability Ratios



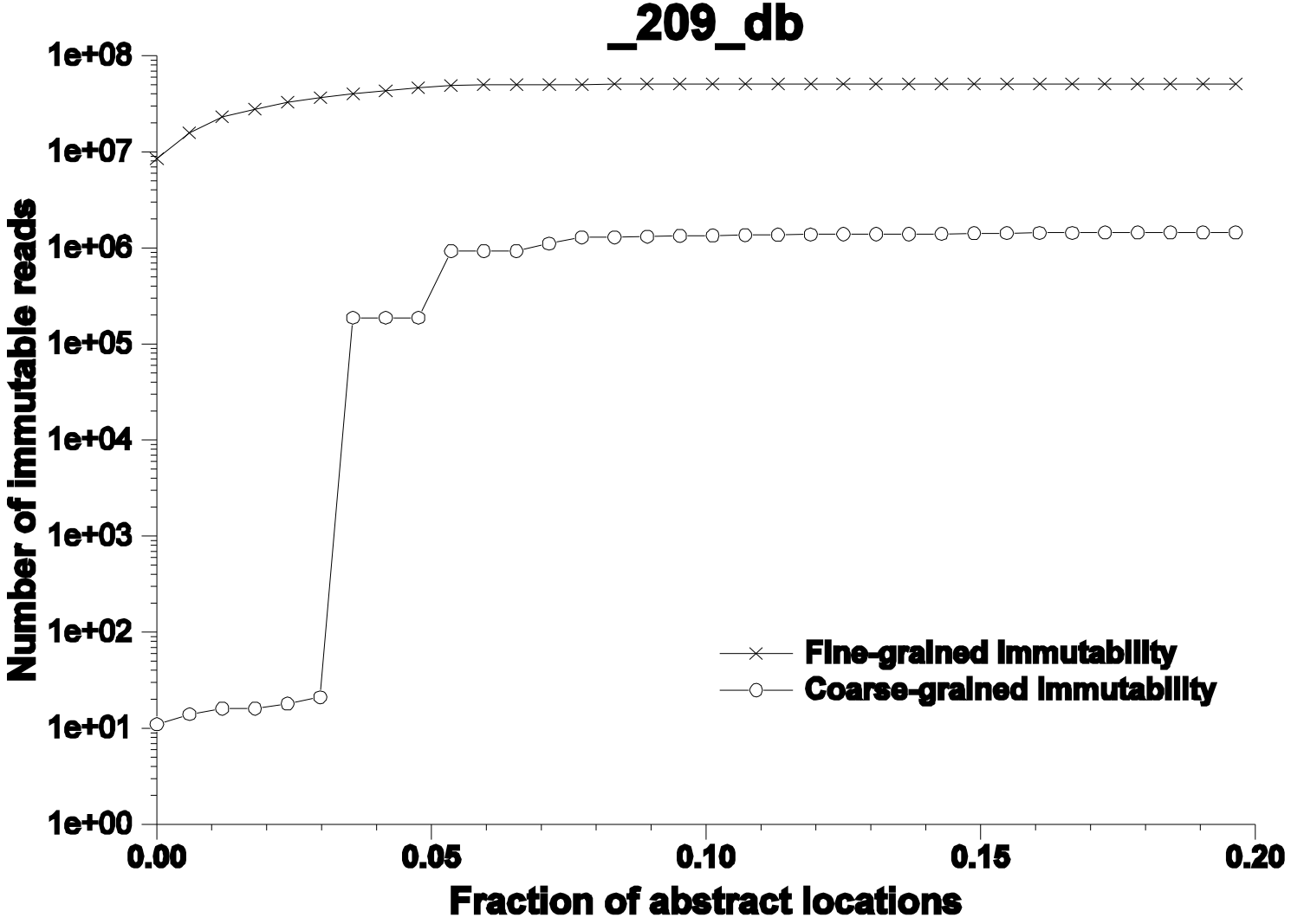
Limit Study: Abstract Locations

- Abstract location = static representative for set of dynamic locations
  - Each declared *field* is a distinct abstract location
  - Each declared *array type* is a distinct abstract location
- Coarse-grained immutability: measured by merging all dynamic instances of the same abstract location
- Goals:
  - Measure gap between fine-grained and coarse-grained immutability
  - Determine how immutable reads are distributed across abstract locations

