Lecture 27

CIS 341: COMPILERS

Announcements

- HW 7: Optimization & Experiments
  - Post your benchmark programs early (i.e. tonight!)
  - Due: Tomorrow April 29th

- Final Exam:
  - Thursday, May 7th
  - 9:00AM
  - Moore 216

Vellvm

VERIFYING COMPILER TRANSFORMATIONS

LLVM_{ND} Operational Semantics

- Define a transition relation:

$$f \vdash \sigma_1 \longmapsto \sigma_2$$

  - f is the program
  - σ is the program state: pc, locals(δ), stack, heap
- Nondeterministic
  - δ maps local `%uids` to sets.
  - Step relation is nondeterministic
- Mostly straightforward (given the heap model)
  - Another wrinkle: phi-nodes executed atomically

Operational Semantics

	Small Step	Big Step
Nondeterministic	$\boxed{\text{LLVM}_{ND}}$	
Deterministic		

Deterministic Refinement

	Small Step	Big Step
Nondeterministic	LLVM_{ND}	
Deterministic	LLVM_{D}	

Instantiate ‘undef’ with default value (0 or null) ⇒ deterministic.

Big-step Deterministic Refinements

	Small Step	Big Step
Nondeterministic	$LLVM_{ND}$ $\uplus$	
Deterministic	$LLVM_{Interp} \approx LLVM_D$	

Bisimulation up to "observable events":

- external function calls

Big-step Deterministic Refinements

	Small Step	Big Step
Nondeterministic	LLVM_{ND} ⩛	
Deterministic	$\mathrm{LLVM}_{Interp} \approx \mathrm{LLVM}_{D}$	$\gtrsim \mathrm{LLVM}^{*}_{DFn} \gtrsim \mathrm{LLVM}^{*}_{DB}$

Simulation up to "observable events":

- useful for encapsulating behavior of function calls
- large step evaluation of basic blocks

[Tristan, et al. *POPL '08*, Tristan, et al. *PLDI '09*]

Strategy for Proving Optimizations

- Decompose the program transformation into a sequence of “micro” transformations
  - e.g. code motion =
    1. insert “redundant” instruction
    2. substitute equivalent definitions
    3. remove the “dead” instruction

- Use the backward simulations to show each “micro” transformation correct.
  - Often uses a *safety property*
  - Safety: establish an invariant of the execution of the program

- Compose the individual proofs of correctness

Safety Properties

- A well-formed program never accesses undefined variables.

$$\text{If } \vdash f \text{ and } f \vdash \sigma_0 \longmapsto^* \sigma \text{ then } \sigma \text{ is not stuck.}$$

$\vdash f$	program f is well formed
σ	program state
$f \vdash \sigma \longmapsto^* \sigma$	evaluation of f

- *Initialization*:

$$\text{If } \vdash f \text{ then } wf(f, \sigma_0).$$

- *Preservation*:

$$\text{If } \vdash f \text{ and } f \vdash \sigma \longmapsto \sigma' \text{ and } wf(f, \sigma) \text{ then } wf(f, \sigma')$$

- *Progress*:

$$\text{If } \vdash f \text{ and } wf(f, \sigma) \text{ then } f \vdash \sigma \longmapsto \sigma'$$

Safety Properties

- A well-formed program never accesses undefined variables.

If $\vdash f$ and $f \vdash \sigma_0 \longmapsto^* \sigma$ then σ is not stuck.

$\vdash f$	program f is well formed
σ	program state
$f \vdash \sigma \longmapsto^* \sigma$	evaluation of f

- *Initialization*:

If $\vdash f$ then $wf(f, \sigma_0)$.

- *Preservation*:

If $\vdash f$ and $f \vdash \sigma \longmapsto \sigma'$ and $wf(f, \sigma)$ then $wf(f, \sigma')$

- *Progress*:

If $\vdash f$ and $wf(f, \sigma)$ then $f \vdash \sigma \longmapsto \sigma'$

Well-formed States

```
entry:
  r0 = ...
  r1 = ...
  r2 = ...

  br r0 loop exit
```

```
loop:
  r3 = φ[0;entry][r5;loop]
  r4 = r1 x r2
  r5 = r3 + r4
  r6 = r5 ≥ 100
  br r6 loop exit
```

```
exit:
  r7 = φ[0;entry][r5;loop]
  r8 = r1 x r2
  r9 = r7 + r8
  ret r9
```

