Superoptimization

How fast can your code go?

James Pallister
University of Bristol & Embecosm
What is superoptimization?

Unoptimized code
What is superoptimization?

Unoptimized code

Compiler optimized code
What is superoptimization?

Unoptimized code

Compiler optimized code

Superoptimized code
Plan for today

What is superoptimization?

Latest developments

The GNU Superoptimizer
Plan for today

What is superoptimization?

Latest developments

The GNU Superoptimizer
int sign(int n)
{
    if(n > 0)
        return 1;
    else if(n < 0)
        return -1;
    else
        return 0;
}
int sign(int n)
{
    if(n > 0)
        return 1;
    else if(n < 0)
        return -1;
    else
        return 0;
}
Superoptimization in action

```c
int sign(int n)
{
    if(n > 0)
        return 1;
    else if(n < 0)
        return -1;
    else
        return 0;
}
```

```assembly
cmp.l d0, 0
ble L1
move.l d1, 1
bra L3
L1:
bge L2
move.l d1, -1
bra L3
L2:
move.l d1, 0
L3:
```
Superoptimization in action

```c
int sign(int n)
{
    if(n > 0)
        return 1;
    else if(n < 0)
        return -1;
    else
        return 0;
}
```

```assembly
add.l  d0, d0
subx.l d1, d1
negx.l d0
addx.l d1, d1
```
Superoptimization in action

```c
int sign(int n)
{
    if(n > 0)
        return 1;
    else if(n < 0)
        return -1;
    else
        return 0;
}
```

```assembly
cmp.l d0, 0
ble L1
move.l d1, 1
bra L3
L1:
bge L2
move.l d1, -1
bra L3
L2:
move.l d1, 0
add.l d0, d0
subx.l d1, d1
negx.l d0
addx.l d1, d1
L3:
```
How does it work?

d0 ← n

\[
\begin{align*}
\text{add.l } & \quad d0, d0 \\
\text{subx.l } & \quad d1, d1 \\
\text{negx.l } & \quad d0 \\
\text{addx.l } & \quad d1, d1 \\
\end{align*}
\]

d1 → sign(n)
How does it work?

\[
\begin{array}{c|c|c}
\text{d0} & \text{d0} & \text{d1} \\
\hline
0 & -3 & \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{d0} & \text{d0} & \text{d1} \\
\hline
0 & 0 & \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{d0} & \text{d0} & \text{d1} \\
\hline
0 & 2 & \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{d0} & \rightarrow \text{sign(n)} \\
\end{array}
\]

- \text{d0} \leftarrow \text{n}
- \text{add.l} \text{ d0, d0}
- \text{subx.l} \text{ d1, d1}
- \text{negx.l} \text{ d0}
- \text{addx.l} \text{ d1, d1}
- \text{d1} \rightarrow \text{sign(n)}
How does it work?

d0 ← n

\[
\begin{array}{c}
0 & -3 \\
\end{array}
\]

\[
\begin{array}{c}
1 & -6 \\
\end{array}
\]

\[
\begin{array}{c}
0 & 0 \\
\end{array}
\]

\[
\begin{array}{c}
0 & 2 \\
\end{array}
\]

\[
\begin{array}{c}
0 & 4 \\
\end{array}
\]

add.l d0, d0
subx.l d1, d1
negx.l d0
addx.l d1, d1

d1 → sign(n)
How does it work?

d0 ← n
add.l d0, d0
subx.l d1, d1
negx.l d0
addx.l d1, d1
d1 → sign(n)
How does it work?

d0 ← n
add.l d0, d0
subx.l d1, d1
negx.l d0
addx.l d1, d1

d1 → sign(n)
How does it work?

d0 ← n

`d1 ← sign(n)`

<table>
<thead>
<tr>
<th>x</th>
<th>d0</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>d0</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>d0</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-4</td>
<td>1</td>
</tr>
</tbody>
</table>
How does it work?

d0 ← n

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>d0</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>-3</td>
<td></td>
</tr>
</tbody>
</table>

add.l d0, d0

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>d0</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>-6</td>
<td></td>
</tr>
</tbody>
</table>

subx.l d1, d1

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>d0</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>-6</td>
<td>-1</td>
</tr>
</tbody>
</table>

negx.l d0

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>d0</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>6</td>
<td>-1</td>
</tr>
</tbody>
</table>

addx.l d1, d1

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>d0</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>6</td>
<td>-1</td>
</tr>
</tbody>
</table>

d1 → sign(n)

