Program Equivalence

(slides due to Rahul Sharma)

Motivation

- Verification is specification-limited
  - We need specifications to verification
  - And specifications are hard to come by
- Much research focuses on "well-known" specs
  - Buffer overruns, null dereference, string termination, integer overflows …

Program Equivalence

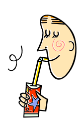
- Are two programs equal?
- Arbitrarily hard verification problem
  - Instances vary from easy to impossible
- And a very common "specification"
  - Specifically, for program optimization
  - Two equal programs, one is faster than the other

Performance

- Many systems where code performance matters
  - Compute-bound
  - Repeatedly executed
- Scientific computing
- Graphics
- Low-latency server code
- Encryption/decryption
- …

More Pain, More Gain

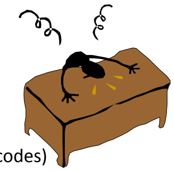
- Programmer Bob requires the best possible code
- Use "best available compiler –O best"
  - E.g., `gcc -O3, cl /Ox, icc -O3`
- Provide unchecked annotations to the compilers
  - E.g, `restrict, __builtin`
- Optimize by hand



Goal

- Minimum pain, maximum gain
- Machine code in 64 bit x86
  - The universal ISA for desktops/servers/laptops
- Automatically generate best possible x86 code

X86 Optimizations are Hard



- x86 is a CISC instruction set
- Many complicated instructions
  - E.g., dpp, popcnt, crc, …
  - ~2000 instruction variants (~300 opcodes)
  - 3439 page manual
- x86 optimizations requires expert knowledge
  - Experts take hours, days, or even weeks
  - For each new processor

Superoptimization

- Massalin [ASPLOS 87], Bansal and Aiken [ASPLOS 06]
  - Enumerate all possible straight line programs
- STOKE, Schkufza, Sharma, Aiken [ASPLOS 13]
  - Random enumeration instead of exhaustive
- Correctness of straight line code is relatively easy
- But we want to prove correctness of loops …

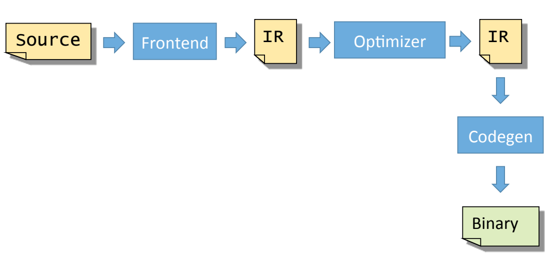
Outline

- Equivalence checker for x86
  - Prove correctness of aggressive optimizations
- From checker to optimizer
  - Different application of verification technology
- Use information from context: even better code
- Conclusion

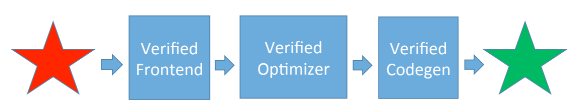
Outline

- Equivalence checker for x86
- From checker to optimizer
- Use information from context: even better code
- Conclusion

Compiler structure

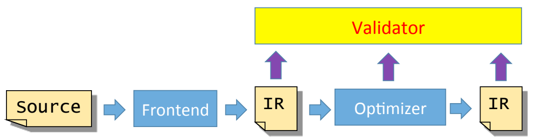


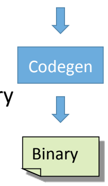
Certified Compilation



- Implementation requires a mammoth effort
  - CompCert does not do any loop optimizations
- Maintaining a certified compiler is difficult

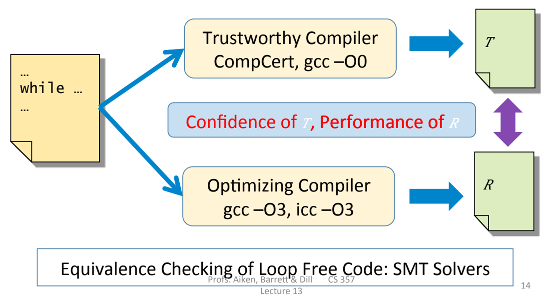
Translation Validation





- Instrument the compiler (gcc/llvm)
- Proves correctness of an IR and not the binary
- Validating binaries is harder
  - Lack of structure, static analysis is difficult