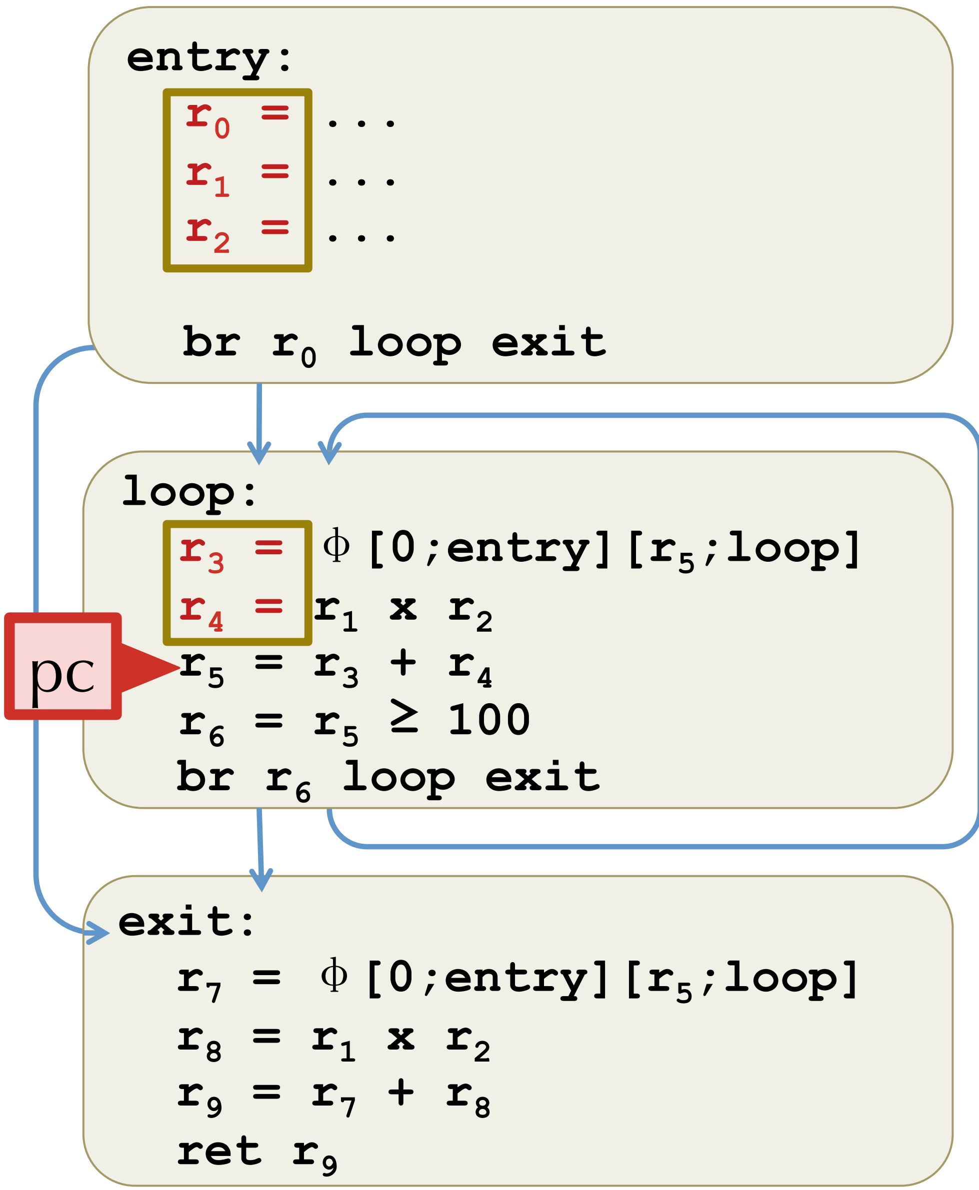
pc

State σ is:

pc = program counter

δ = local values

Well-formed States (Roughly)



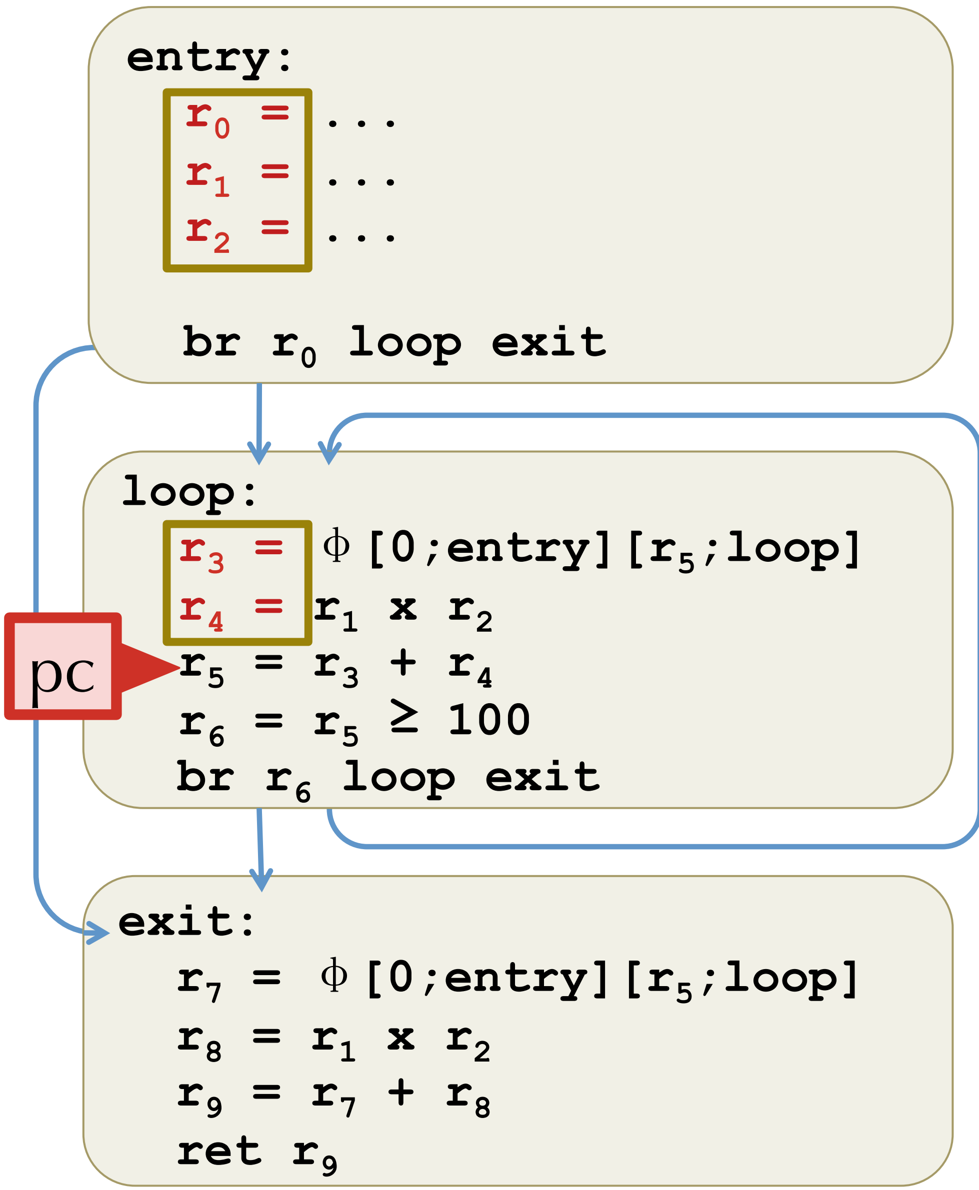
State σ is:

pc = program counter

δ = local values

sdom(f,pc) = variable defns. that *strictly dominate* pc.