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>d0</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>d0</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>d0</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
“A look at the smallest program”

A recent example: Atmel AVR
A recent example: Atmel AVR

Is $x$ a power of 2?
A recent example: Atmel AVR

Is x a power of 2?

\[ r = !((x - 1) \& x) \&\& x; \]
A recent example: Atmel AVR

Is x a power of 2?

\[ r = !((x-1) \& x) \&\& x; \]

Compiled code

```
mov  r20, r24
ldi  r21, 0
mov  r22, r20
mov  r19, r21
subi r22, 1
sbc  r19, r1
and  r22, r20
and  r19, r21
or   r22, r19
brne .+0
ldi  r25, 0x01
cpse r24, 1
rjmp .+0
ldi  r25, 0
mov  r24, r25
ret
ldi  r24, 0
ret
```
A recent example: Atmel AVR

Is \(x\) a power of 2?

\[ r = !((x-1) \& x) \&\& x; \]

<table>
<thead>
<tr>
<th>Compiled code</th>
<th>Superoptimized code</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov r20, r24</td>
<td>mov r1, r0</td>
</tr>
<tr>
<td>ldi r21, 0</td>
<td>dec r1</td>
</tr>
<tr>
<td>mov r22, r20</td>
<td>eor r0, r1</td>
</tr>
<tr>
<td>mov r19, r21</td>
<td>sub r1, r0</td>
</tr>
<tr>
<td>subi r22, 1</td>
<td>adc r1, r1</td>
</tr>
<tr>
<td>sbc r19, r1</td>
<td>and r1, r0</td>
</tr>
<tr>
<td>and r22, r20</td>
<td>ret</td>
</tr>
<tr>
<td>and r19, r21</td>
<td></td>
</tr>
<tr>
<td>or r22, r19</td>
<td></td>
</tr>
<tr>
<td>brne .+0</td>
<td></td>
</tr>
<tr>
<td>ldi r25, 0x01</td>
<td></td>
</tr>
<tr>
<td>cpse r24, 1</td>
<td></td>
</tr>
<tr>
<td>rjmp .+0</td>
<td></td>
</tr>
<tr>
<td>ldi r25, 0</td>
<td></td>
</tr>
<tr>
<td>mov r24, r25</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
<tr>
<td>ldi r24, 0</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>
Superoptimization fundamentals: Enumeration
Generating the sequences of instructions
Superoptimization fundamentals: Enumeration

Generating the sequences of instructions
- But doing them *all* takes far too long
Superoptimization fundamentals: Enumeration

Generating the sequences of instructions

- But doing them all takes far too long
Generating the sequences of instructions

- But doing them *all* takes far too long

Superoptimization fundamentals: Enumeration

Instruction set

- Longer sequences
- Possible suboptimal result
- Short sequences
- Optimal result
Generating the sequences of instructions

- But doing them all takes far too long

How to select the sequences of instructions?
Superoptimization fundamentals: Pruning

Instruction set
Not all instruction sequences are valid.
Not all instruction sequences are valid. How do we quickly ignore bad sequences?

Instruction set
Not all instruction sequences are valid.
How do we quickly ignore bad sequences?

Instruction set
Not all instruction sequences are valid.

How do we quickly ignore bad sequences?

Instruction set

Register renaming
add r0,r1 = add r2,r3
Superoptimization fundamentals: Pruning

Not all instruction sequences are valid.
How do we quickly ignore bad sequences?

Register renaming
\[ \text{add } r0, r1 = \text{add } r2, r3 \]

Commutativity
\[ A + B = B + A \]

Instruction set
Not all instruction sequences are valid.

How do we quickly ignore bad sequences?

- **Register renaming**: $\text{add } r0, r1 = \text{add } r2, r3$
- **Commutativity**: $A + B = B + A$
- **Redundant computation**: move $r0, r0$

Instruction set
Not all instruction sequences are valid.

How do we quickly ignore bad sequences?

Register renaming
\[ \text{add } r0, r1 = \text{add } r2, r3 \]

Commutativity
\[ A + B = B + A \]

Redundant computation
\[ \text{move } r0, r0 \]

Unused results

Instruction set

Superoptimization fundamentals: Pruning
Superoptimization fundamentals: Testing
Is the sequence correct?
Superoptimization fundamentals: Testing

Is the sequence correct?

