STANDARD

Souper-Charging Peepholes with Target Machine Info

Min-Yih Hsu, University of California Irvine (minyihh@uci.edu)

Stan Kvasov, MediaTek USA (Stan.Kvasov@mediatek.com)

Vince Del Vecchio, MediaTek USA (Vince.DelVecchio@mediatek.com)

Outline

- Introduction to Souper
- Souper with Target Machine Info
- Peepholes with More Context
- Summary

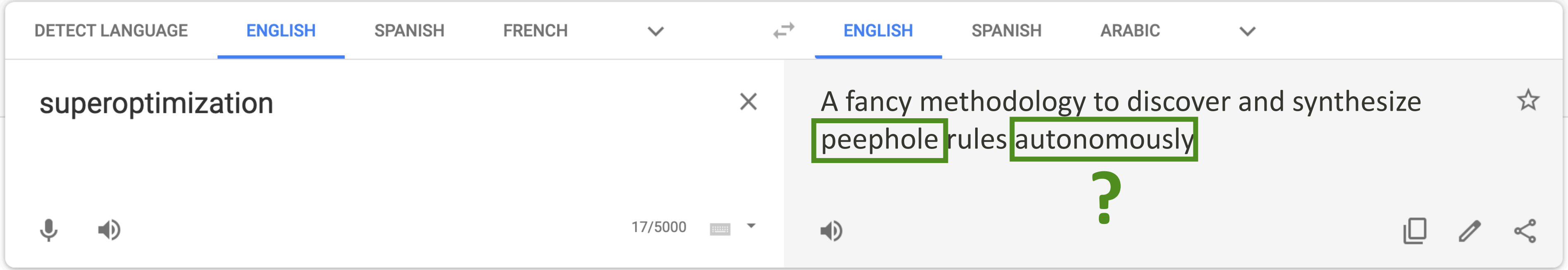
Introduction to Souper

STANDARD

What is Souper?

STANDARD

Souper is a LLVM-based superoptimization framework by R Sasnauskas et al [1].



[1] Sasnauskas, Raimondas, et al. "Souper: A synthesizing superoptimizer." arXiv preprint arXiv:1711.04422 (2017).

What is Superoptimization?

Drawbacks of Hand-Written Peepholes Rules

- Requires **manual work** to discover potential peephole transformations.
- Need extra effort to verify the **correctness** of the rules.
- Hard to **maintain** if there are too many rules.

What is Superoptimization?

Discover Peephole Rules Autonomously

Replace manual effort with automated peephole generation:

1. A **systematic** search for potential peepholes from input code.
2. A peephole rule verifier backed up by **formal proofs**.

STANDARD

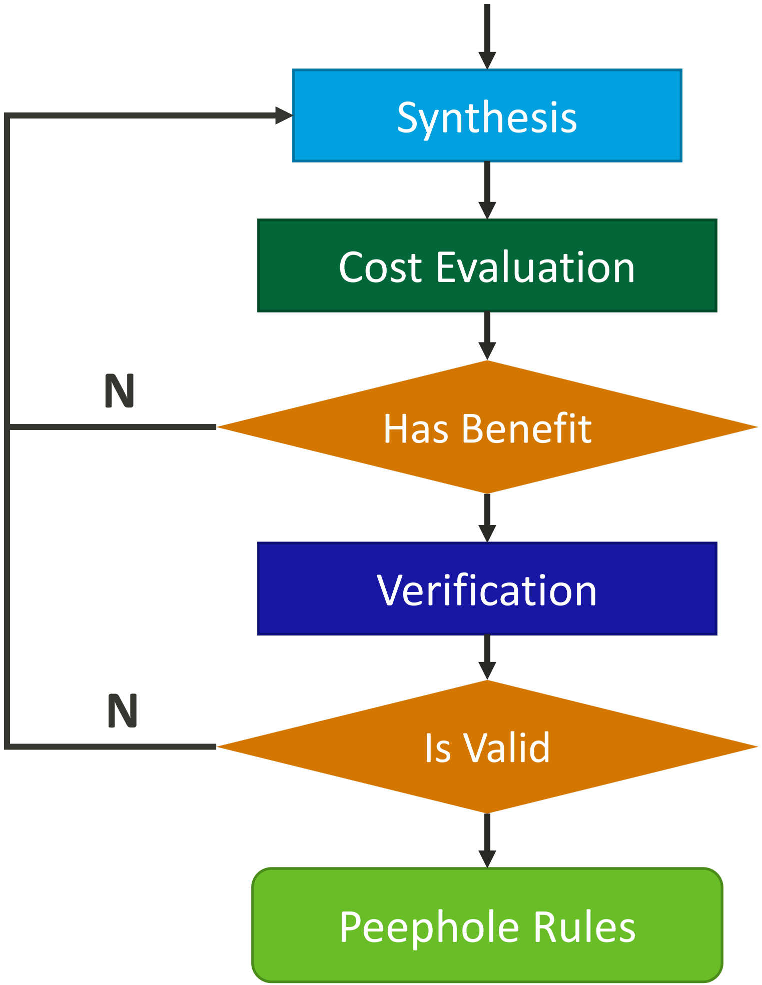
What is Souper?

