## Advanced Topics

### Last Time

- Experimental Methodology

### Today

- What's a managed language?
- Alias Analysis - dealing with pointers
  - Focus on statically typed managed languages
  - Method invocation resolution



## Alias Analysis

What is pointer analysis?

- For umanaged languages like C, C++, and managed languages like C#, Java, Modula-3
- Goal: given two memory references, *m* and *n*, and a program point, *p*, can *m* and *n* access the same memory location?

Pointer analysis is an important problem

- Compiler optimizations require alias information
- Error detection
- Program slicing/debugging
- Data structure *shape* analysis: is it a tree?
- Object inlining, object colocation, synchronization elimination, etc.

## Pointer analysis is hard

- Undecidable in general [Landi 92]
- Many approximations
  - Near linear, but imprecise [Steensgaard 97]
  - Double exponential shape analysis [Sagiv 98]
- Why haven't we solved this problem? [Hind 01]

$\Rightarrow$ No one algorithm satisfies all client needs

Sample Client driven results:

- Error detection: needs high precision, but a demand driven precision can provide precision at low cost [Guyer/Lin 03]
- Scalar optimizations: do not need high precision [Chowdhury et al. 03, Diwan et al. 98/01]
- Object colocation: a fast, simple, and *unsound* analysis is effective [Guyer/McKinley 04]



## Taxonomy

Interprocedural data-flow analysis

- Interprocedural or intraprocedural? Do we compute the effects of calls?
- Context-sensitive or insensitive? Do we take into account the calling context?
- Flow-sensitive or insensitive? Do we take into account control flow

More precise means more expensive.

**Address-taken:** Flow and context insensitive. Intraprocedural or interprocedural. Every variable that may be aliased goes in a single set. Single pass.

**Steensgaard:** Flow and context insensitive. Interprocedural. Uses union-find to create sets of aliases. Single pass.

**Anderson:** Flow and context insensitive. Interprocedural. Creates sets of aliases. Iterative on the CFG.

**Burke:** Flow-insensitive and context sensitive. Interprocedural. Iterative on the CFG and Call graph.

**Choi et al.:** Flow and context sensitive. Interprocedural. Iterative on the CFG and Call graph.

## Example: Simple Alias Analysis

Language context

- Modula-3: a statically-typed, object-oriented programming language.
- Techniques apply directly to Java and C#, and may apply to C++.

Clients: scalar optimizations and method resolution

Alias analysis: Type-based alias analysis

Using types is

- Fast and surprisingly effective for these clients

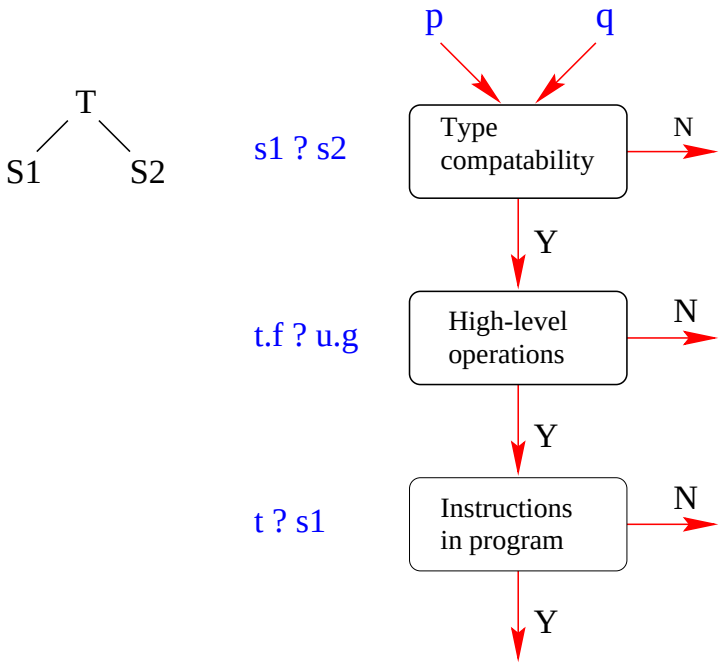
Evaluation: A limit analysis computes a runtime upper bound on effectiveness of these compiler optimizations.