Testing (simulation)
Superoptimization fundamentals: Testing

Is the sequence correct?

Testing (simulation)

Mathematical proof (symbolic solving)

```
5 3 7
6 1 9 5
9 8 6
8 4 1
```

- \( x + x = x \)
- \( y + x = x + y \)
- \( x + y = y + x \)
- \( (x + y)z = x(y + z) \)
- \( x(y + z) = xy + xz \)
- \( (x + y)z = x(y + z) \)
- \( x(y + z) = xy + xz \)
- \( x + y = x + y \)
- \( y + x = y + x \)
- \( x + x = x \)
- \( y + y = y + y \)
- \( x + y = x + y \)
- \( y + x = y + x \)
- \( x + x = x \)
- \( y + y = y + y \)
- \( x + x = x \)
- \( y + y = y + y \)
```
Superoptimization fundamentals: Testing

Is the sequence correct?

Testing (simulation)

Mathematical proof (symbolic solving)

1. Choose some input
2. Run/simulate
3. Check output
Superoptimization fundamentals: Testing

Is the sequence correct?

Mathematical proof (symbolic solving)

Formal verification
Proves the sequence correct
Slow

1. Choose some input
2. Run/simulate
3. Check output
Superoptimization fundamentals: Testing

Is the sequence correct?

Testing (simulation)

Mathematical proof (symbolic solving)

Use Both

1. Choose some input
2. Run/simulate
3. Check output

Formal verification
Proves the sequence correct
Slow
Superoptimization fundamentals: Costing
Which sequence is the best?
Which sequence is the best?

Execution time
Superoptimization fundamentals: Costing

Which sequence is the best?

Execution time

Code size
Which sequence is the best?

- Execution time
- Code size
- Energy consumption
Superoptimization fundamentals: Costing

Which sequence is the best?

- Execution time
- Code size
- Energy consumption

If you can enumerate the instructions in cost order, the first correct sequence is the optimal sequence.
Plan for today

What is superoptimization?

*Latest developments*

The GNU Superoptimizer
Search space pruning
Search space pruning

Restrict parameters

- Registers
  - 50% of instruction sequences of length 8 use less than 4 registers
- Immediate constants
  - Frequently used constants: -16 to +16, \(2^n\), \(2^n-1\)
Search space pruning

Restrict parameters

- Registers
  - 50% of instruction sequences of length 8 use less than 4 registers
- Immediate constants
  - Frequently used constants: -16 to +16, $2^n$, $2^n-1$

Remove meaningless constructs

- `mov r0, r0`
- `add r0, r0, #0`
Search space pruning

Restrict parameters

- Registers
  - 50% of instruction sequences of length 8 use less than 4 registers

- Immediate constants
  - Frequently used constants: -16 to +16, $2^n$, $2^n-1$

Remove meaningless constructs

- `mov r0, r0`
- `add r0, r0, #0`

Canonical form
mov r1, r0 has many equivalent versions
Canonical form

mov r1, r0 has many equivalent versions

Rename each register so they appear in sequence:

- mov r1, r0
- mov r4, r2
- mov r2, r8
- mov r1, r0
Canonical form

`mov r1, r0` has many equivalent versions

Rename each register so they appear in sequence:

\[
\begin{align*}
\text{mov } & r1, r0 \\
\text{mov } & r4, r2 \\
\text{mov } & r2, r8 \\
\text{mov } & r1, r0
\end{align*}
\]

With 16 registers this replaces \(16\times15\) equivalent versions
Canonical form

\[
\begin{align*}
\text{add} & \quad r4, \quad r8, \quad r1 \\
\text{orr} & \quad r8, \quad r4, \quad #1 \\
\text{sub} & \quad r1, \quad r2, \quad #8 \\
\text{add} & \quad r2, \quad r1, \quad r0 \\
\text{orr} & \quad r1, \quad r2, \quad #1 \\
\text{sub} & \quad r0, \quad r3, \quad #8
\end{align*}
\]
Canonical form

```
add r4, r8, r1
orr r8, r4, #1
sub r1, r2, #8

add r2, r1, r0
orr r1, r2, #1
sub r0, r3, #8
```

Single three operand instruction:

```
add rX, rX, rX
```

5 unique forms
Canonical form – reduction
Data processing instructions

- 16 ops, each using 3 of 16 possible registers.

- E.g. \textbf{add} r0, r1, r2

\textbf{sub} r3, r4, r5
Canonical form – reduction

Data processing instructions

- 16 ops, each using 3 of 16 possible registers.
- E.g. \texttt{add r0, r1, r2}
  \texttt{sub r3, r4, r5}

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Normal</th>
<th>Canonical</th>
<th>Canonical (4 registers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>65,536</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>2</td>
<td>4,294,967,296</td>
<td>51,968</td>
<td>47,872</td>
</tr>
<tr>
<td>3</td>
<td>281,474,976,710,656</td>
<td>4,157,669,376</td>
<td>45,264,896</td>
</tr>
<tr>
<td>4</td>
<td>18,446,744,073,709,551,616</td>
<td>276,142,292,992</td>
<td>45,880,115,200</td>
</tr>
</tbody>
</table>
Canonical form – reduction

Data processing instructions

- 16 ops, each using 3 of 16 possible registers.
- E.g. \textbf{add} r0, r1, r2
  \textbf{sub} r3, r4, r5

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Normal</th>
<th>Canonical</th>
<th>Canonical (4 registers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>65,536</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>2</td>
<td>4,294,967,296</td>
<td>51,968</td>
<td>47,872</td>
</tr>
<tr>
<td>3</td>
<td>281,474,976,710,656</td>
<td>4,157,669,376</td>
<td>45,264,896</td>
</tr>
<tr>
<td>4</td>
<td>18,446,744,073,709,551,616</td>
<td>276,142,292,992</td>
<td>45,880,115,200</td>
</tr>
</tbody>
</table>

@200,000 tests/second 2.9 million years
Data processing instructions

- 16 ops, each using 3 of 16 possible registers.
- E.g. \texttt{add r0, r1, r2}
  \texttt{sub r3, r4, r5}

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Normal</th>
<th>Canonical</th>
<th>Canonical (4 registers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>65,536</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>2</td>
<td>4,294,967,296</td>
<td>51,968</td>
<td>47,872</td>
</tr>
<tr>
<td>3</td>
<td>281,474,976,710,656</td>
<td>4,157,669,376</td>
<td>45,264,896</td>
</tr>
<tr>
<td>4</td>
<td>18,446,744,073,709,551,616</td>
<td>276,142,292,992</td>
<td>45,880,115,200</td>
</tr>
</tbody>
</table>

@200,000 tests/second  2.9 million years  16 days
Data processing instructions

- 16 ops, each using 3 of 16 possible registers.
- E.g. \texttt{add r0, r1, r2} \texttt{sub r3, r4, r5}

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Normal</th>
<th>Canonical</th>
<th>Canonical (4 registers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>65,536</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>2</td>
<td>4,294,967,296</td>
<td>51,968</td>
<td>47,872</td>
</tr>
<tr>
<td>3</td>
<td>281,474,976,710,656</td>
<td>4,157,669,376</td>
<td>45,264,896</td>
</tr>
<tr>
<td>4</td>
<td>18,446,744,073,709,551,616</td>
<td>276,142,292,992</td>
<td>45,880,115,200</td>
</tr>
</tbody>
</table>

@200,000 tests/second 2.9 million years 16 days <3 days
Instruction costing
Instruction costing

Sequence cost is simple if code size is to be minimised
Sequence cost is simple if code size is to be minimised.

Difficult to accurately measure the performance of short sequences of instructions.
- Pipeline modelling
- Cycle accurate simulation
Sequence cost is simple if code size is to be minimised.

Difficult to accurately measure the performance of short sequences of instructions.
- Pipeline modelling
- Cycle accurate simulation

Energy
- Total Software Energy and Reporting (TSERO)
Instruction sets
Characteristics of the instruction set affect how well a superoptimizer will perform.
Instruction sets

Characteristics of the instruction set affect how well a superoptimizer will perform.

Smaller instruction set $\rightarrow$ fewer optimal sequences (?)
Characteristics of the instruction set affect how well a superoptimizer will perform.

Smaller instruction set $\rightarrow$ fewer optimal sequences (?)
Instruction sets

Characteristics of the instruction set affect how well a superoptimizer will perform.

Smaller instruction set $\rightarrow$ fewer optimal sequences (?)

Large instruction set

Many short sequences

Small instruction set

Few longer sequences
Characteristics of the instruction set affect how well a superoptimizer will perform.

Smaller instruction set \(\rightarrow\) fewer optimal sequences (?)

Large instruction set

\[\begin{align*}
\text{Many short sequences} & \quad \text{Hard for standard compilers} \\
\end{align*}\]

Small instruction set

\[\begin{align*}
\text{Few longer sequences} & \\
\end{align*}\]
Characteristics of the instruction set affect how well a superoptimizer will perform.

Smaller instruction set $\rightarrow$ fewer optimal sequences (?)

- Large instruction set
  - Many short sequences
    - Hard for standard compilers
- Small instruction set
  - Few longer sequences
    - Easier for standard compilers
Superoptimizers
Superoptimizers

GNU SuperOptimizer (GSO)
Superoptimizers

GNU SuperOptimizer (GSO)

Denali
Superoptimizers

GNU SuperOptimizer (GSO)
Denali
Peephole Superoptimizer
Superoptimizers

GNU SuperOptimizier (GSO)
Denali
Peephole Superoptimizer
Stochastic Superoptimization
Superoptimizers

- GNU SuperOptimizer (GSO)
- Denali
- Peephole Superoptimizer
- Stochastic Superoptimization
- Other
  - A Hacker's Assistant (similar to GSO)
  - TOAST (similar to Denali)
GNU SuperOptimizer

Focuses on elimination of short basic blocks

GNU SuperOptimizer

Focuses on elimination of short basic blocks