Equivalence Checking

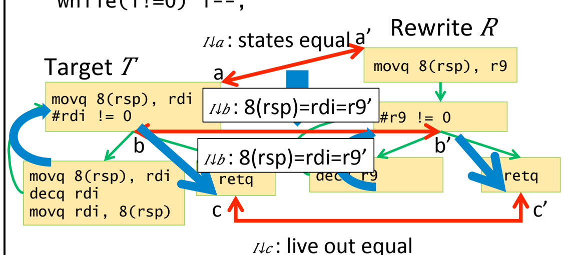


Proof Decomposition

- SMT solvers can prove properties of loop-free code
- We want to prove equivalence of loops
- Decompose proof for loops into subproofs
  - Subproofs about loop free fragment, query SMT solvers
  - *Cutpoints*: break the loops
  - *Invariants:* relationships between $\mathcal{T}$ and $\mathcal{R}$ at cutpoints

Proof Decomposition Example

```
while(i!=0) i--;
```



Simulation Relation

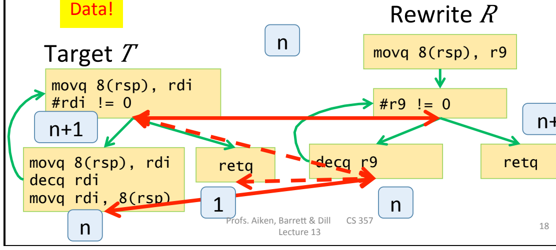
Inference

- Given a simulation relation, proofs for loops reduce to subproofs for loop free fragments
  - Query SMT solvers
- Main challenge: infer a simulation relation
  - Infer cutpoints
  - Infer invariants
- We use compilers as black boxes
- Mine relations from data (program executions)

Cutpoints From Data

- Attempt to detect cutpoints
  - Number of times program points are executed

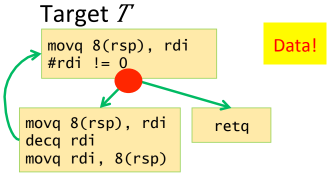
Data!



Invariants

- Invariants are restricted to equalities
  - Infer invariants from observed data values

Target T

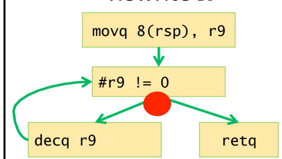


8(rsp)	rdi
2	2
1	1
0	0

Invariants

- Invariants are restricted to equalities
  - Infer invariants from observed data values

Rewrite R



8(rsp)	rdi	r9'
2	2	2
1	1	1
0	0	0

Linear algebra

$A\equiv$

8(rsp)	rdi	r9'
2	2	2
1	1	1
0	0	0

- Mine all equalities
- Find all w s.t. $Aw=0$
- Nullspace

$I\equiv(8(rsp)=rdi)\wedge(rdi=r9')$

- $w_1 = [-1,1,0]$
- $w_2 = [0,1,-1]$

$4eax=edx' + 3$
$10eax+edx=ecx'$

Check Simulation Relation

- Query SMT solvers
  - Incorporate counter-examples in relations
- Sound but not complete
  - If checking succeeds then equivalent
  - Can fail to infer a correct simulation relation
  - Learn from counter-examples

Results [OOPSLA'13]

Program	Run-time (s)
lerner1a	12.94
lerner3b	53.72
bansal	9.89
chomp	11.00
fannkuch	17.03
knucleotide	6.56
lists	1.40
nsievebits*	36.44
nsieve*	166.31
qsort*	140.87
sha1*	12888.87

- T: from CompCert
- R: from GCC
- Prove $T\equiv R$

Outline

- Equivalence checker for x86
- From checker to optimizer
- Use information from context: even better code
- Conclusion

From Checkers to Optimizers

- We have a checker to prove correctness
  - We can be adventurous with optimizations!
- Bombard the checker with random programs
  - Take the fastest correct program
- STOKE (Schkufza, Sharma, Aiken ASPLOS'13)
  - Start from an initial program (unoptimized, empty, …)
  - Generate new programs by repeated random changes
  - Output the fastest **correct** program found in time limit

Random Transformations

original

```
...
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
movq rcx, rax
movl edx, edx
imulq r9, rax
...
```


Random Transformations

original

```
...
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
movq rcx, rax
movl edx, edx
imulq r9, rax
...
```

insert

```
...
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
movq rcx, rax
movl edx, edx
imulq r9, rax
imulq rsi, rdx
...
```


Random Transformations

original

```
...
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
movq rcx, rax
movl edx, edx
imulq r9, rax
...
```

insert

```
...
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
movq rcx, rax
movl edx, edx
imulq r9, rax
imulq rsi, rdx
...
```

delete

...
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
~~movq rcx, rax~~
movl edx, edx
imulq r9, rax
...