A. Diwan, K. S. McKinley, and J. E. B. Moss, Using Types to Analyze and Optimize Object-Oriented Programs, ACM Transactions on Programming Languages and Systems, 23(1): 30-72, January 2001.



## Type-Based Alias Analysis

Use language types to disambiguate memory references



## Strengths and Weakness of Analysis

| | Types | Instructions |
|---|---|---|
| Speed | O(Instructions) | O(Instructions$^d$) |
| Unsafe programs | No | Yes |
| Precision | Good | Better |

Type-based alias analysis is fast. Is it effective?

### Evaluation

| | Static | Run time | Upper-bound |
|---|---|---|---|
| Coverage | √ | | √ |
| Run-time impact | | √ | |
| Compare analyses | √ | √ | ∼ |
| Compare to best | | | √ |
| Deficiencies | | | √ |

Lots of alias analysis work uses static metrics and does not consider the clients.

Client 1: Redundant Load Elimination (RLE)



## Modula-3 Benchmarks

| Name | Lines | Instructions | % Heap loads | % Other loads |
|---|---|---|---|---|
| format | 395 | 1,879,195 | 10 | 17 |
| dformat | 602 | 1,442,541 | 9 | 19 |
| write-pickle | 654 | 1,614,437 | 13 | 16 |
| k-tree | 726 | 50,297,517 | 10 | 21 |
| slisp | 1,645 | 11,462,791 | 27 | 9 |
| pp | 2,328 | 45,779,402 | 11 | 19 |
| m2tom3 | 10,574 | 50,894,990 | 8 | 28 |
| m3cg | 16,475 | 5,636,004 | 8 | 21 |

## Static Evaluation

Average number of potential aliases for each reference.

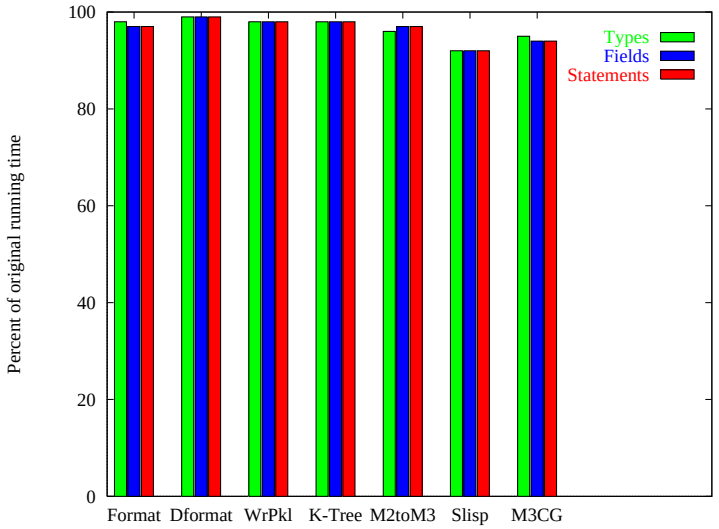
| | Declarations | Fields | Statements |
|---|---|---|---|
| Interprocedural | 4.7 | 3.4 | 3.4 [0.3 to 21] |
| Intraprocedural | 54.1 | 12.7 | 12.7 [2 to 28] |

- Bad news!
- Much higher accuracy with interprocedural analysis.
- Flow-insensitive data-flow analysis contributes very little to the precision of a type-based alias analysis.