Well-formed States (Roughly)



State σ contains:

pc = program counter

δ = local values

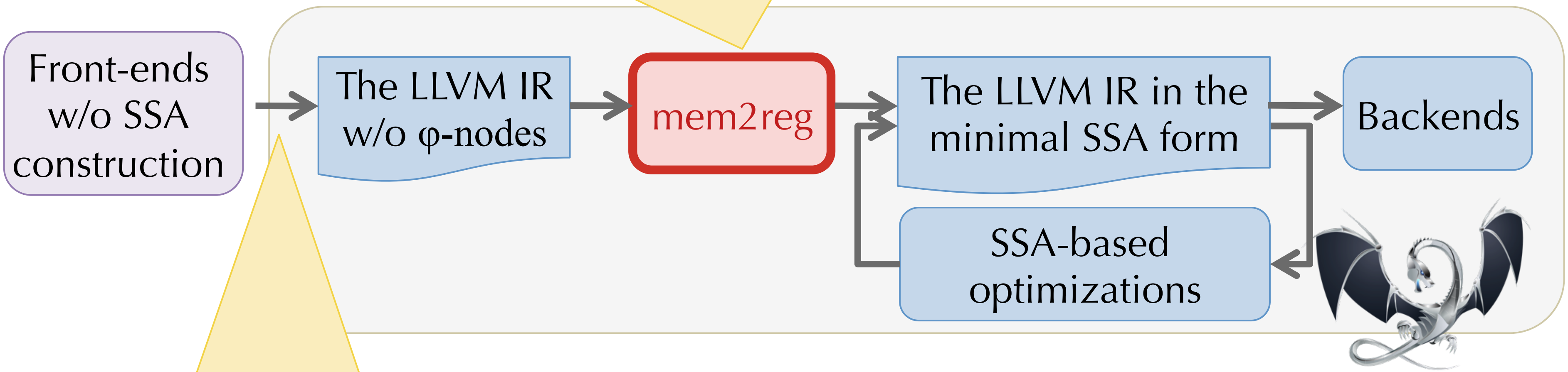
sdom(f,pc) = variable defns. that *strictly dominate* pc.

wf(f, σ) =
$\forall r \in$ sdom(f,pc). $\exists v.$ $\delta(r) = \lfloor v \rfloor$

"All variables in scope are initialized."

mem2reg in LLVM (part of SROA)

- Promote stack allocas to temporaries
- Insert minimal φ-nodes



- imperative variables ⇒ stack allocas
- no φ-nodes
- trivially in SSA form

mem2reg Example

```
int x = 0;
if (y > 0)
  x = 1;
return x;
```

```
l1: %p = alloca i32
    store 0, %p
    %b = %y > 0
    br %b, %l2, %l3
```

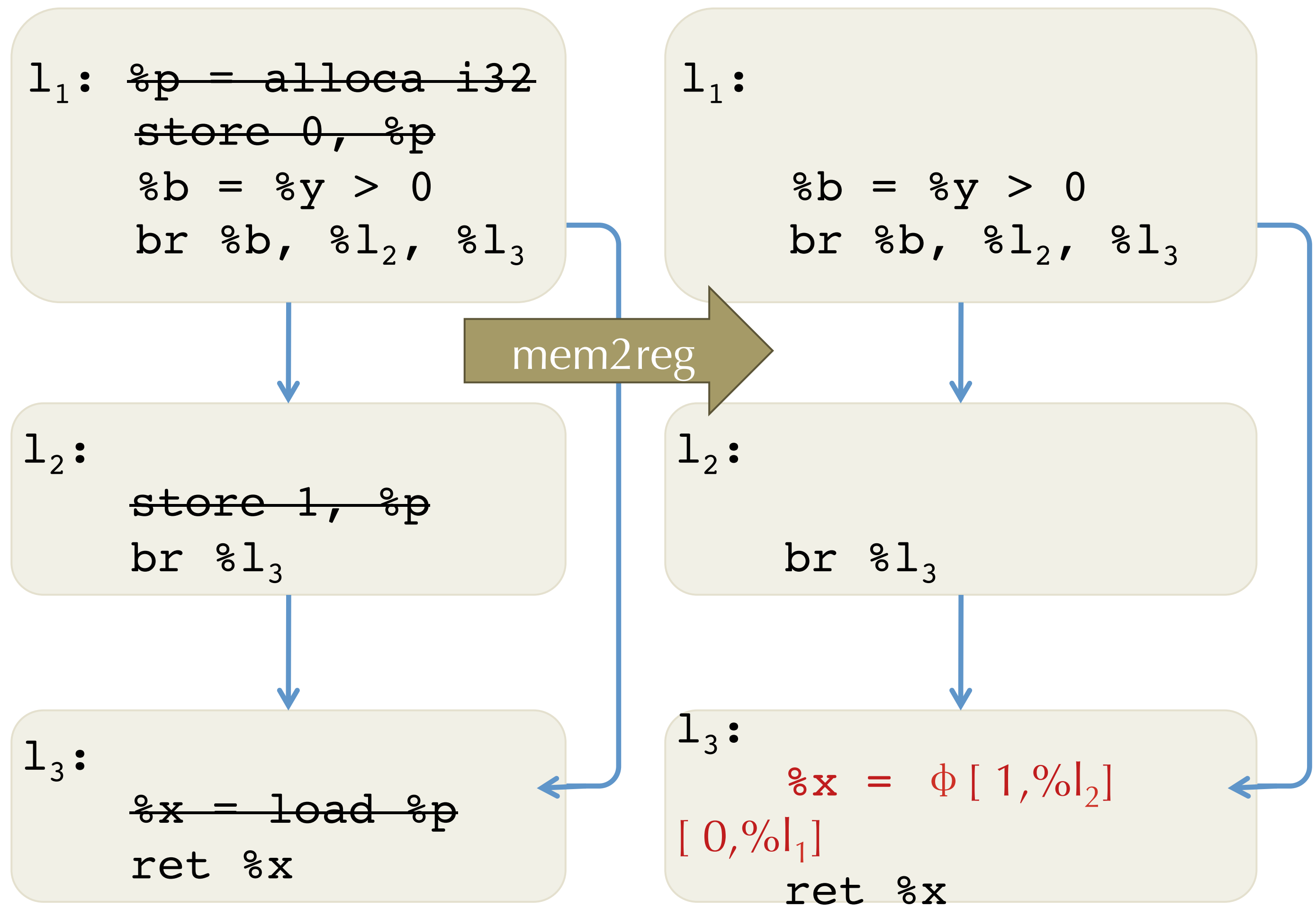
```
l2:
    store 1, %p
    br %l3
```

```
l3:
    %x = load %p
    ret %x
```