Random Transformations

original

```
...
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
movq rcx, rax
movl edx, edx
imulq r9, rax
...
```

insert

```
...
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
movq rcx, rax
movl edx, edx
imulq r9, rax
imulq rsi, rdx
...
```

delete

...
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
~~movq rcx, rax~~
movl edx, edx
imulq r9, rax
...

instruction

```
...
movl ecx, ecx
shrq 32, rsi
salq 16, rcx
movq rcx, rax
movl edx, edx
imulq r9, rax
...
```


Random Transformations

original

```
...
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
movq rcx, rax
movl edx, edx
imulq r9, rax
...
```

insert

```
...
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
movq rcx, rax
movl edx, edx
imulq r9, rax
imulq rsi, rdx
...
```

delete

...
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
~~movq rcx, rax~~
movl edx, edx
imulq r9, rax
...

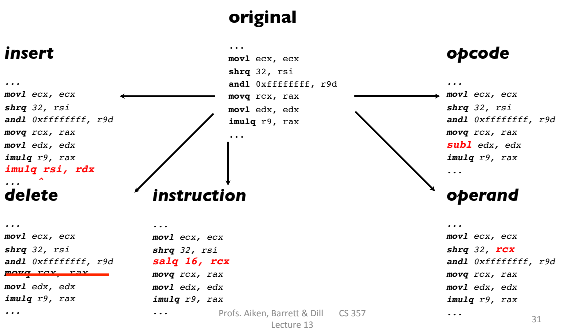
instruction

```
...
movl ecx, ecx
shrq 32, rsi
salq 16, rcx
movq rcx, rax
movl edx, edx
imulq r9, rax
...
```

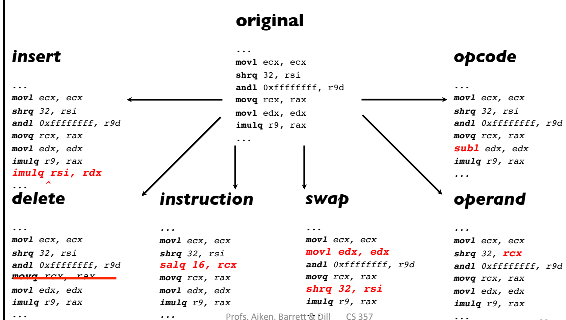
opcode

```
...
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
movq rcx, rax
subl edx, edx
imulq r9, rax
...
```

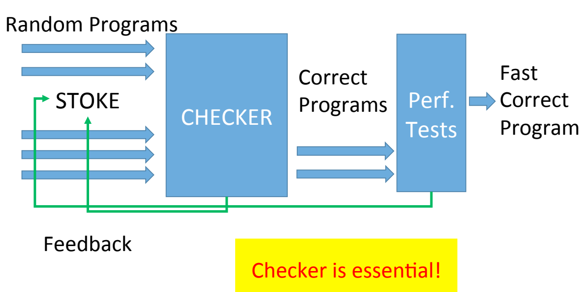
Random Transformations



Random Transformations



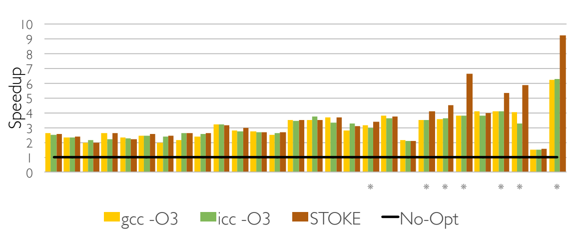
Optimizer



Benchmarks

- **Synthesis Kernels:** 25 loop-free kernels taken from A Hacker's Delight (Gulwani el al. '11)
- **SSL:** Montgomery multiplication kernel
- **Heap Modifying:** Linked list traversal
- **Linear Algebra:** SAXPY from BLAS

Results (ASPLOS' 13, OOPSLA' 13)



- STOKE is comparable or outperforms gcc and icc while maintaining correctness w.r.t. unoptimized code

Observation

- Speedups are good: 70% over production compiler
- However, x86 experts produce much better code
- Why can't we get 2X, 3X speedups over gcc –O3?
- Checker rejects many "good" programs
  - Work perfectly with the application
  - Fail on some weird corner case that cannot arise

Outline

- Equivalence checker for x86
- From checker to optimizer
- Use information from context: even better code
- Conclusion

Example

- Target

```
foo(int32_t* a,
    int32_t* b)
*a++=*b++;
*a++=*b++;
*a++=*b++;
*a++=*b++;
return;
```

- Expert Rewrite

```
MOVAPD xmm0, [b]
MOVAPD [a], xmm0
RET
```

- Incorrect, if a and b overlap
- Segfault if a and b are not 16 byte aligned

Conditions

- Cannot use the fast rewrite in an arbitrary context
- Developers know conditions on the context
  - Used in adding unchecked annotations, writing assembly
- Unconditionally correct optimizations are effective
  - But fall short of the fastest code desired
- For performance, we need *conditional equivalence*