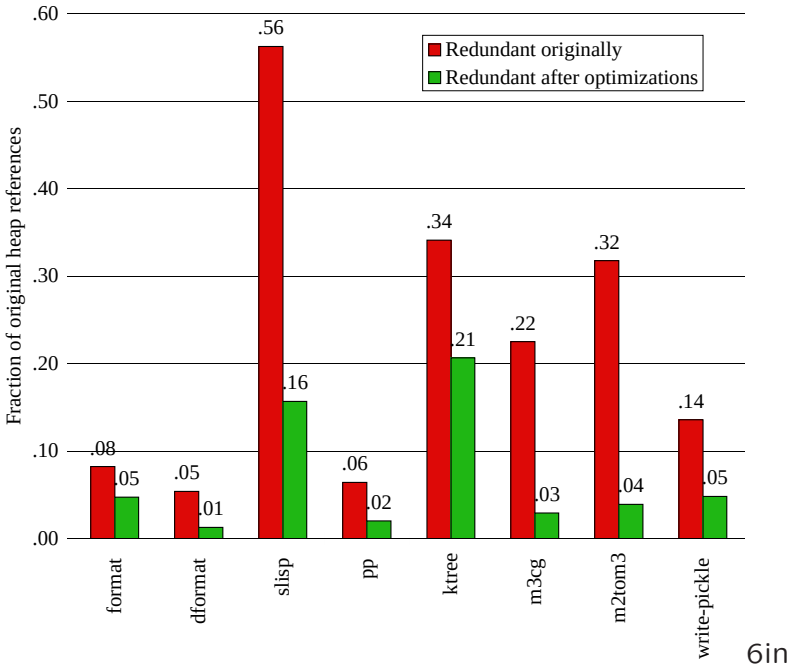


## Run-Time Impact of Alias Analysis

Measure the performance improvement due to RLE.



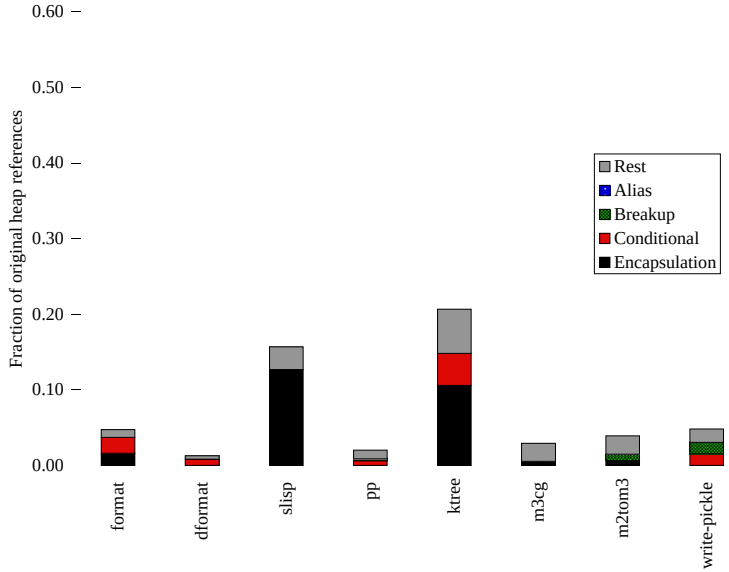
## Comparing Alias Analysis to an Upper-bound




6in



## What's Left to Disambiguate?

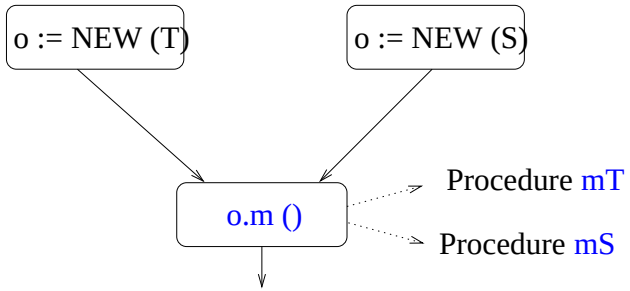


## Bottom Line

- All metrics are useful.
- In this case, static measures are misleading.
- Alias analysis & RLE yield modest improvements for our benchmarks.
- There is not much room for improvement in the alias analysis for RLE.
- RLE acts locally.



