Peek: A Formally Verified Peephole Optimization Framework for x86

Eric Mullen, Zachary Tatlock, Dan Grossman
University of Washington
CompCert
CompCert

World’s $nth$ C compiler
CompCert

World’s \( \textit{nth} \) C compiler

World’s first correct C compiler by machine checked proof
CompCert
Stages of Optimization

- High Level
- Mid Level
- Low Level

Machine Independent

Machine Dependent
Stages of Optimization

- **High Level**
  - e.g. function inlining

- **Mid Level**

- **Low Level**
  - Machine Independent
  - Machine Dependent
Stages of Optimization

- **High Level**
  - e.g. function inlining

- **Mid Level**
  - e.g. CSE

- **Low Level**
  - Machine Independent
  - Machine Dependent
Stages of Optimization

- **High Level**
  - e.g. function inlining

- **Mid Level**
  - e.g. CSE

- **Low Level**
  - Machine Independent
  - Machine Dependent
  - e.g. instruction selection
Stages of Optimization

- High Level
- Mid Level
- Low Level

Machine Independent

Machine Dependent
Stages of Optimization

- High Level
  - Pipelining
    - [Tristan and Leroy, POPL ’10]
  - XCert
    - [Tatlock and Lerner, PLDI ’10]
- Mid Level
  - LCM
    - [Tristan and Leroy, PLDI ’09]
  - Alias Analysis
    - [Robert and Leroy, CPP ’12]
  - Reg Allocation
    - [Rideau and Leroy, CC ’10]
  - Instr Scheduling
    - [Tristan and Leroy, POPL ’08]
- Low Level
  - Machine Independent
  - Machine Dependent
Stages of Optimization

- **High Level**
  - Pipelining
    - Tristan and Leroy, POPL ’10
- **Mid Level**
  - XCert
    - Tatlock and Lerner, PLDI ’10
  - LCM
    - Tristan and Leroy, PLDI ’09
  - Alias Analysis
    - Robert and Leroy, CPP ’12
  - Reg Allocation
    - Rideau and Leroy, CC ’10
- **Low Level**
  - Machine Independent
    - Instr Scheduling
      - Tristan and Leroy, POPL ’08
  - Machine Dependent
    - PEEK
#include <stdlib.h>
#include <stdio.h>

#define NUM_ITER 1000000000

int loop(int a) {
    int res = 37;
    int i = 0;
    for (; i < NUM_ITER; i++) {
        res += (a + i);
    }
    return res;
}

int main() {
    int x = loop(47);
    printf("x: %d\n", x);
    return 0;
}
#include <stdlib.h>
#include <stdio.h>

#define NUM_ITER 1000000000

int loop(int a) {
    int res = 37;
    int i = 0;
    for (; i < NUM_ITER; i++) {
        res += (a + i);
    }

    return res;
}

int main() {
    int x = loop(47);
    printf("x: %d\n", x);

    return 0;
}
```c
#include <stdlib.h>
#include <stdio.h>

#define NUM_ITER 1000000000

int loop(int a) {
    int res = 37;
    int i = 0;
    for (; i < NUM_ITER; i++) {
        res += (a + i);
    }
    return res;
}

int main() {
    int x = loop(47);
    printf("x: %d\n", x);
    return 0;
}
```
Compilation Result

CompCert

.L101:
  leal   0(%ecx,%ebx,1), %edx
  leal   0(%eax,%edx,1), %eax
  leal   1(%ebx), %ebx
  cmpl   $1000000000, %ebx
  jl     .L101
LEA vs. ADD

• Processors are weird
• x86 processors have many adders
• Compilers get to exploit this
Compilation Result

CompCert

.L101:
leal 0(%ecx,%ebx,1), %edx
leal 0(%eax,%edx,1), %eax
leal 1(%ebx), %ebx
cmpl $1000000000, %ebx
jl .L101
Compilation Result

CompCert

Hand Tuned

.L101:
  leal 0(%ecx,%ebx,1), %edx
  leal 0(%eax,%edx,1), %eax
  leal 1(%ebx), %ebx
  cmpl $1000000000, %ebx
  jl .L101
Compilation Result

CompCert

.L101:
  leal 0(%ecx,%ebx,1), %edx
  leal 0(%eax,%edx,1), %eax
  leal 1(%ebx), %ebx
  cmpl $1000000000, %ebx
  jl .L101

Hand Tuned

.L101:
  leal 0(%ecx,%ebx,1), %edx
  addl %edx,%eax
  addl $1,%ebx
  cmpl $1000000000, %ebx
  jl .L101
Compilation Result

CompCert

```
.L101:
    leal    0(%ecx,%ebx,1), %edx
    leal    0(%eax,%edx,1), %eax
    leal    1(%ebx), %ebx
    cmpl    $1000000000, %ebx
    jl      .L101
```