```c
unsigned a, b, c;

if (a < b)
    c++;

    cmp   eax, ebx
    jge   L1
    add   ecx, #1

L1:
    ...
```

GNU SuperOptimizer

Focuses on elimination of short basic blocks

```c
unsigned a, b, c;

if(a < b)
    c++;

else
    c = (a < b) + c;
```

Denali

Rule based program generation:

Rule based program generation:

- Prove the condition: there exists no instruction sequence that solves \( F(x) \) in \(< K \) cycles.

Rule based program generation:

- Prove the condition: there exists no instruction sequence that solves $F(x)$ in $< K$ cycles
- Failure yields an example sequence

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td></td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

Rule based program generation:
- Prove the condition: there exists no instruction sequence that solves $F(x)$ in $< K$ cycles
- Failure yields an example sequence
- Use hand-written transformation rules

\[
\begin{align*}
x \ast 2 & \leftrightarrow x \ll 1 \\
a + b & \leftrightarrow b + a
\end{align*}
\]

Peephole Superoptimizers
Peephole Superoptimizers

Training Programs

Harvester -> Canonicalizer -> Fingerprinter

Check for match

Fingerprinter

Test case matcher

Fingerprint Hasetable

Enumerator
Peephole Superoptimizers

Training Programs

Harvester → Canonicalizer → Fingerprinter

Check for match

Fingerprinter

match? yes

Boolean Equivalence Test → pass

Optimization Database

Test case matcher

Fingerprint Hashtable

Enumerator
Peephole Superoptimizers

Input program

Compiler peephole pass

Canonicalizer

Fingerprinting

Lookup

Optimization Database

Better sequence found

Replace Input sequence
Stochastic superoptimization

A different approach to instruction sequence enumeration

Stochastic superoptimization

A different approach to instruction sequence enumeration

A different approach to instruction sequence enumeration

A different approach to instruction sequence enumeration

Longer sequences of instructions
- Sequences of >14 instructions were considered

A different approach to instruction sequence enumeration

Longer sequences of instructions

- Sequences of >14 instructions were considered
- E.g. OpenSSL Montgomery multiplication 60% faster

Discovering new algorithms
Discovering new algorithms

Correct programs

Space of all programs

Superoptimized

Algorithmically distinct programs

gcc -O3

llvm -O0
Discovering new algorithms

Correct programs

Space of all programs

Stochastic superoptimization's longer sequences make this more likely
Plan for today

What is superoptimization?

Latest developments

The GNU SuperOptimizer
The sign function, AVR:

```
cp    r1, r24
brlt  .+14
ldi   r25, 0x01
cpse  r24, r1
rjmp  .+2
ldi   r25, 0x00
mov   r24, r25
neg   r24
rjmp  .+2
ldi   r24, 0x01
```

Compiler (-Os)
11 instructions
4-10 cycles (if r1 initialised to 0)

Superoptimizer:
5 instructions
5 cycles
How does it work?

Iterative deepening depth first search

1 instruction
How does it work?

Iterative deepening depth first search

1 instruction

add r0, r0 → Check
How does it work?

Iterative deepening depth first search

1 instruction

add r0, r0 → Check
sub r0, r0 → Check
How does it work?

Iterative deepening depth first search

1 instruction

add r0, r0 → Check
sub r0, r0 → Check
mul r0, r0 → Check
How does it work?

Iterative deepening depth first search

2 instructions
How does it work?

Iterative deepening depth first search

2 instructions

add r0, r0
How does it work?

Iterative deepening depth first search

2 instructions

\[ \text{add } r0, r0 \rightarrow \text{Check} \]

\[ \text{add } r0, r0 \]
How does it work?

Iterative deepening depth first search

2 instructions

add r0, r0 → Check
sub r0, r0 → Check

add r0, r0
How does it work?

Iterative deepening depth first search

2 instructions

add r0, r0 → Check
sub r0, r0 → Check
mul r0, r0 → Check
How does it work?

Iterative deepening depth first search

2 instructions

add r0, r0
sub r0, r0

→ sub r0, r0 → Check
→ mul r0, r0 → Check
Iterative deepening depth first search

2 instructions

- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check
- add r0, r0 → Check
How does it work?

Iterative deepening depth first search

2 instructions

add r0, r0
sub r0, r0

add r0, r0 → Check
sub r0, r0 → Check
mul r0, r0 → Check

add r0, r0 → Check
sub r0, r0 → Check
How does it work?

Iterative deepening depth first search

2 instructions

- add r0, r0
- sub r0, r0
- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check
- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check
How does it work?

Iterative deepening depth first search

2 instructions

```
add r0, r0  \rightarrow \text{Check}
sub r0, r0  \rightarrow \text{Check}
mul r0, r0  \rightarrow \text{Check}
```

```
add r0, r0  \rightarrow \text{Check}
sub r0, r0  \rightarrow \text{Check}
mul r0, r0  \rightarrow \text{Check}
```

```
add r0, r0  \rightarrow \text{Check}
sub r0, r0  \rightarrow \text{Check}
mul r0, r0  \rightarrow \text{Check}
```
How does it work?

Iterative deepening depth first search

2 instructions

```plaintext
add r0, r0
sub r0, r0
mul r0, r0
```

```
add r0, r0 → Check
sub r0, r0 → Check
mul r0, r0 → Check
```

```
add r0, r0 → Check
sub r0, r0 → Check
mul r0, r0 → Check
```

...
How does it work?

Iterative deepening depth first search

3 instructions
How does it work?

Iterative deepening depth first search

3 instructions

add r0, r0
How does it work?

Iterative deepening depth first search

3 instructions

- add r0, r0
- add r0, r0
How does it work?

Iterative deepening depth first search

3 instructions

- add r0, r0
- add r0, r0
- add r0, r0 \(\rightarrow\) Check
How does it work?

Iterative deepening depth first search

3 instructions

add r0, r0

→ add r0, r0

→ sub r0, r0 → Check

→ add r0, r0 → Check
How does it work?

Iterative deepening depth first search

3 instructions

- add r0, r0
- sub r0, r0 → Check
- mul r0, r0 → Check
- add r0, r0
How does it work?

Iterative deepening depth first search

3 instructions

add r0, r0
sub r0, r0
mul r0, r0 → Check
add r0, r0 → Check
sub r0, r0 → Check
How does it work?

Iterative deepening depth first search

3 instructions

- add r0, r0
- sub r0, r0
- add r0, r0
- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check
- add r0, r0 → Check
Iterative deepening depth first search

3 instructions

- add r0, r0
- sub r0, r0
- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check
- add r0, r0 → Check
- sub r0, r0 → Check
How does it work?

Iterative deepening depth first search

3 instructions

- add r0, r0
- sub r0, r0

- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check
Iterative deepening depth first search

3 instructions

- add r0, r0
- sub r0, r0
- mul r0, r0

- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check

- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check
How does it work?

Iterative deepening depth first search

3 instructions

- add r0, r0
- sub r0, r0
- mul r0, r0

- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check

- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check

...
How does it work?

Iterative deepening depth first search

3 instructions

- add r0, r0
- sub r0, r0
- mul r0, r0

- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check

- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check

...
How does it work?

Iterative deepening depth first search

3 instructions

- add r0, r0
- sub r0, r0
- mul r0, r0

- add r0, r0 \rightarrow Check
- sub r0, r0 \rightarrow Check
- mul r0, r0 \rightarrow Check

...
How does it work?

Iterative deepening depth first search

3 instructions

- add r0, r0
- sub r0, r0
- mul r0, r0

- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check

- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check

...
How does it work?

Iterative deepening depth first search

3 instructions

add r0, r0
sub r0, r0
mul r0, r0

add r0, r0
sub r0, r0
mul r0, r0

add r0, r0 → Check
sub r0, r0 → Check
mul r0, r0 → Check

...
How does it work?

Iterative deepening depth first search

3 instructions

- add r0, r0
- sub r0, r0

- add r0, r0
- sub r0, r0
- mul r0, r0

- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check

...
How does it work?

Iterative deepening depth first search

3 instructions

- add r0, r0
- sub r0, r0
- add r0, r0
- sub r0, r0
- mul r0, r0
- add r0, r0
- sub r0, r0
- mul r0, r0

...
How does it work?

Iterative deepening depth first search

3 instructions

- add r0, r0
- sub r0, r0
- mul r0, r0

- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check

- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check

...
How does it work?

Iterative deepening depth first search

3 instructions

- add r0, r0
- sub r0, r0
- mul r0, r0

- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check

- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check

...
How does it work?

Iterative deepening depth first search

3 instructions

add r0, r0
sub r0, r0
mul r0, r0

add r0, r0 → Check
sub r0, r0 → Check
mul r0, r0 → Check

add r0, r0 → Check
sub r0, r0 → Check
mul r0, r0 → Check

add r0, r0 → Check
sub r0, r0 → Check
mul r0, r0 → Check

...
How does it work?

Iterative deepening depth first search

3 instructions

- add r0, r0
- sub r0, r0
- mul r0, r0

- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check

- add r0, r0 → Check
- sub r0, r0 → Check
- mul r0, r0 → Check

...
How to add a new goal
Goal is GSO's term for a target function to superoptimize
How to add a new goal

Goal is GSO's term for a target function to superoptimize

File: goal.def
How to add a new goal

*Goal* is GSO's term for a target function to superoptimize

File: *goal.def*

Examples:

```python
DEF_GOAL (SHIFTL_1, 1, "sll1", { r = v0 << 1; })
DEF_GOAL (SHIFTL_2, 1, "sll2", { r = v0 << 2; })
```
How to add a new goal

*Goal* is GSO's term for a target function to superoptimize

File: *goal.def*

Examples:

```plaintext
DEF_GOAL (SHIFTL_1, 1, "sll1", { r = v0 << 1; })
DEF_GOAL (SHIFTL_2, 1, "sll2", { r = v0 << 2; })
DEF_GOAL (SIGNUM, 1, "signum",
{
  if(v0 < 0)
    r = -1;
  else if(v0 > 0)
    r = 1;
  else
    r = 0;
})
```
How to add a new goal
How to add a new goal

DEF_GOAL (EQ_MINUS, 3, "eq-", { r = v2 - (v0 == v1); })
How to add a new goal

```
DEF_GOAL (EQ_MINUS, 3, "eq-", { r = v2 - (v0 == v1); })
```

**Format:**

```
DEF_GOAL (unique id, #args, "name", { C code; })
```

- `r` result
- `v0` argument 1
- `v1` argument 2
- `...`
How to add a new goal

DEF_GOAL (EQ_MINUS, 3, "eq-", { r = v2 - (v0 == v1); })

Format:

DEF_GOAL (unique id, #args, "name", { C code; })

- **r**  result
- **v0** argument 1
- **v1** argument 2
- ...

Using the new goal:

```
$ make all
$ ./superopt-avr -f name
```
How to add a new architecture
How to add a new architecture

Define the implementation in `superopt.h`
- Name, types (bitwidth)
How to add a new architecture

Define the implementation in `superopt.h`

- Name, types (bitwidth)
- “Unusual” instructions

```c
#ifndef PERFORM_AND
#define PERFORM_AND(d, co, r1, r2, ci) \  
    ((d) = (r1) & (r2), (co) = (ci))
#endif
```
How to add a new architecture

Define the implementation in `superopt.h`

- Name, types (bitwidth)
- “Unusual” instructions

```c
#ifndef PERFORM_AND
#define PERFORM_AND(d, co, r1, r2, ci) \
    ((d) = (r1) & (r2), (co) = (ci))
#endif
```

- Also add mnemonic into `insn.def`