## Client 2: Resolving Method Invocations




- A polymorphic method invocation may call more than one procedure at run time.
- A monomorphic method invocation always calls the same procedure at run time.

## Resolving Method Invocations

Method invocations

- are more expensive than direct calls
- inhibit compiler optimizations

We want to resolve monomorphic method invocations: replacing method invocations with direct calls.

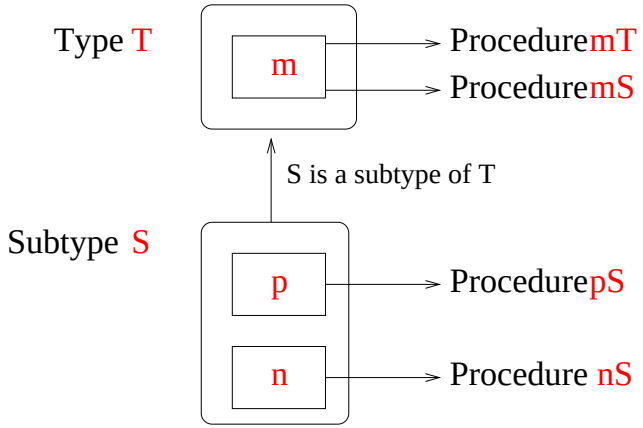
Full resolution is undecidable so any resolution technique must be conservative and assume polymorphic.

| Technique | Complexity | Source |
|---|---|---|
| Type hierarchy (Fernandez, Dean et al.) | $O(\|Types\| * \|Methods\|)$ | borrowed |
| Intraprocedural type propagation | $O(\sum_p n_p * v_p)$ | simplified |
| Aggregate | $O(\sum_p n_p * v_p)$ | new |
| Interprocedural type propagation | $O(p \sum_p n_p * v_p)$ | simplified |

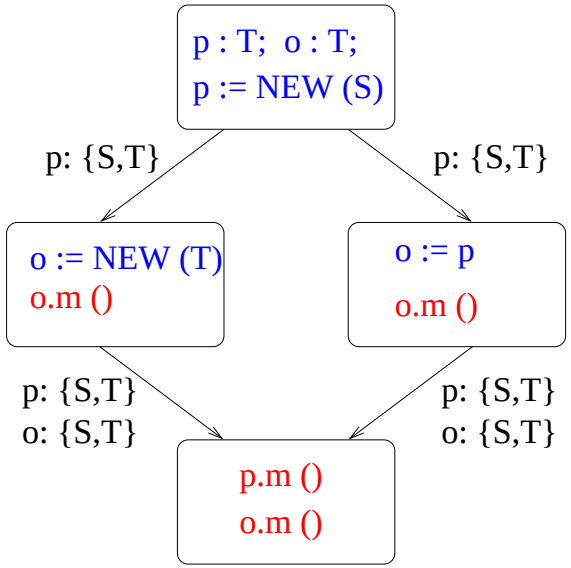


## Type Hierarchy Analysis

Bounds the procedures a method invocation may call by examining the type hierarchy declarations for method overrides.



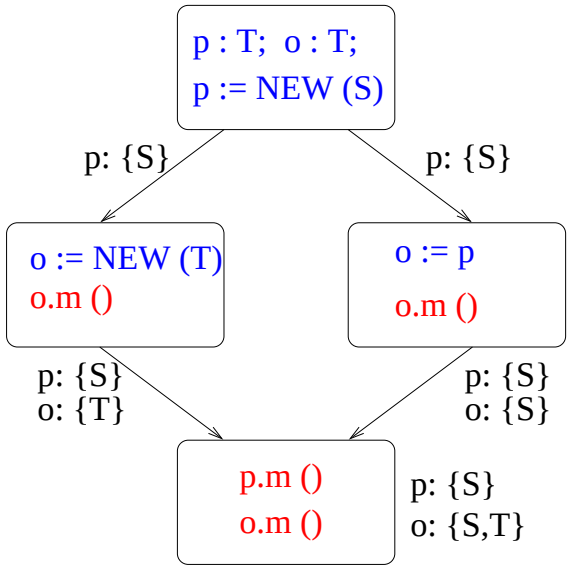
## Type Hierarchy Analysis




Declarations set types. Analysis ignores control flow.