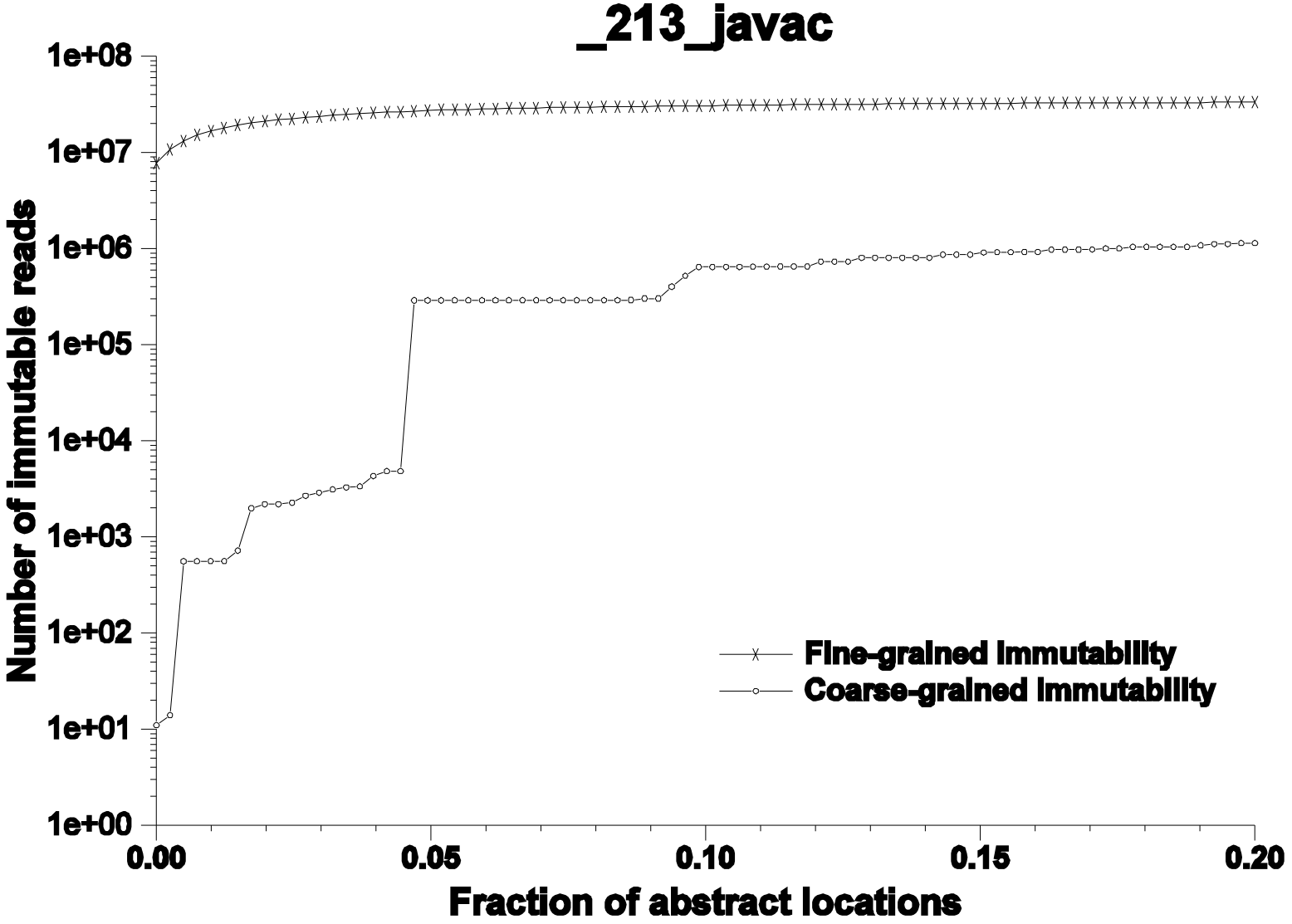
Distribution of immutable reads across abstract locations: _202_jess