A LLVM-based Superoptimization Framework

- Performs superoptimization on the input LLVM IR bitcode to generate peephole rules.
    - Certain LLVM instructions (e.g. Load and Store) are not supported
- Supports several kinds of peephole synthesis
    - We’re primarily focusing on **Instruction Synthesis (CEGIS)**.
- Proves correctness of peephole rules using an SMT solver.

Souper

Workflow



STANDARD

STANDARD

Souper

Some Terminologies

LHS

```
%0 = zext i32 %var to i64
%1 = and i64 4294927264, %0
%2 = add i64 18446744073709551584, %1
%3 = lshr i64 %2, 5
%4 = add i64 1, %3
%5 = and i64 1, %4
%result = icmp eq i64 0, %5
```

Inference / Synthesis

RHS

```
%0 = or i32 %var, 32
%result = icmp ne i32 %0, %var
```

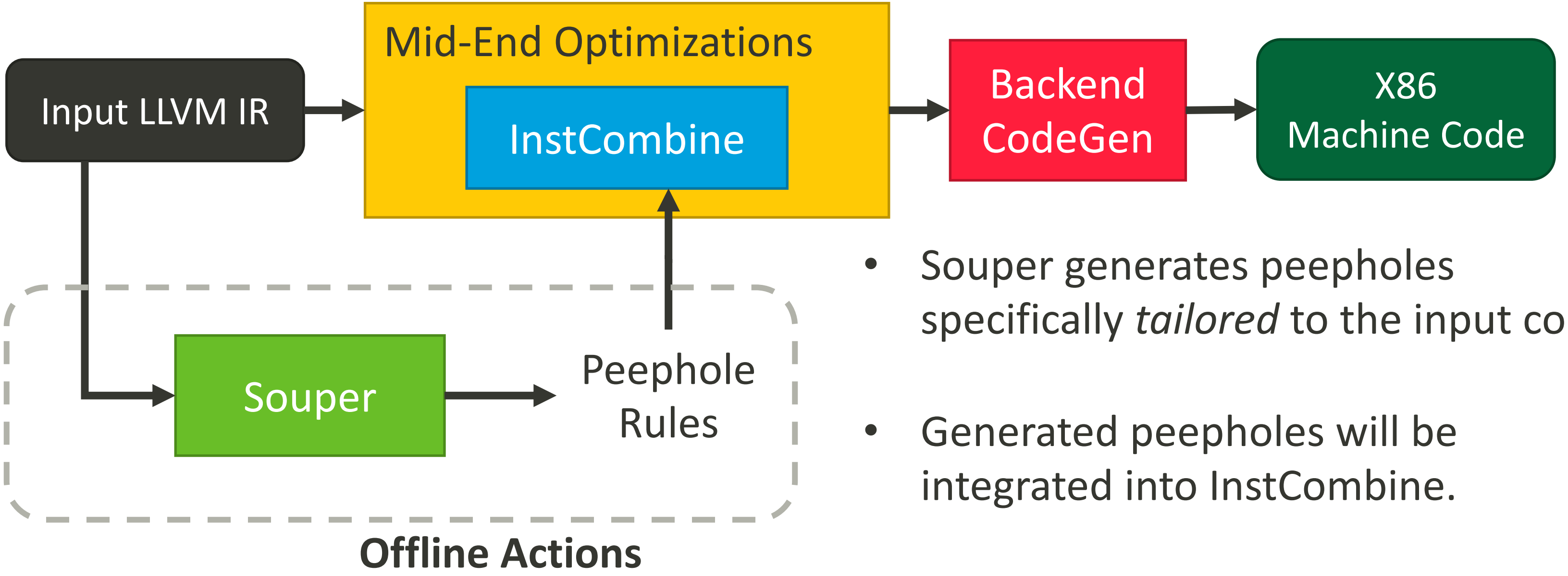
A Peephole is Found

Souper with Target Machine Info

STANDARD

Scenario

Using Souper Offline



- Souper generates peepholes specifically *tailored* to the input code.
- Generated peepholes will be integrated into InstCombine.