Conditional Correctness

- Target

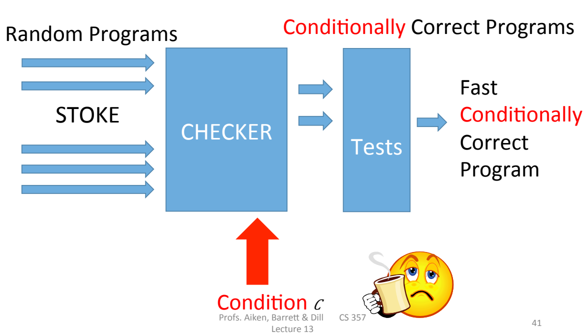
```
foo(int32_t* a,
    int32_t* b)
*a++=*b++;
*a++=*b++;
*a++=*b++;
*a++=*b++;
return;
```

- Rewrite

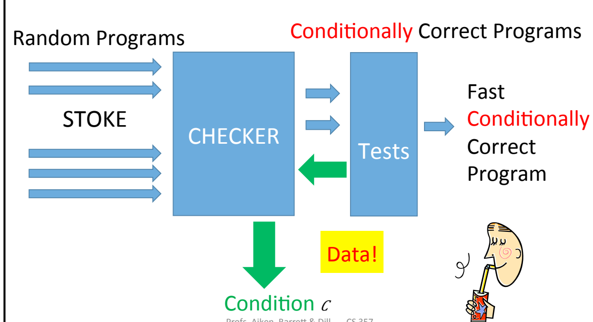
```
MOVAPD xmm0, [b]
MOVAPD [a], xmm0
RET
```

- Conditionally correct
  - restrict(a), restrict(b)
  - a, b are 16 byte aligned

Conditionally correct optimizer



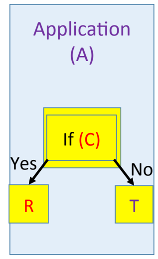
Conditionally correct optimizer



Condition Inference

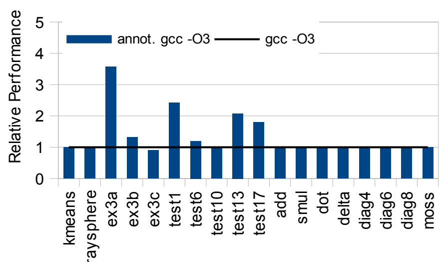
- Tests encode the conditions on contexts implicitly
- Learn facts about tests
  - Keep adding facts until the proof succeeds
- Aliasing: two memory dereferences do not alias
- Alignment: an address is x-byte aligned
- Equalities, inequalities, floating point specific

Conditions



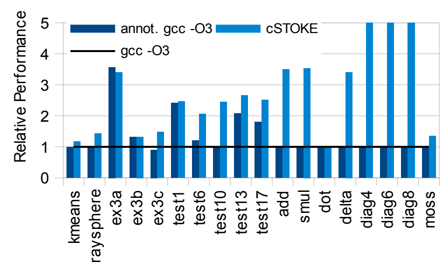
1. Analyze A to verify C
2. Manually verify C
3. Runtime check

Conditional GCC



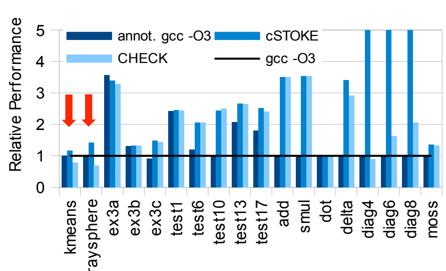
Annotations can help a compiler significantly

Conditional STOKE



STOKE with condition inference is comparable or outperforms gcc with or without annotations

Runtime Checks



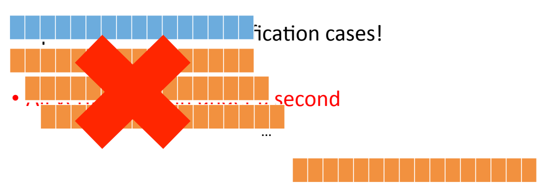
Overhead of condition checking can be bearable

Aliasing

- Restricting aliasing results in simpler queries
  - For equivalence, $p=q \vee p \neq q$
  - Byte addressable memory, vector instructions

fication cases!

second



Conclusion

- Proving the correctness of optimizations is hard
  - Facilitated by data-driven invariant inference
- Better checkers lead to better optimizers
- Context insensitive optimizations fall short
  - Leverage context by inferring conditions from data
- Generate fast and provably correct code