The LLVM IR in the trivial SSA form

mem2reg Example

```
int x = 0;
if (y > 0)
  x = 1;
return x;
```



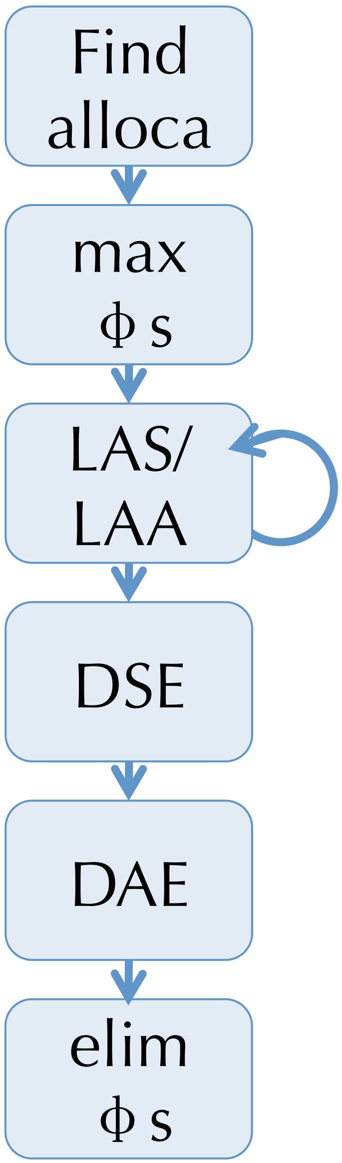
The LLVM IR in the trivial SSA form   Minimal SSA after mem2reg

mem2reg Algorithm

- Two main operations
  - Phi placement (Lengauer-Tarjan algorithm)
  - Renaming of the variables

- Intermediate stage breaks SSA invariant
  - Defining semantics & well formedness non-trivial

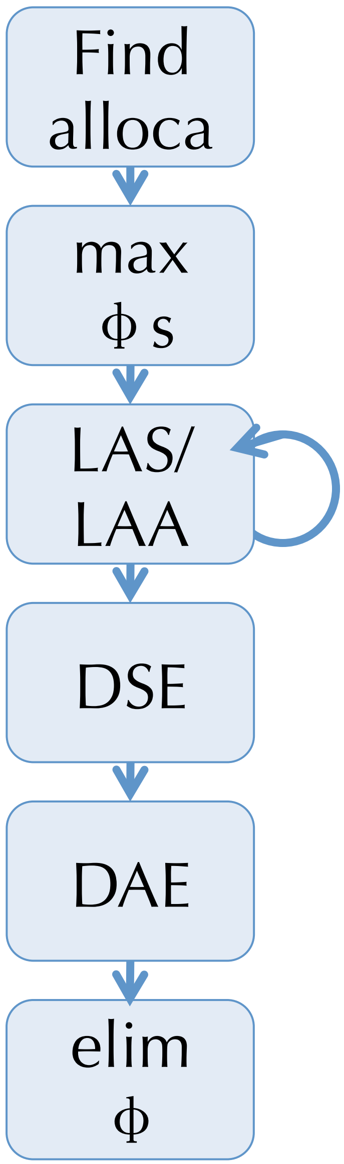
vmem2reg Algorithm



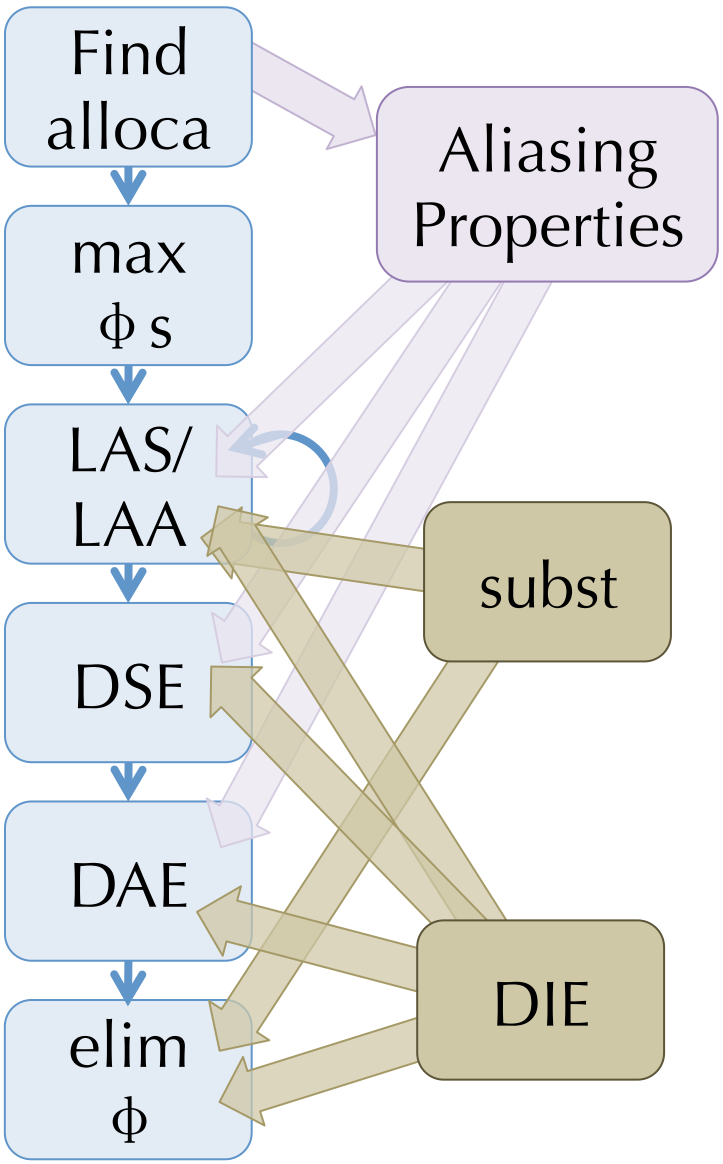
- Incremental algorithm
- Pipeline of micro-transformations
  - Preserves SSA semantics
  - Preserves well-formedness

- Inspired by Aycock & Horspool 2002.