Motivating Example

A Peephole in LLVM IR

LHS

```
%0 = fcmp olt double %v0, %v1
%1 = fcmp ogt double %v0, %v1
%2 = zext i1 %1 to i32
%result = select i1 %0, -1, %2
```

RHS

```
%0 = fcmp olt double %v0, %v1
%1 = fcmp ogt double %v0, %v1
%2 = zext i1 %1 to i32
%3 = sext i1 %0 to i32
%result = or i32 %2, %3
```

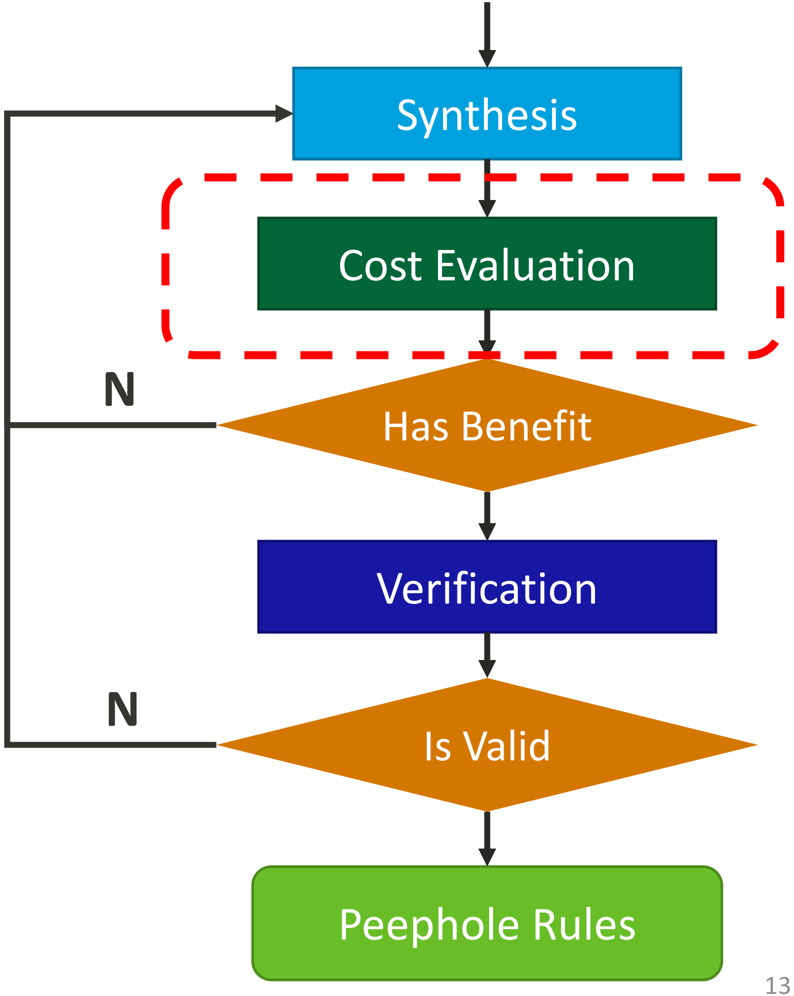
Changed

Unchanged

Cost Functions in Souper


```
int cost(Inst* I) {
  int C = getOpCost(I->Op);
  for(Inst* Op : I->Operands)
    C += cost(Op);
  return C;
}
```

Cost	Op
3	Mul, Select …
5	FP Operations, Div …
1	Other



Motivating Example

A Peephole in LLVM IR

LHS

```
%0 = fcmp olt double %v0, %v1
%1 = fcmp ogt double %v0, %v1
%2 = zext i1 %1 to i32
%out = select i1 %0, i32 -1, i32 %2
```

cost = 3

RHS

```
%0 = fcmp olt double %v0, %v1
%1 = fcmp ogt double %v0, %v1
%2 = zext i1 %1 to i32
%3 = sext i1 %0 to i32
%out = or i32 %2, %3
```

cost = 1 + 1 = 2

Changed

Unchanged

Motivating Example

Comparing Compiled X86-64 Assembly

STANDARD

ucomisd: Compare floating point registers
cmovbel: Move on condition
seta: Set byte on condition