Hand Tuned

```
.L101:
    leal    0(%ecx,%ebx,1), %edx
    addl    %edx,%eax
    addl    $1,%ebx
    cmpl    $1000000000, %ebx
    jl      .L101
```

.466 seconds
Compilation Result

CompCert

.L101:
leal 0(%ecx,%ebx,1), %edx
leal 0(%eax,%edx,1), %eax
leal 1(%ebx), %ebx
cmpl $1000000000, %ebx
jl .L101

Hand Tuned

.L101:
leal 0(%ecx,%ebx,1), %edx
addl %edx,%eax
addl $1,%ebx
cmpl $1000000000, %ebx
jl .L101

.466 seconds

0.250 seconds
Compilation Result

CompCert

```assembly
.L101:
leal 0(%ecx,%ebx,1), %edx
leal 0(%eax,%edx,1), %eax
leal 1(%ebx), %ebx
cmpl $1000000000, %ebx
jl .L101
```

Hand Tuned

```assembly
.L101:
leal 0(%ecx,%ebx,1), %edx
addl %edx,%eax
addl $1,%ebx
cmpl $1000000000, %ebx
jl .L101
```

.466 seconds

1.86x speedup

0.250 seconds
Motivation
Motivation

We want to:
Motivation

We want to:

• Characterize correct assembly transformations

• Verify applying them to CompCert x86 assembly output
Motivation

We want to:  It’s useful for:

• Characterize correct assembly transformations
• Verify applying them to CompCert x86 assembly output
Motivation

We want to:

• Characterize correct assembly transformations
• Verify applying them to CompCert x86 assembly output

It’s useful for:

• Peephole Optimization
• Software Fault Isolation
• Sandboxing
• Profiling and Instrumentation
Outline

• Motivation
• What we built
• Current status
Outline

• Motivation
• What we built
• Current status
CompCert + Peek

CompCert → Peek → x86

Peek: formally verified* peephole optimizer

*Assuming program follows calling convention*
CompCert + Peek

Correct Peephole

CompCert → Peek → x86
Example Peephole

leal 0(%eax,%edx,1), %eax
leal 1(%ebx), %ebx

addl %edx,%eax
addl $1,%ebx
How do we verify the example?

- This is a program transformation
- We prove that the old and the new program do the same thing (bisimulation)
- In CompCert we get bisimulation for free from only forward simulation
Forward Simulation Proof

\[ \sigma \quad \text{match} \quad \delta \]

\[ \sigma' \quad \text{match} \quad \delta' \]
Can we allow this Transformation?

```
leal 0(%eax,%edx,1), %eax
leal 1(%ebx), %ebx
addl %edx,%eax
addl $1,%ebx
```
Can we allow this Transformation?

```
leal 0(%eax,%edx,1), %eax
leal 1(%ebx), %ebx
addl %edx,%eax
addl $1,%ebx
```
Can we allow this Transformation?

```
leal 0(%eax,%edx,1), %eax
leal 1(%ebx), %ebx
addl %edx,%eax
addl $1,%ebx
```
Can we allow this Transformation?

\[
\begin{align*}
\text{prog} & : \\
\text{tprog} & :
\end{align*}
\]

- leal 0(%eax,%edx,1), %eax
- leal 1(%ebx), %ebx
- addl %edx,%eax
- addl $1,%ebx
Can we allow this Transformation?

```
leal 0(%eax,%edx,1), %eax
leal 1(%ebx), %ebx
addl %edx,%eax
addl $1,%ebx
```
Can we allow this Transformation?

```
leal 0(%eax,%edx,1), %eax
leal 1(%ebx), %ebx
addl %edx,%eax
addl $1,%ebx
```
Can we allow this Transformation?

prog  

leal  0(%eax,%edx,1), %eax
leal  1(%ebx), %ebx

addl  %edx,%eax
addl  $1,%ebx

tprog

| eax | ebx | edx |...
|-----|-----|-----|...
| a+d | b+1 | d |...

| eax | ebx | edx |...
|-----|-----|-----|...
| a+d | b+1 | d |...
Can we allow this Transformation?

\[
\text{prog}
\begin{align*}
\text{leal} & \ 0(%eax,%edx,1), \ %eax \\
\text{leal} & \ 1(%ebx), \ %ebx
\end{align*}
\text{tprog}
\begin{align*}
\text{addl} & \ %edx,%eax \\
\text{addl} & \ $1,%ebx
\end{align*}
\]
Can we allow this Transformation?

```
leal  0(%eax,%edx,1), %eax
leal  1(%ebx), %ebx
addl  %edx,%eax
addl  $1,%ebx
```

```
leal  0(%eax,%edx,1), %eax
leal  1(%ebx), %ebx
addl  %edx,%eax
addl  $1,%ebx
```

```plaintext
eax  ebx  edx  ...
a+d  b+1  d  ...
```

```plaintext
eax  ebx  edx  ...
a+d  b+1  d  ...
```
Can we allow this Transformation?