How to Establish Correctness?



How to Establish Correctness?



1. Simple aliasing properties
   (e.g. to determine promotability)
2. Instantiate proof technique for
   – Substitution
   – Dead Instruction Elimination

   P_{DIE} = …
   Initialize(P_{DIE})
   Preservation(P_{DIE})

   Progress(P_{DIE})
4. Put it all together to prove composition of "pipeline" correct.

vmem2reg is Correct

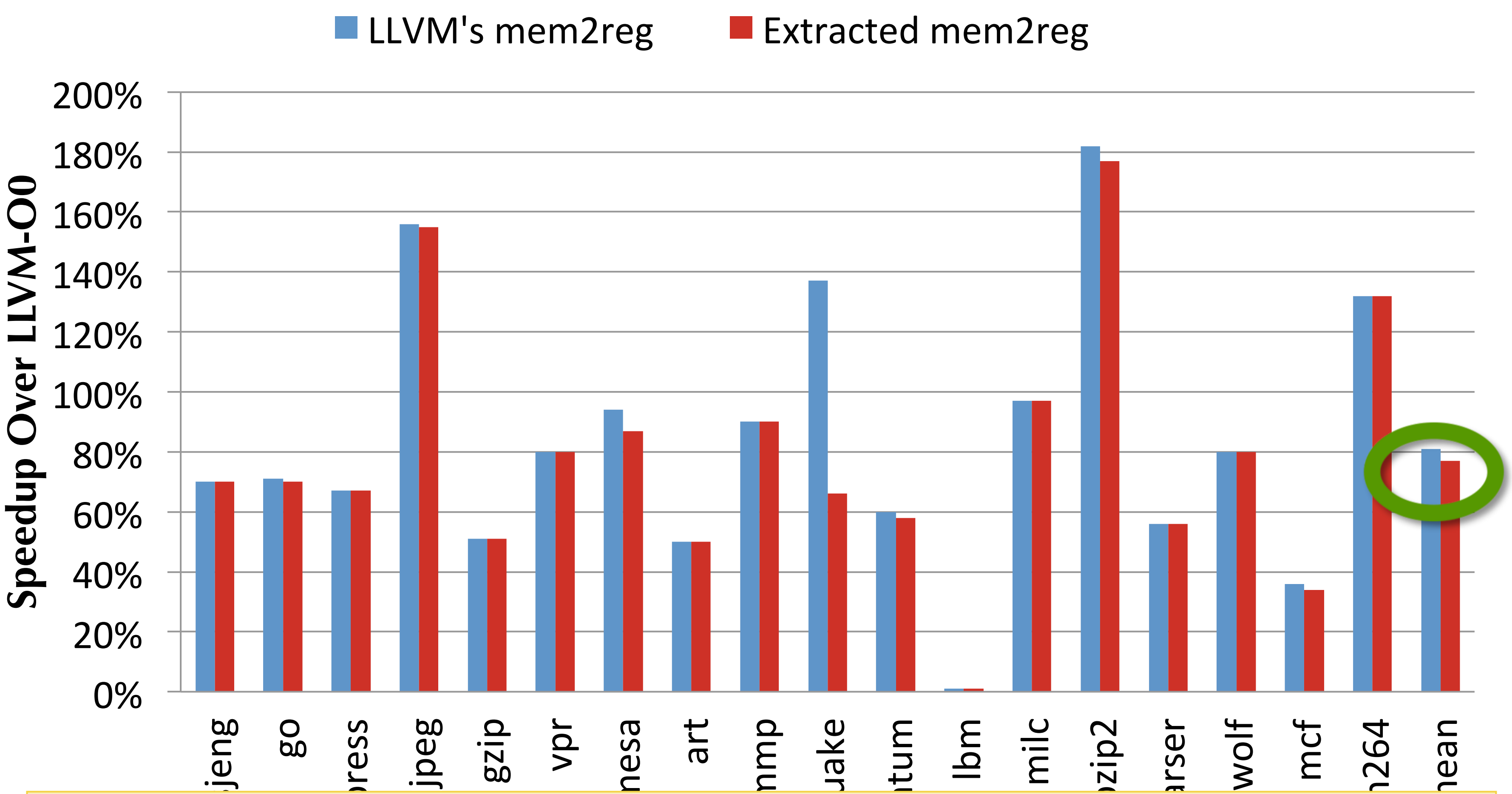
Theorem: The vmem2reg algorithm preserves the semantics of the source program.

Proof:

Composition of simulation relations from the "mini" transformations, each built using instances of the sdom proof technique.

(See Coq Vellvm development.) □

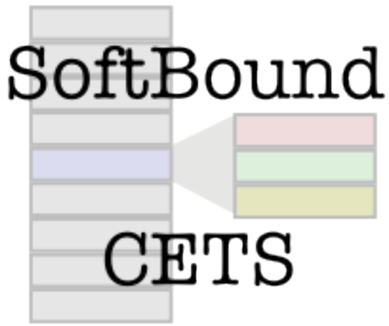
Runtime overhead of verified mem2reg



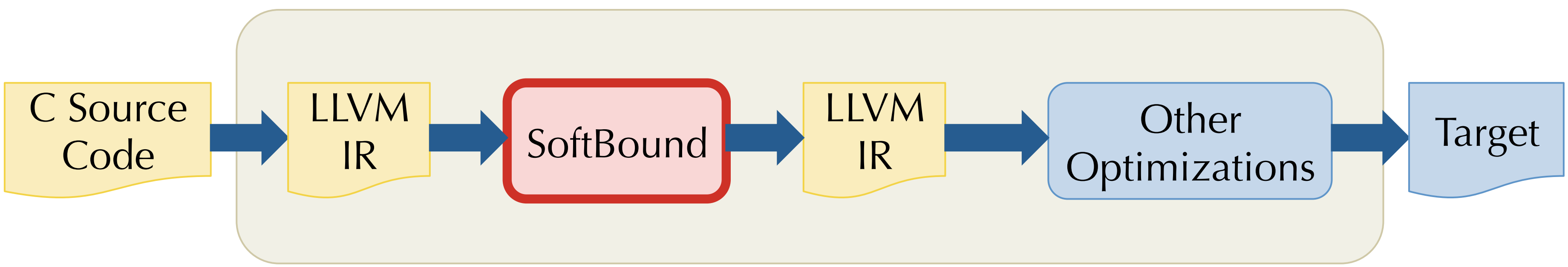
Vmem2reg: 77% LLVM's mem2reg: 81%

(LLVM's mem2reg promotes allocas used by intrinsics)