X86-64 Assembly for LHS

```
xorl    %ecx, %ecx
ucomisd %xmm1, %xmm0
seta    %cl
ucomisd %xmm0, %xmm1
movl    -1, %eax
cmovbel %ecx, %eax
```

1~2 cycles

X86-64 Assembly for RHS

```
xorl    %ecx, %ecx
ucomisd %xmm1, %xmm0
seta    %cl
xorl    %eax, %eax
ucomisd %xmm0, %xmm1
seta    %al
negl    %eax
orl     %ecx, %eax
```

1 cycle

Changed
Unchanged

Motivating Example

Comparing Compiled X86-64 Assembly

STANDARD

`ucomisd:`	Compare floating point registers
`cmovbel:`	Move on condition
`seta:`	Set byte on condition

X86-64 Assembly for LHS

```
        xorl    %ecx, %ecx
        ucomisd %xmm1, %xmm0
        seta    %cl
        ucomisd %xmm0, %xmm1
select  movl    -1, %eax
        cmovbel %ecx, %eax
```

X86-64 Assembly for RHS

```
xorl    %ecx, %ecx
ucomisd %xmm1, %xmm0
seta    %cl
xorl    %eax, %eax
ucomisd %xmm0, %xmm1
seta    %al
negl    %eax
orl     %ecx, %eax
```

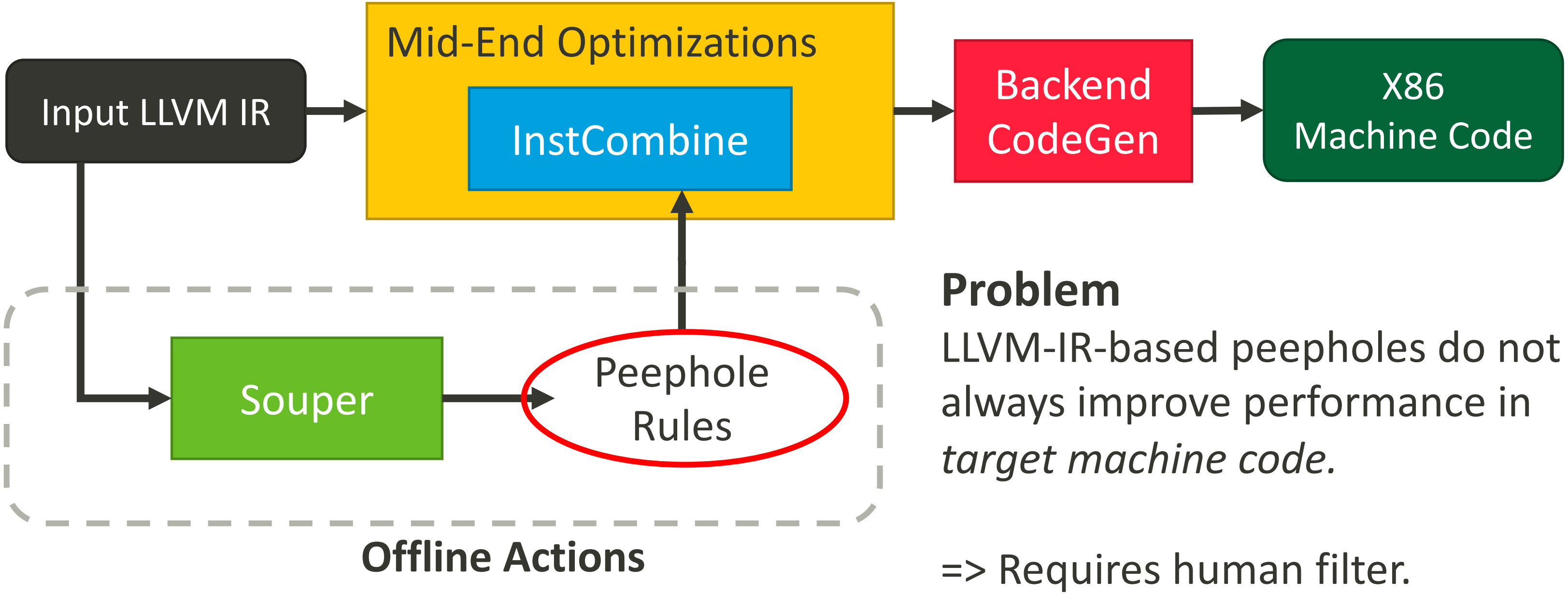
Part of RHS LLVM IR

```
%0 = fcmp olt …
…
%3 = sext i1 %0 …
%out = or …, %3
```