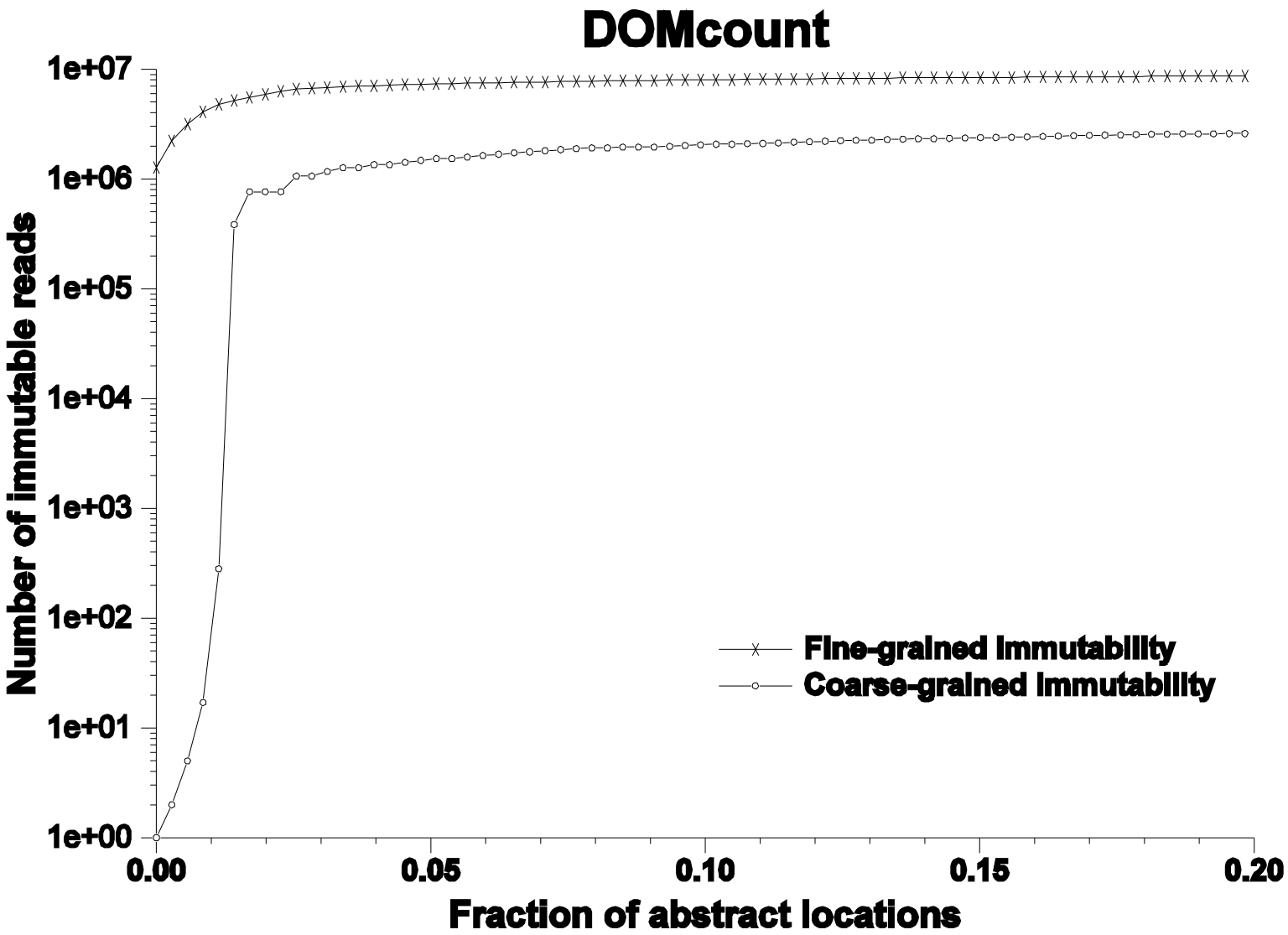
Distribution of immutable reads across abstract locations: _209_db