SoftBound



- Implemented as an LLVM pass.
- Detect spatial/temporal memory safety violations in legacy C code.
- Good test case:
  - Safety Critical ⇒ Proof cost warranted
  - Non-trivial Memory transformation



SoftBound

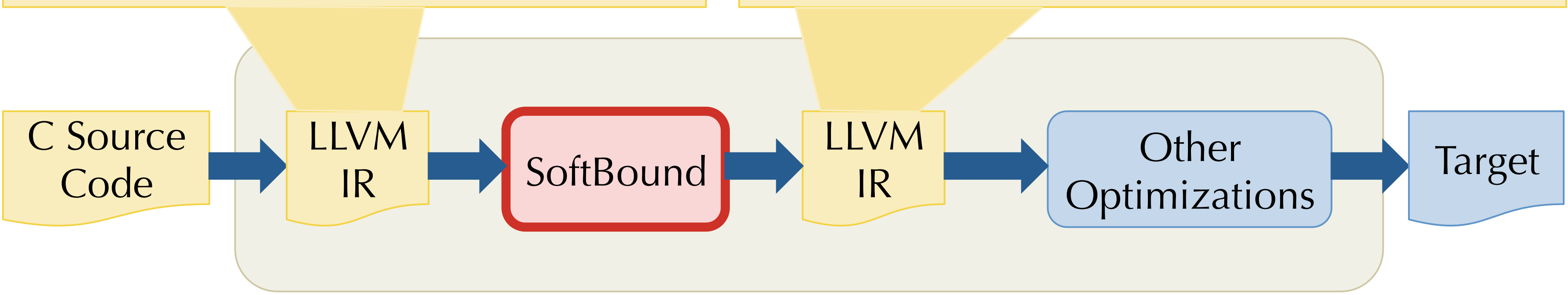
```
%p = call malloc [10 x i8]

%q = gep %p, i32 0, i32 255

store i8 0, %q
```

Maintain base and bound for all pointers

Propagate metadata on assignment

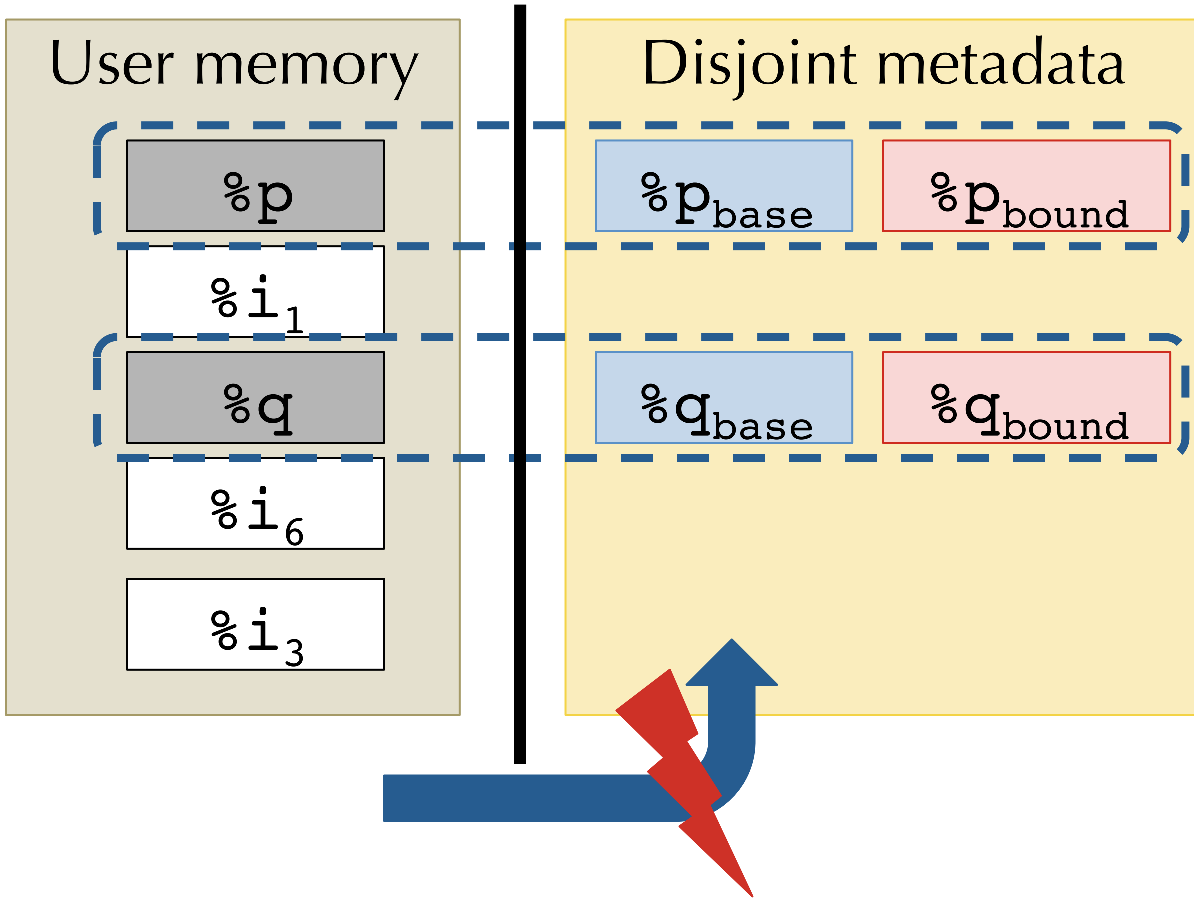
Check that a pointer is within its bounds when being accessed

```
%p = call malloc [10 x i8]
%p_base  = gep %p, i32 0
%p_bound = gep %p, i32 0, i32 10

%q = gep %p, i32 0, i32 255
%q_base  = %p_base
%q_bound = %p_bound

assert %q_base <= %q
    /\ %q+1 < %q_bound
store i8 0, %q
```



Disjoint Metadata

- Maintain pointer bounds in a separate memory space.
- Key Invariant: Metadata cannot be corrupted by bounds violation.