STANDARD

Using Souper Offline

Challenges of Peepholes on Machine Code Performance



Problem

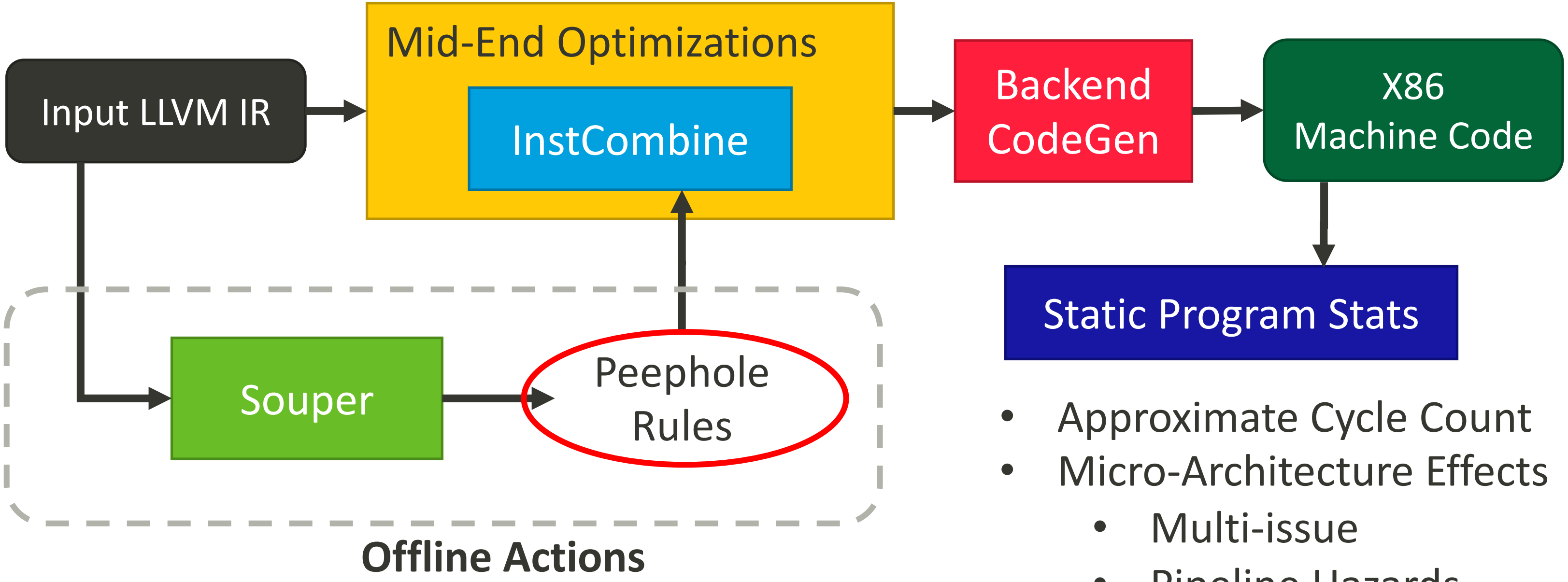
LLVM-IR-based peepholes do not always improve performance in *target machine code.*

=> Requires human filter.

STANDARD

Using Souper Offline

Performance Evaluation on Machine Code



- Approximate Cycle Count
- Micro-Architecture Effects
  - Multi-issue
  - Pipeline Hazards

Performance Evaluation on Machine Code

LLVM MC Analyzer (MCA) in a Nutshell

- Statically measure the performance of **machine code** for a specific CPU
- Measure the approximate **instruction cycles** with the effects of:
    - Hardware resource pressure
    - Dynamic instruction scheduling stats

$$\textbf{Benefit} = \textbf{Cycle}_{\textbf{orig}} - \textbf{Cycle}_{\textbf{after applying a peephole}}$$

Using Souper Offline

Baseline Experiment - Setup

- **Benchmark:** the *SingleSource* benchmarks in llvm-test-suite.
    - Split one function per file. Total of 1788 functions.
- **Target Architecture:** x86_64
- **Synthesis Mode**: Instruction synthesis using CEGIS.
- **Timeout**: 3600 seconds (1 hour) per file.
- **Components Used for Synthesis**: const, and, or, shl, lshr, eq, ne
- **Running Environment:** 20 nodes w/ 20 cores and 32 GB RAM per node.