## Intraprocedural Type Propagation: Data-Flow Analysis




Allocation, type discrimination, and assignment affect types

## Aggregate Analysis

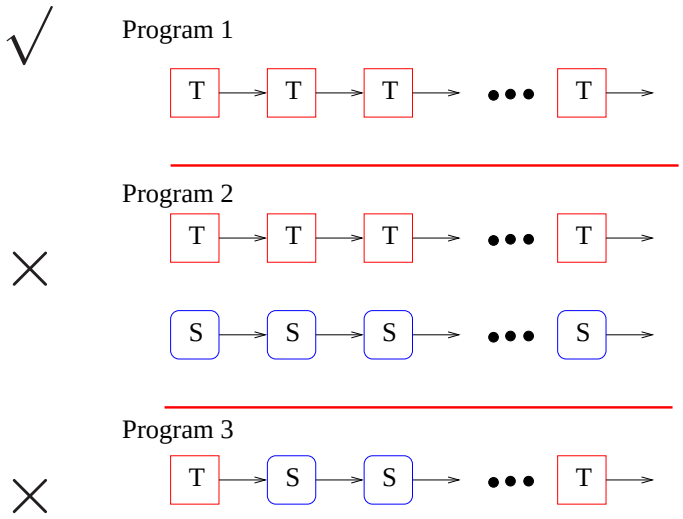
Use type-based alias analysis to disambiguate pointer references. Type-based alias analysis

- merges all instances of an object or record type, and
- ignores all control flow.

Aggregate analysis finds monomorphic uses of general data structures.

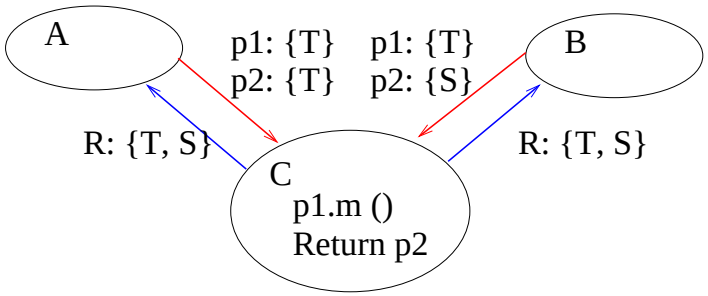


## Aggregate Analysis Example



## Interprocedural Type Propagation

- Merge parameter information for all callers (context insensitive)
- Merge return information for all callers at a call site
- Reanalyze when more refined information becomes available
- Update call graph as necessary