Proving SoftBound Correct

1. Define SoftBound(f, σ) = (f_s, σ_s)
   - Transformation pass implemented in Coq.
2. Define predicate: MemoryViolation(f, σ)
3. Construct a *non-standard* operational semantics:

   $$f \vdash \sigma \overset{SB}{\longmapsto} \sigma'$$

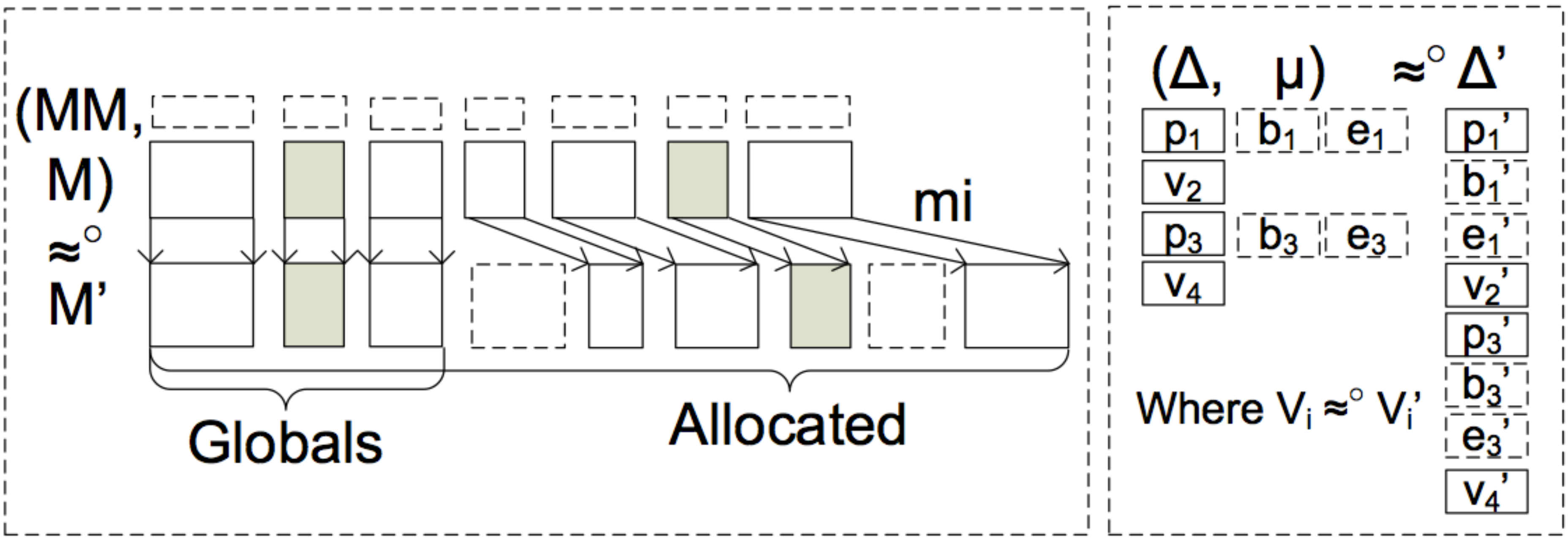
   - Builds in safety invariants "by construction"

   $$f \vdash \sigma \overset{SB}{\longmapsto}{}^{*} \sigma' \Rightarrow \neg\text{MemoryViolation}(f, \sigma')$$

4. Show that the instrumented code simulates the "correct" code:

$$\text{SoftBound}(f, \sigma) = (f_s, \sigma_s) \Rightarrow [f \vdash^{S}_{B} \sigma \longmapsto^{*} \sigma'] \gtrsim [f_s \vdash \sigma_s \longmapsto^{*} \sigma'_s]$$