STANDARD

STANDARD

Using Souper Offline

Baseline Experiment - Results

# of files	# of peepholes found	# of peepholes that bring *positive* benefit	# of peepholes that bring *negative* benefit	# of peepholes that bring *no* benefit
1788	257	145	**63 (~25%)**	**49 (~18%)**

- Among the 257 peepholes we found, about **43%** of them bring no positive benefit.
- About *a quarter* of the peepholes even bring performance *regressions*.

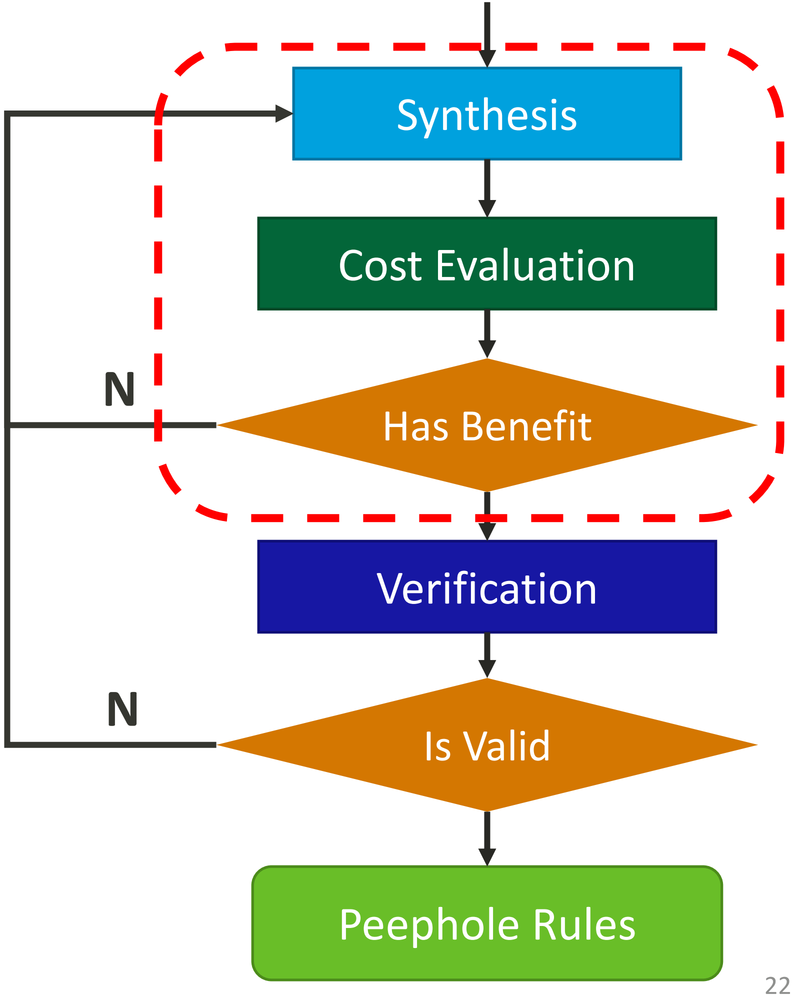
Tuning Cost Functions in Souper

Current Cost Function

```
bool
has_benefit(Inst* LHS, Inst* RHS) {
  return cost(RHS) < cost(LHS);
}

while(…) {
  RHSCandidates
    = Synthesize(LHS, blacklist);

  for(auto* C: RHSCandidates) {
    if(has_benefit(LHS, C))
      return C;
    else
      blacklist.insert(C);
  }
}
```