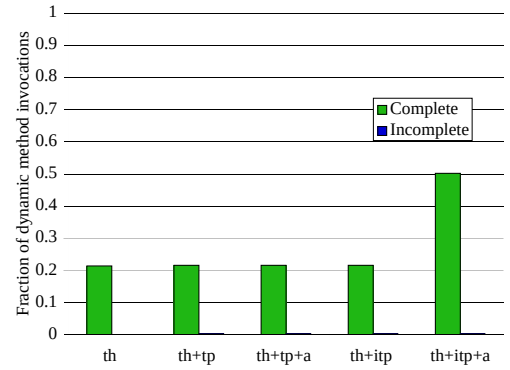




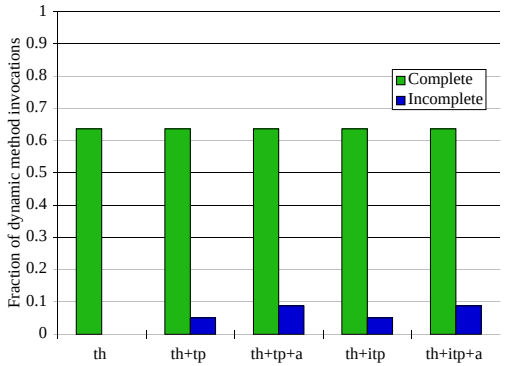

## Modula-3 Benchmarks

| Name | Lines | Static Method Inv. Total | Mono | Rslv | Dyn MI |
|---|---|---|---|---|---|
| format | 395 | 37 | 26 | 26 | 47,064 |
| dformat | 602 | 95 | 77 | 77 | 30,775 |
| k-tree | 726 | 13 | 5 | 4 | 714,619 |
| slisp | 1645 | 223 | 94 | 94 | 67,253 |
| pp | 2328 | 24 | 1 | 1 | 458 |
| dom | 6186 | 222 | 136 | 128 | 12,377 |
| postcard | 8214 | 293 | 129 | 87 | 3,076 |
| m2tom3 | 10574 | 1821 | 1110 | 1110 | 19,886,862 |
| m3cg | 16475 | 1808 | 295 | 213 | 32,850 |
| trestle | 28977 | 430 | 22 | 9 | 10,756 |
| **Total** | | 4966 | 1895 | 1749 (92%) | |

## Effectiveness




m3cg

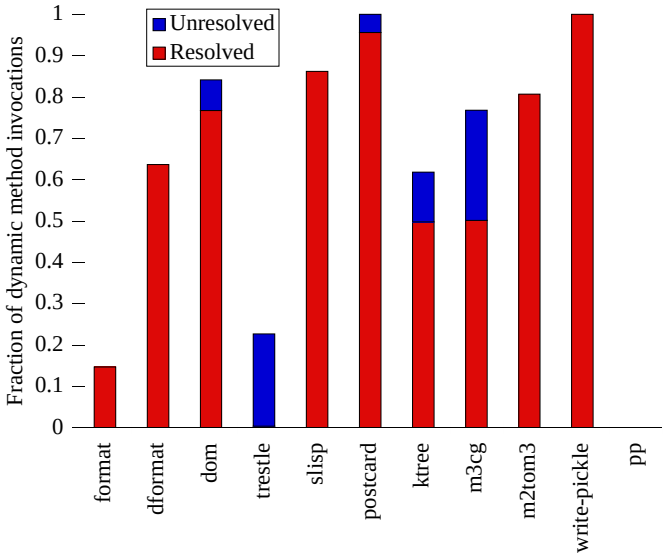



DFormat



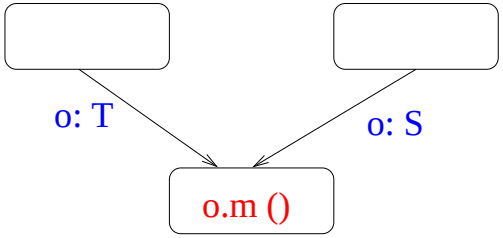
## Upper-Bound Comparison

Compare results with monomorphic method invocations at run time



## Cause Evaluation

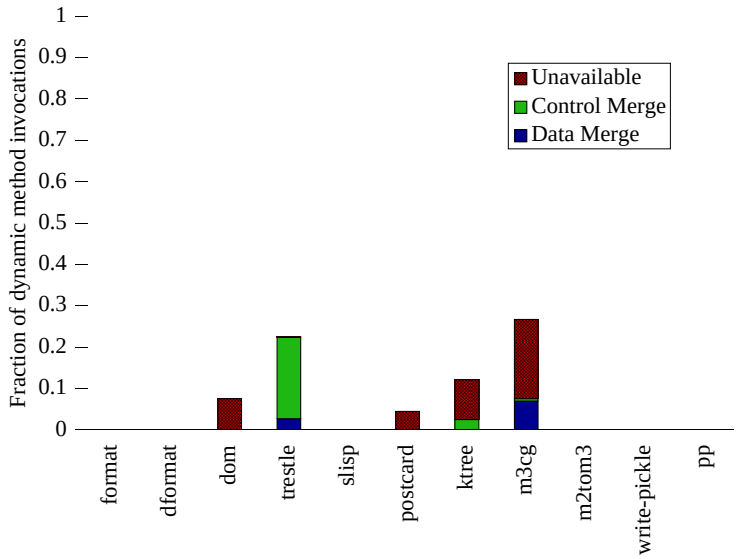
Find the source of loss of type information