Memory Simulation Relation



Memory simulation

Frame simulation

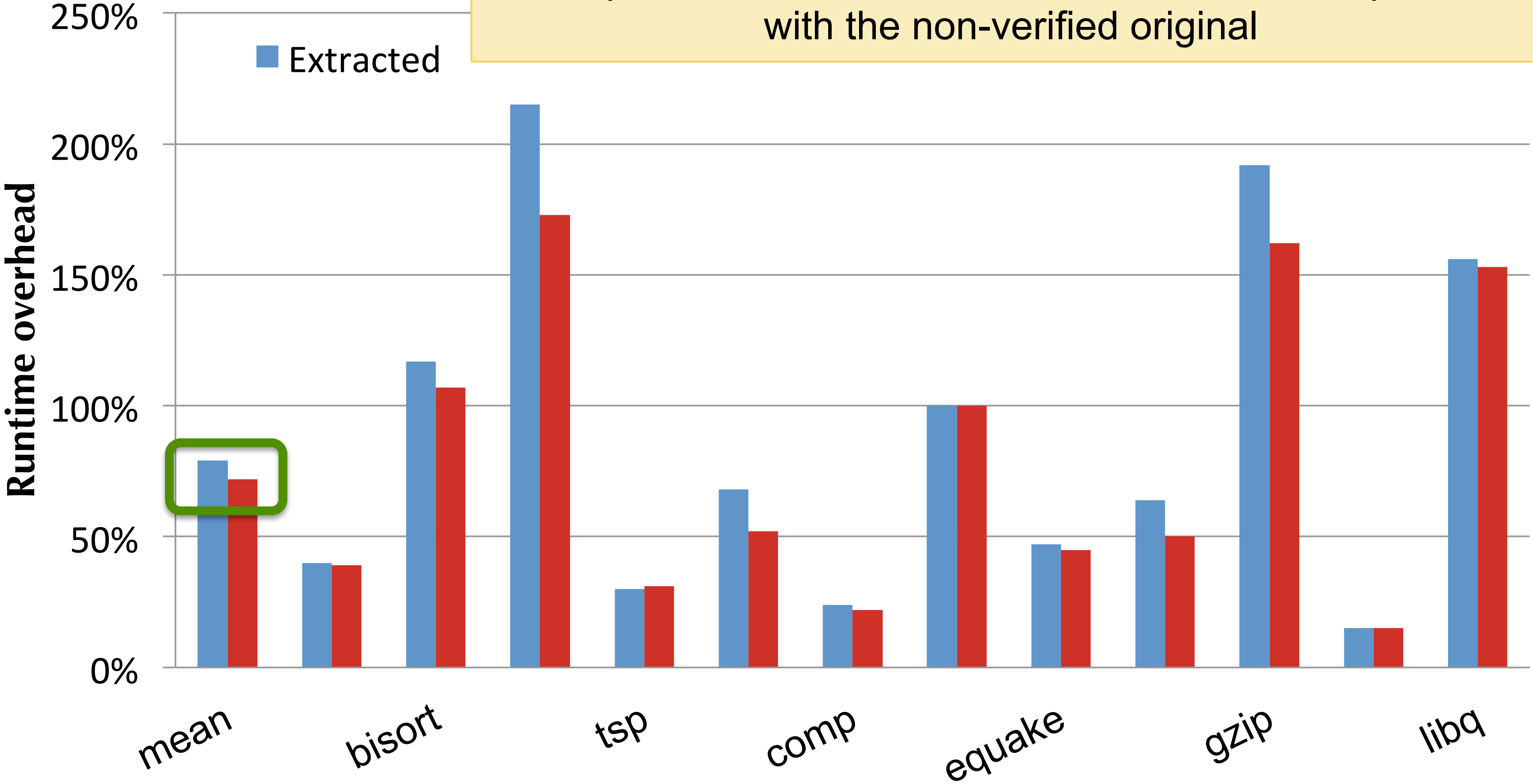
Lessons About SoftBound

- Found several bugs in our C++ implementation
  - Interaction of undef, 'null', and metadata initialization.

- Simulation proofs suggested a redesign of SoftBound's handling of stack pointers.
  - Use a "shadow stack"
  - Simplify the design/implementation
  - Significantly more robust (e.g. varargs)

Competitive Runtime Overhead

The performance of extracted SoftBound is competitive with the non-verified original



FINAL EXAM

Final Exam

- Will cover material since the midterm almost exclusively
  - Starting from Lecture 14
  - Objects, inheritance, types, implementation of dynamic dispatch
  - Basic optimizations
  - Dataflow analysis (forward vs. backward, fixpoint computations, etc.)
    - Liveness
  - Control flow analysis
    - Loops, dominator trees
  - SSA
  - Graph-coloring Register Allocation

- Will focus more on the theory side of things
- Format will be similar to the midterm
  - Simple answer, computation, multiple choice, etc.
  - Sample exam from last time is on the web

What have we learned?

Where else is it applicable?

What next?

COURSE WRAP-UP

Why CIS 341?

- You will learn:
  - Practical applications of theory
  - Parsing
  - How high-level languages are implemented in machine language
  - (A subset of) Intel x86 architecture
  - A deeper understanding of code
  - A little about programming language semantics
  - Functional programming in OCaml
  - How to manipulate complex data structures
  - How to be a better programmer
- Did we meet these goals?

Stuff we didn't Cover

- We skipped stuff at every level…
- Concrete syntax/parsing:
  - Much more to the theory of parsing…
  - Good syntax is art not science!
- Source language features:
  - Exceptions, recursive data types (easy!), advanced type systems, type inference, concurrency
- Intermediate languages:
  - Intermediate language design, bytecode, bytecode interpreters, just-in-time compilation (JIT)
- Compilation:
  - Continuation-passing transformation, efficient representations, scalability
- Optimization:
  - Scientific computing, cache optimization, instruction selection/ optimization

Course Work

- 72% Projects: *The Quaker OAT Compiler*

- 12% Midterm
- 16% Final exam

- Expect this to be a challenging, implementation-oriented course.

I think we met this goal…

Related Courses: Fall 2013

- CIS 500: Software Foundations
  - Dr. Pierce
  - Theoretical course about functional programming, proving program properties, type systems, lambda calculus. Uses the theorem prover Coq.
- CIS 501: Computer Architecture
  - Dr. Devietti
  - 371++: pipelining, caches, VM, superscalar, multicore,…
- CIS 552: Advanced Programming
  - Dr. Weirich
  - Advanced functional programming in Haskell, including generic programming, metaprogramming, embedded languages, cool tricks with fancy type systems
- CIS 670: Special topics in programming languages
  - TBA

Where to go from here?

- Conferences (proceedings available on the web):
  - Programming Language Design and Implementation (PLDI)
  - Principles of Programming Langugaes (POPL)
  - Object Oriented Programming Systems, Languages & Applications (OOPSLA)
  - International Conference on Functional Programming (ICFP)
  - European Symposium on Programming (ESOP)
  - …

- Technologies / Open Source Projects
  - Yacc, lex, bison, flex, …
  - LLVM – low level virtual machine
  - Java virtual machine (JVM), Microsoft's Common Language Runtime (CLR)
  - Languages: OCaml, F#, Haskell, Scala, Go, Rust, …?

Where else is this stuff applicable?

- General programming
  - In C/C++, better understanding of how the compiler works can help you generate better code.
  - Ability to read assembly output from compiler
  - Experience with functional programming can give you different ways to think about how to solve a problem
- Writing domain specific languages
  - lex/yacc very useful for little utilities
  - understanding abstract syntax and interpretation
- Understanding hardware/software interface
  - Different devices have different instruction sets, programming models

Thanks!

- To the TAs: Dmitri, Rohan, and Mitchell
  - for doing an amazing job putting together the projects for the course.

- To *you* for taking the class!

- How can I improve the course?
  - Feedback survey posted to Piazza