Tuning Cost Functions in Souper

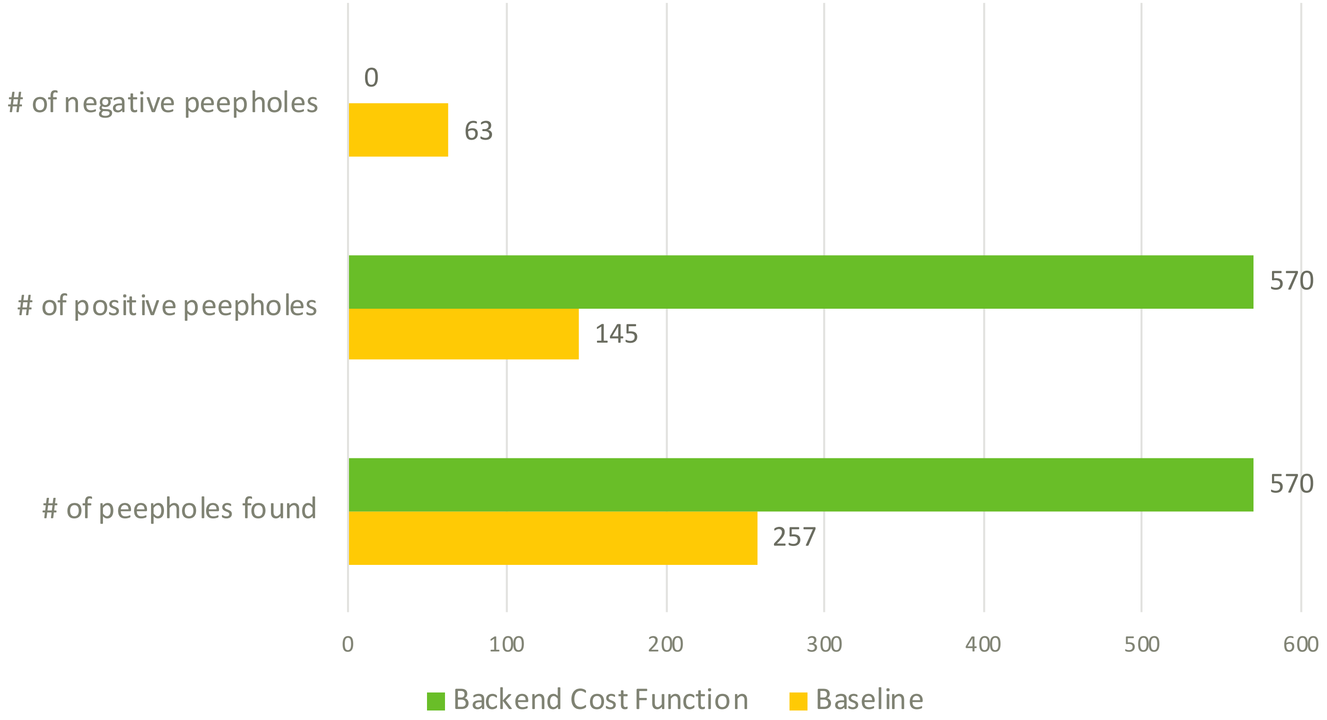
Using Backend in the Cost Function

1. For each RHS candidate (i.e. peephole candidate), clone a llvm::Module for it.
2. Apply the candidate peephole on the cloned Module.
3. Generate machine code for the cloned Module.
4. Using the approximate cycle count measured by LLVM MCA as the **cost** for this RHS candidate.

STANDARD

Tuning Cost Functions in Souper

Backend Cost Function Experiment Results



STANDARD

Tuning Cost Functions in Souper

Backend Cost Function Experiment Results – Running Time

	Total # of functions	# of timed out functions	Total Running Time
Baseline	1788	81	~ 99 minutes
Backend Cost Function		127	~ 98 minutes

Souper with More Context

STANDARD

STANDARD

Revisiting Baseline Regressions

A Peephole in LLVM IR

LHS

```
for (iv = 0; iv < 30; ++iv){
    %0 = sub i32 30, %iv
    %1 = shl i32 1, %0
    %2 = and i32 %1, %var1
    %result = icmp i32 %2, 0
}
```

RHS

```
for (iv = 0; iv < 30; ++iv){
    %0 = lshr i32 (1 << 30), %iv
    %1 = and i32 %0, %var1
    %result = icmp i32 %1, 0
}
```

Changed

Unchanged

Induction Variable

Revisiting Baseline Regressions

Comparing Generated X86-64 Assembly

```
bt r0, r1
```
Set *CF* status flag with the value of *r0*-th bit in *r1*

X86-64 Assembly for LHS

```
movl 30, %ecx
loop:
  btl  %ecx, %eax
  …
  decl %ecx
  …
  jne loop
```

LHS LLVM IR

```
%0 = sub i32 30, %iv
%1 = shl i32 1, %0
%2 = and i32 %1, %var1
%result = icmp i32 %2, 0
```

X86-64 Assembly for RHS

```
movl 0, %ecx
loop:
  movl  (1 << 30), %esi
  shrl  %cl, %esi
  testl %eax, %esi
  …
  incq  %rcx
  …
  jne loop
```

RHS LLVM IR

```
%0 = lshr i32 (1 << 30), %iv
%1 = and i32 %0, %var1
%result = icmp i32 %1, 0
```

Red: Differences caused by ISel

Blue: Differences caused by other optimizations

Revisiting Baseline Regressions

Interactions Between the Peephole and ISel

STANDARD

What Souper Replaced

```
%0 = sub i32 30, %iv
%1 = shl i32 1, %0
%2 = and i32 %1, %var1
%result = icmp i32 %2, 0
```

Reduced to BT after X86 ISel

Finding a Proper Peephole Context

- This example shows we need to add a **predicate** to Souper's peepholes when converting to InstCombine rules.
  - We call this the **context**.
- Previous backend cost function is basically using *whole Module* as the context.
  - It won't work well in *InstCombine*.