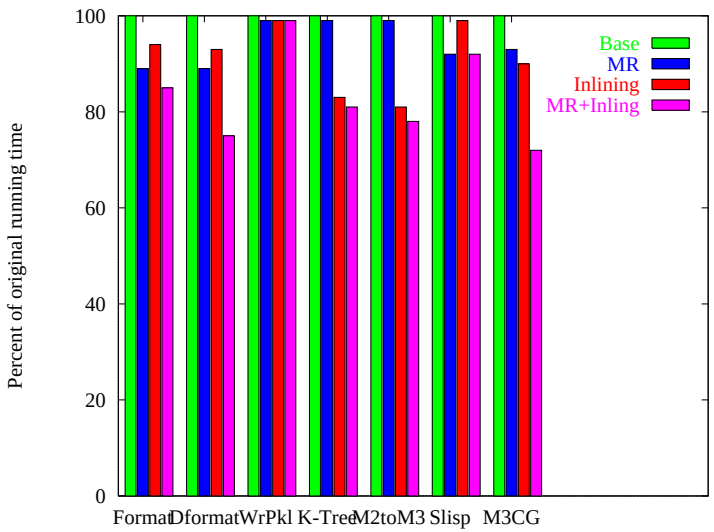

| Source | Solution |
|---|---|
| Control merge | Context sensitive analysis |
| Data merge | More powerful alias analysis |
| Unavailable | Analyze libraries |



## Cause of analysis failure



## Run-Time Impact of Resolving Method Invocations



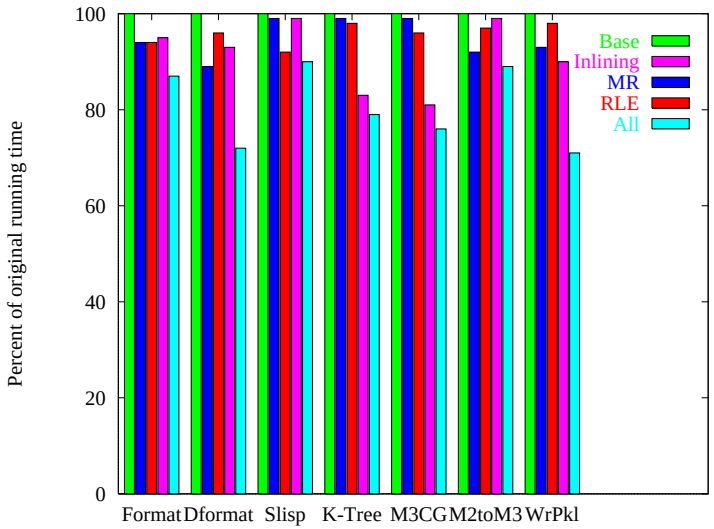



## Bottom Line

- Method resolution improves performance up to 11%.
- Inlining & method resolution improves performance up to 28% for these benchmarks.
- Different analyses are effective under different situations; thus we need a range of analyses.
- There is not much room for improvement here.

$\Longrightarrow$ We can eliminate method invocation overhead for monomorphic call sites.

## Cumulative Impact of Optimizations






## Summary

Type-based analyses are effective at

- reducing the overhead of linked structures, and
- reducing the overhead of method invocations,

Compiler optimizations need static, dynamic, and limit evaluation.

### Open Questions

- Further evaluation of type-based alias analysis.
- Applying limit analysis to other optimizations.

## Next Time

Interprocedural optimization

Grove & Torczon, Interprocedural Constant Propagation: A Study of Jump Function Implementations, ACM Conference on Programming Language Design and Implementation, pp. 90-99, Albuquerque NM, June 1993.