Distribution of immutable reads across abstract locations: _213_javac



Distribution of immutable reads across abstract locations: DOMcount



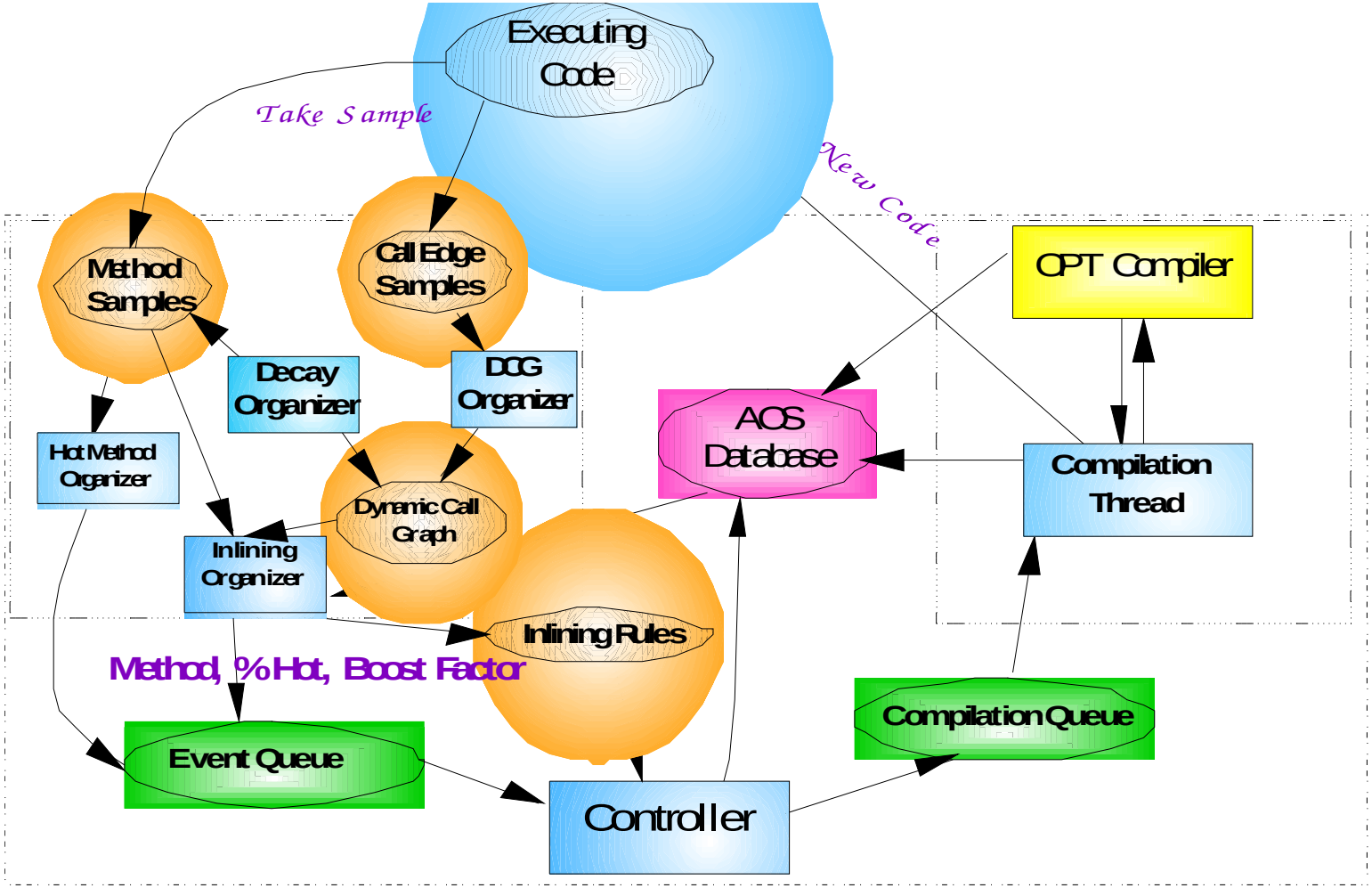
Invalidation Issues in Dynamic IPA

- **Correctness:** must always be possible to undo the optimization
  - need recovery procedure; may limit scope of optimization
- **Efficiency:** cost; depends on
  - what optimization is performed, *e.g.*,
    - preexistence based inlining only needs recompilation
    - dead store elimination needs on-stack replacement
    - object inlining needs data structure rewriting
  - when optimization is performed
    - delaying optimization may avoid need for invalidation

Integrating Dynamic IPA into Adaptive Optimization Framework

- Invalidation cost supplied to adaptive system
  - which uses cost-benefit model
- Optimization considered worthwhile if cost of invalidation less than potential benefit
  - invalidation cost may vary dynamically
- Optimizations may be more profitable for long-running programs

Adaptive Optimization System w/ Adaptive Inlining



Outline

1. Motivation
2. Dynamic Optimistic Interprocedural Type Analysis (DOIT)
3. Immutability Analysis Opportunities for Dynamic IPA
4. *Future Challenges*

Future Challenges

- Integrating Dynamic IPA into Adaptive Optimization and Invalidation
- Automatic inference of Dynamic IPA properties of interest
- Application of Dynamic IPA to verification
- Refining granularity of Dynamic IPA from methods to basic blocks