How do we find the proper context?

Finding a Proper Peephole Context

Ideas

STANDARD

1. **Reduction**. Given a Module with a peephole, reduce to a *minimum region* as the context that enables the peephole to bring positive benefit.

   OR

2. **Expansion**. Given a peephole LHS, expand to include more context until the peephole has positive benefit.

Finding a Proper Peephole Context

Idea: Context Reduction via LLVM bugpoint

What is bugpoint?

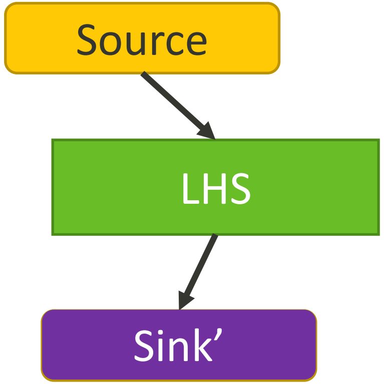
A tool to help developers identify crashes in LLVM Passes by reducing input IR into *minimum region* that recreates the problem.

	What Happened	What bugpoint can find
Normal Usage	“Interesting” when the compiler crashes.	Minimum region that causes the same crash
Use with Souper + Backend Cost Function	“Interesting” when the target peephole is found	Minimum region where the target peephole brings positive benefit

Finding a Proper Peephole Context

Idea: Expanding Context from the Peephole

STANDARD

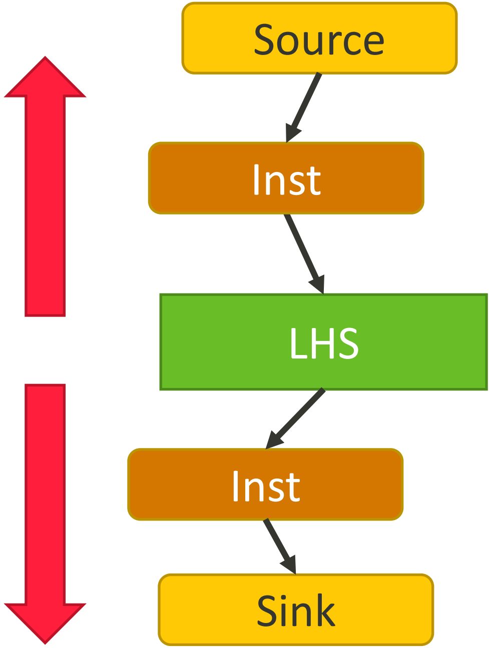


- Starting with the LHS, add source (e.g. parameters) & sink (e.g. return) instructions to get a self-contained fragment.
- We can also try different source / sink instructions (e.g. load / store) to see if that affects the benefit.

STANDARD

Finding a Proper Peephole Context

Idea: Expanding Context from the Peephole



- Starting with the LHS, add source (e.g. parameters) & sink (e.g. return) instructions to get a self-contained fragment.
- We can also try different source / sink instructions (e.g. load / store) to see if that affects the benefit.
- Following the (SSA) Def-Use chain, add more original code until we see the benefit.

Open Questions

- What to do about noise in MCA measurements?
  - E.g. Variance due to RA / spilling
- How often is context an issue?
- What are the runtime benefits?
- How much gain if we target code size instead of performance?
- Souper is very sensitive to RHS component selection. How should we choose?

Input welcome!

STANDARD

Summary

- Souper needs a more *precise profitability test* with regards to performance improvements on different target machines.
    - Whether or not a peephole is beneficial is a function of:
        - Target ISA
        - Peephole Context
        - Optimizations
- Integrating backend into cost functions brings nearly **4x of improvement** on the amount of beneficial peepholes Souper found.
- We presented some ideas for helping Souper find a **proper peephole context** autonomously.

Thank You

Question & Answer

MediaTek is Hiring!!

Full-Time or Interns

Contact: Vince.DelVecchio@mediatek.com