```
DEF_INSN (AND, 'b', "and")
```

Identifier

Type (binary, unary, etc)

Name
How to add a new architecture
How to add a new architecture

To enable output in native assembler
How to add a new architecture

To enable output in native assembler

Extend `output_assembly`

- Register names
- Disassembly code
How to add a new architecture

To enable output in native assembler

Extend output_assembly

- Register names
- Disassembly code

```c
case ADD_CO:
    printf("add %s,%s",NAME(s1),NAME(s2)); break;

case ADD_CIO:
    printf("adc %s,%s",NAME(s1),NAME(s2)); break;
```
How to add a new architecture
How to add a new architecture

SYNTH(...) in synth.def
How to add a new architecture

SYNTH(...) in synth.def

- Where the magic happens
How to add a new architecture

SYNTH(...) in synth.def

- Where the magic happens
- Loops over all the registers and all the instructions
How to add a new architecture

SYNTH(...) in synth.def
- Where the magic happens
- Loops over all the registers and all the instructions
- Use the macros for performing the instruction
How to add a new architecture

SYNTH(...) in synth.def
  - Where the magic happens
  - Loops over all the registers and all the instructions
  - Use the macros for performing the instruction
  - Then recurses (next level of iterative deepening)
How to add a new architecture

SYNTH(...) in synth.def

- Where the magic happens
- Loops over all the registers and all the instructions
- Use the macros for performing the instruction
- Then recurses (next level of iterative deepening)
- Add architecture entries for standard & custom instructions
How to add a new architecture

SYNTH(...) in synth.def
- Where the magic happens
- Loops over all the registers and all the instructions
- Use the macros for performing the instruction
- Then recurses (next level of iterative deepening)
- Add architecture entries for standard & custom instructions

```c
#if SPARC || M88000
  /* sparc:       addx
     m88000:     addu.ci */
  PERFORM_ADD_CI(v, co, r1, r2, ci);
  RECURSE(ADD_CI, s1, s2, prune_hint & ~CY_JUST_SET);
#endif
```
#if SPARC || M88000
    /* sparc: addx
       m88000: addu.ci */
    PERFORM_ADD_CI(v, co, r1, r2, ci);
    RECURSE(ADD_CI, s1, s2, prune_hint & ~CY_JUST_SET);
#endif
How to add a new architecture

```c
#if SPARC || M88000
    /* sparc:       addx
       m88000:      addu.ci */
    PERFORM_ADD_CI(v, co, r1, r2, ci);
    RECURSE(ADD_CI, s1, s2, prune_hint & ~CY_JUST_SET);
#endif
```

Identifier (insn.def)
How to add a new architecture

```c
#if SPARC || M88000
    /* sparc: addx  
m88000:   addu.ci */
    PERFORM_ADD_CI(v, co, r1, r2, ci);
    RECURSE(ADD_CI, s1, s2, prune_hint & ~CY_JUST_SET);
#endif
```

Identifier (insn.def)

Registers used
How to add a new architecture

```c
#if SPARC || M88000
   /* sparc: addx
      m88000: addu.ci */
   PERFORM_ADD_CI(v, co, r1, r2, ci);
   RECURSE(ADD_CI, s1, s2, prune_hint & ~CY JUST SET);
#endif
```

Identifier (insn.def)

Identifier

Registers used

Speed hints.

(we didn't just set the carry)
Our plans to extend GSO
Our plans to extend GSO

Machine learning
Our plans to extend GSO

Machine learning

Parallelism
Our plans to extend GSO

Machine learning

Parallelism

Instruction sequence verification
Our plans to extend GSO

Machine learning  Parallelism  Instruction sequence verification

More instruction types
Our plans to extend GSO

- Machine learning
- Parallelism
- Instruction sequence verification
- More instruction types
- Memory access
Our plans to extend GSO

- Machine learning
- Parallelism
- Instruction sequence verification
- More instruction types
- Memory access
- Floating point
Our plans to extend GSO

- Machine learning
- Parallelism
- Instruction sequence verification
- More instruction types
- Memory access
- Floating point
- Multiple outputs
Thank You
Thank You

Superoptimization works
Superoptimization works

New techniques are making it better
Superoptimization works

New techniques are making it better

Lots of *free* software and tools
Superoptimization works

New techniques are making it better

Lots of *free* software and tools
Superoptimization works

New techniques are making it better

Lots of free software and tools

Try it yourself

github.com/embecosm/gnu-superopt