\[
\begin{align*}
\text{prog} & : \\
\text{tprog} & : \\
\text{leal} & 0(%eax,%edx,1), %eax \\
\text{leal} & 1(%ebx), %ebx \\
\text{addl} & %edx,%eax \\
\text{addl} & $1,%ebx
\end{align*}
\]
Can we allow this Transformation?

```assembly
leal  0(%eax,%edx,1), %eax
leal  1(%ebx), %ebx
```

```
addl  %edx,%eax
addl  $1,%ebx
```

What are these Flags?

- Carry Flag
- Parity Flag
- Zero Flag
- Sign Flag
- Overflow Flag
- etc.

- Condition Flags: pieces of state updated by instructions
  - Never mentioned by those instructions
  - Primarily used for conditional branching
We can still allow this Transformation

- If the flags are dead, this peephole is fine
- Dead just means written before read
- Flags are just stored in registers
We can still allow this Transformation

- If the flags are dead, this peephole is fine
- Dead just means written before read
- Flags are just stored in registers
Match States

• Full state equality *too strict*

\[ \sigma_0 \leftrightarrow \text{match} \leftrightarrow \sigma_1 \]

• Same location in the code

• Agree on *live* locations
CompCert + Peek

CompCert → Peek → x86

Liveness Analysis → Peephole Transformation → Rewrites
CompCert + Peek

CompCert → Peek → x86

Proof

Liveness Analysis

Peephole Transformation

Rewrites

Correct

Proof
What is a correct rewrite?
What is a correct rewrite?

- Find pattern

```
leal 0(%eax,%edx,1), %eax
leal 1(%ebx), %ebx
```
What is a correct rewrite?

- Find pattern
  
  \[
  \text{leal } 0(\%eax,\%edx,1), \%eax \\
  \text{leal } 1(\%ebx), \%ebx
  \]

- Replace Pattern
  
  \[
  \text{addl } \%eax,\%edx \\
  \text{addl } \$1,\%ebx
  \]
What is a correct rewrite?

- Find pattern
  \[
  \begin{align*}
  \text{leal} &\ 0(%eax,%edx,1), \ %eax \\
  \text{leal} &\ 1(%ebx), \ %ebx
  \end{align*}
  \]

- Replace Pattern
  \[
  \begin{align*}
  \text{addl} &\ %eax,%edx \\
  \text{addl} &\ \$1,%ebx
  \end{align*}
  \]

- live registers in
  \[
  \{%eax,%edx,%ebx\}\]

-
What is a correct rewrite?

- Find pattern
  
  \[
  \text{leal } 0(%eax,%edx,1), %eax
  \]
  
  \[
  \text{leal } 1(%ebx), %ebx
  \]

- Replace Pattern
  
  \[
  \text{addl } %eax,%edx
  \]
  
  \[
  \text{addl } $1,%ebx
  \]

- Live registers in
  
  \{%eax,%edx,%ebx\}

- Live registers out
  
  \{%eax,%edx,%ebx\}
What is a correct rewrite?

- **Find pattern**
  ```
  leal 0(%eax,%edx,1), %eax
  leal 1(%ebx), %ebx
  ```

- **Replace Pattern**
  ```
  addl %eax,%edx
  addl $1,%ebx
  ```

- **live registers in**
  ```
  {%eax,%edx,%ebx}
  ```

- **live registers out**
  ```
  {%eax,%edx,%ebx}
  ```

- **preserved registers**
  ```
  {%esi,%edi,%esp,%ebp,...}
  ```
Peephole Interface

Proofs:

- Find/Replace Simulation
- Execution preserves values of Preserved registers
- Execution in find pattern doesn’t diverge
- Auxiliary Results (patterns nonempty, patterns same length, no calls in patterns, etc…)
CompCert + Peek

CompCert → Peek → x86

Proof
Liveness Analysis

Correct
Rewrites

Proof
Peephole Transformation
Outline

• Motivation
• What we built
• Current status
Outline

• Motivation
• What we built
• Current status
Assuming the Calling Convention
Assuming the Calling Convention

- We assumed (**did not verify**) that programs we transform follow the C calling convention. We admit facts about:
Assuming the Calling Convention

• We assumed (did not verify) that programs we transform follow the C calling convention. We admit facts about:

• Where call and return instructions send control flow
Assuming the Calling Convention

- We assumed *(did not verify)* that programs we transform follow the C calling convention. We admit facts about:

  - Where call and return instructions send control flow
  - Which registers are live and dead across call and return instructions
Assuming the Calling Convention

- We assumed *(did not verify)* that programs we transform follow the C calling convention. We admit facts about:
  - Where call and return instructions send control flow
  - Which registers are live and dead across call and return instructions

*Every other peephole optimizer makes this same assumption — we would like to verify this, but haven’t yet*
Another Example
Peephole

Bansal and Aiken, ASPLOS ‘06

subl %eax, %ecx
movl %ecx, %eax
decl %eax

notl %eax
addl %ecx, %eax

c - a - 1 = c + ~a
Another Example
Peephole
Bansal and Aiken, ASPLOS '06

\[
\begin{align*}
&\text{subl} \ %\text{eax}, \ %\text{ecx} \\
&\text{movl} \ %\text{ecx}, \ %\text{eax} \\
&\text{decl} \ %\text{eax}
\end{align*}
\]

\[
\begin{align*}
&\text{notl} \ %\text{eax} \\
&\text{addl} \ %\text{ecx}, \ %\text{eax}
\end{align*}
\]

\[c - a - 1 = c + \sim a\]

Unfortunately, not supported by current CompCert model
CompCert Values

Inductive val: Type :=
  Vundef: val
  Vint: int -> val
  Vlong: int64 -> val
  Vfloat: float -> val
  Vptr: block -> int -> val.
CompCert Values

Definition notint (v: val) : val :=
  match v with
  | Vint n => Vint (Int.not n)
  | _ => Vundef
end.
subl %eax, %ecx
movl %ecx, %eax
decl %eax

notl %eax
addl %ecx, %eax
%eax -> Vptr b i,       %ecx -> Vptr b j

subl %eax, %ecx
movl %ecx, %eax
dcl %eax

notl %eax
addl %ecx, %eax
%eax -> Vptr b i,   %ecx -> Vptr b j

subl %eax, %ecx
movl %ecx, %eax
dcl %eax

%ecx -> Vint (j - i)

notl %eax
addl %ecx, %eax
\%eax \rightarrow \text{Vptr } b \ i, \quad \%ecx \rightarrow \text{Vptr } b \ j

\begin{align*}
\text{subl} \ %eax, \ %ecx & \quad \%ecx \rightarrow \text{Vint } (j - i) \\
\text{movl} \ %ecx, \ %eax & \quad \%eax \rightarrow \text{Vint } (j - i) \\
\text{decl} \ %eax & \\
\text{notl} \ %eax & \\
\text{addl} \ %ecx, \ %eax &
\end{align*}
%eax -> Vptr b i,       %ecx -> Vptr b j

subl %eax, %ecx
movl %ecx, %eax
dcl %eax

%ecx -> Vint (j - i)
%eax -> Vint (j - i)
%eax -> Vint (j - i - 1)

notl %eax
addl %ecx, %eax
%eax -> Vptr b i, %ecx -> Vptr b j

subl %eax, %ecx
movl %ecx, %eax
decl %eax

%ecx -> Vint (j - i)
%eax -> Vint (j - i)
%eax -> Vint (j - i - 1)

%eax -> Vundef

notl %eax
addl %ecx, %eax
%eax -> Vptr b i,   %ecx -> Vptr b j

\[
\begin{align*}
\text{subl} &\ %eax,\ %ecx \\
\text{movl} &\ %ecx,\ %eax \\
\text{decl} &\ %eax
\end{align*}
\]

%ecx -> Vint (j - i)
%eax -> Vint (j - i)
%eax -> Vint (j - i - 1)

%eax -> Vundef
%eax -> Vundef

notl %eax
addl %ecx, %eax
%eax -> Vptr b i, %ecx -> Vptr b j

subl %eax, %ecx
movl %ecx, %eax
decl %eax

%ecx -> Vint (j - i)
%eax -> Vint (j - i)
%eax -> Vint (j - i - 1)

notl %eax
addl %ecx, %eax

%eax -> Vundef
%eax -> Vundef

Vint (j - i - 1) <=> Vundef
Takeaway

• CompCert x86 can subtract pointers, can’t bitwise not pointers
• intel x86 can do both (they’re just bits)
• CompCert isn’t wrong, it’s conservative
• Conservativity limits valid rewrites
Takeaway

• Instructions are typeless

• Values are typed

• Mismatch makes for many spurious cases to prove, and these cases are frequently unprovable
Future Directions

• Mismatch of presence of types. Either:
  • Remove types from values
  • Carry types down for instructions
Conclusion

We built a verified* peephole optimizer for CompCert x86

* Assuming Calling Convention

Now increasing utility by exploring finer grained models of x86
Questions?
Suggestions?
Email Me!

emullen@cs.washington.edu