Scalability in Compiler Development
How to Get Testing and Optimization Done in a Reasonable Time

Jeremy Bennett
Machine Learning Compilers
• Initial research in 2012 by Embecosm and Bristol University
Do Compilers Affect Energy?

- Initial research in 2012 by Embecosm and Bristol University
- The answer is “yes”
Do Compilers Affect Energy?

- Initial research in 2012 by Embecosm and Bristol University
- The answer is “yes”
- Now published open access in a peer-reviewed journal

Identifying Compiler Options to Minimize Energy Consumption for Embedded Platforms
James Pallister; Simon J. Hollis; Jeremy Bennett
Research into feedback directed optimization
Research into feedback directed optimization

Research into modeling energy usage
Research into feedback directed optimization

Research into modeling energy usage
Research into feedback directed optimization

Research into modeling energy usage

Energy measurement
MAGEEC: Machine Guided Energy Efficient Compilation

Research into feedback directed optimization

Research into modeling energy usage

Energy measurement

Copyright © 2013 Embecosm. Freely available under a Creative Commons license
MAGEEC: **Machine Guided Energy Efficient Compilation**

- Research into feedback directed optimization
- Research into modeling energy usage
- Energy measurement

Copyright © 2013 Embecosm. Freely available under a Creative Commons license
Compiler

GCC

Copyright © 2013 Embecosm. Freely available under a Creative Commons license
Overall Design

Compiler

Compiler Plugin

Copyright © 2013 Embecosm. Freely available under a Creative Commons license
Building the Database

MAGEEC built with -DMAGEEC_FILEML
Building the Database

DejaGnu
Test Harness

MAGEEC built with -DMAGEEC_FILEML
DejaGnu

Test Harness

$MAGEEC_EXECUTE LIST

MAGEEC built with
-DMAGEEC_FILEML

Copyright © 2013 Embecosm. Freely available under a Creative Commons license
Building the Database

Log file

$MAGEEC_EXECUTELIST

MAGEEC built with -DMAGEEC_FILEML

DejaGnu
Test Harness

Copyright © 2013 Embecosm. Freely available under a Creative Commons license
Building the Database

Test Harness

DejaGnu

Log file

$MAGEEC_EXECUTELIST

Machine Learning once, after all tests run

MAGEEC built with -DMAGEEC_FILEML

Database

Copyright © 2013 Embecosm. Freely available under a Creative Commons license
• Run many, many times
• How do we choose the passes to run?

DejaGnu
Test Harness

Machine Learning
once, after all tests run

MAGEEC built with
-DMAGEEC_FILEML

$MAGEEC_EXECUTELIST

Log file
• From all combinations, we can find the impact of one option.
• From all combinations, we can find the impact of one option.
  - example with three options, $x_0$, $x_1$ and $x_2$. 
• From all combinations, we can find the impact of one option.
  – example with three options, $x_0$, $x_1$ and $x_2$.

\[ x_1 = \frac{\sum \text{blue}}{4} - \frac{\sum \text{green}}{4} \]
\( x_2 = \frac{\sum \bullet}{4} - \frac{\sum \circ}{4} \)

- The same data give us the other options as well
The same data give us the other options as well

\[
\chi_2 = \frac{\sum \bullet}{4} - \frac{\sum \circ}{4}
\]

We need a total of 8 runs
• The same data give us the other options as well

\[ x_2 = \frac{\sum \text{blue}}{4} - \frac{\sum \text{green}}{4} \]

• We need a total of 8 runs
  - but what if we had 250 options?
• From a subset, we can find the impact of one option.
Fractional Factorial Design

- From a subset, we can find the impact of one option.
  - same example with three options, $x_0, x_1$ and $x_2$. 
• From a subset, we can find the impact of one option.
  – same example with three options, $x_0$, $x_1$ and $x_2$.

$$x_1 = \frac{\sum \bullet}{2} - \frac{\sum \circ}{2}$$
• The same data give us all the options.
  - by choosing a different combination of data points

\[ x_2 = \frac{\sum \bullet}{2} - \frac{\sum \circ}{2} \]
The same data give us all the options.
- by choosing a different combination of data points

\[ x_2 = \frac{\sum \bullet}{2} - \frac{\sum \circ}{2} \]

We need a total of 4 runs
Fractional Factorial Design

- The same data give us all the options.
  - by choosing a different combination of data points

\[ x_2 = \frac{\Sigma \bullet}{2} - \frac{\Sigma \circ}{2} \]

- We need a total of 4 runs
  - but it could be \( x_0 \) and \( x_1 \) acting together
Gains are more significant with more factors

\[ x_2 = \frac{\Sigma \text{blue}}{4} - \frac{\Sigma \text{green}}{4} \]
• Gains are more significant with more factors
  - deal with multiple factor interaction

\[ x_2 = \frac{\sum \text{blue}}{4} - \frac{\sum \text{green}}{4} \]
• Gains are more significant with more factors
  - deal with multiple factor interaction
  - challenge is tools

\[ \chi^2 = \frac{\sum \text{blue}}{4} - \frac{\sum \text{green}}{4} \]
- Gains are more significant with more factors
  - deal with multiple factor interaction
  - challenge is tools
  - current limit is 120 factors

\[ \chi_2 = \frac{\Sigma \text{blue}}{4} - \frac{\Sigma \text{green}}{4} \]
Building the Database

DejaGnu
Test Harness

Machine Learning

FFD Generator

Log file

MAGEEC built with -DMAGEEC_FILEML

$MAGEEC_EXECUTELIST

Copyright © 2013 Embecosm. Freely available under a Creative Commons license
Scalability

- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
Scalability

- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
- Approx 100 BEEBS tests
  - run 6 boards at once
  - 67s per test run
Scalability

- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
- Approx 100 BEEBS tests
  - run 6 boards at once
  - 67s per test run
- 200+ optimization passes
  - $2^{200}$ possibilities
Scalability

- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
- Approx 100 BEEBS tests
  - run 6 boards at once
  - 67s per test run
- 200+ optimization passes
  - $2^{200}$ possibilities
  - FFD reduces this
Scalability

- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
- Approx 100 BEEBS tests
  - run 6 boards at once
  - 67s per test run
- 200+ optimization passes
  - $2^{200}$ possibilities
  - FFD reduces this
  - to $2^{16} = 65,536$ runs
Scalability

- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
- Approx 100 BEEBS tests
  - run 6 boards at once
  - 67s per test run
- 200+ optimization passes
  - $2^{200}$ possibilities
  - FFD reduces this
  - to $2^{16} = 65,536$ runs
  - = 4,369,067s
Each AVR test takes 4s
- 2s to flash device
- 2s to run test

Approx 100 BEEBS tests
- run 6 boards at once
- 67s per test run

200+ optimization passes
- $2^{200}$ possibilities
- FFD reduces this
- to $2^{16} = 65,536$ runs
- $= 4,369,067s$
- $= 50d 13h 37m 47s$
- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
- Approx 100 BEEBS tests
  - run 6 boards at once
  - 67s per test run
- 200+ optimization passes
  - $2^{200}$ possibilities
  - FFD reduces this
to $2^{16} = 65,536$ runs
  - = 4,369,067s
  - = 50d 13h 37m 47s
- Oh dear 😞
Scalability

- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
- Approx 100 BEEBS tests
  - run 6 boards at once
  - 67s per test run
- 200+ optimization passes
  - $2^{200}$ possibilities
  - FFD reduces this
  - to $2^{16} = 65,536$ runs
  - $= 4,369,067s$
  - $= 50d 13h 37m 47s$
- Oh dear 😞

One compiler on one CPU
Scalability

- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
- Approx 100 BEEBS tests
  - run 6 boards at once
  - 67s per test run
- 200+ optimization passes
  - $2^{200}$ possibilities
  - FFD reduces this to $2^{16} = 65,536$ runs
  - $= 4,369,067s$
  - $= 50d 13h 37m 47s$
- Oh dear 😞

One compiler on one CPU
Atmel have 200+ AVR variants
• A special case of FFD
Plackett Burman to the Rescue

- A special case of FFD
- One more run than the number of factors (passes).
• A special case of FFD
• One more run than the number of factors (passes).
• Assumes independence of factors.
Placket Burman to the Rescue

- A special case of FFD
- One more run than the number of factors (passes).
- Assumes independence of factors.
- 210 optimization passes means 211 test runs
  - 23h 27m on one board.
• A special case of FFD
• One more run than the number of factors (passes).
• Assumes independence of factors.
• 210 optimization passes means 211 test runs
  – 23h 27m on one board.
• Can then use FFD on most important passes
Plackett Burman to the Rescue

- A special case of FFD
- One more run than the number of factors (passes).
- Assumes independence of factors.
- 210 optimization passes means 211 test runs
  - 23h 27m on one board.
- Can then use FFD on most important passes
Superoptimization
Superoptimization is an old technique
Superoptimization is an old technique

There are free and open source implementations
- A Hacker's Assistant (Aha)
- the GNU Superoptimizer (GSO)
- all have limitations
Superoptimization is an old technique

There are free and open source implementations
- A Hacker's Assistant (Aha)
- the GNU Superoptimizer (GSO)
- all have limitations

Can we now build a commercially robust tool?
- computers are faster, algorithms have advanced
- what are the areas where this can be applied?
int sign (int n)
{
    if (n > 0)
        return 1;
    else if (n < 0)
        return -1;
    else
        return 0;
}
```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:
```
int sign (int n) {
    if (n > 0) {
        return 1;
    }
    else if (n < 0) {
        return -1;
    }
    else {
        return 0;
    }
}
d0 ← n

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

d1 → sign(n)
### How Does it Work?

<table>
<thead>
<tr>
<th></th>
<th>$d_0 \leftarrow n$</th>
<th>$d_1 \rightarrow \text{sign}(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_0 \leftarrow n$</td>
<td>$0 \rightarrow -3$</td>
<td>$0 \rightarrow 0$</td>
</tr>
</tbody>
</table>

- add.l $d_0, d_0$
- subx.l $d_1, d_1$
- negx.l $d_0$
- addx.l $d_1, d_1$
**How Does it Work?**

\[
\begin{align*}
\text{d0} & \leftarrow n \\
\text{add.l } & \text{ d0, d0} \\
\text{subx.l } & \text{ d1, d1} \\
\text{negx.l } & \text{ d0} \\
\text{addx.l } & \text{ d1, d1} \\
\text{d1} & \rightarrow \text{sign(n)}
\end{align*}
\]

\[
\begin{array}{c|c|c|c|c}
0 & 0 & 0 & 0 & 2 \\
0 & 1 & 0 & 0 & 0 \\
0 & -3 & 1 & 0 & 0 \\
\end{array}
\]
### How Does it Work?

<table>
<thead>
<tr>
<th>Operation</th>
<th>d0</th>
<th>d1</th>
<th>d0</th>
<th>d1</th>
<th>d0</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d0 \leftarrow n )</td>
<td>0</td>
<td>-3</td>
<td>1</td>
<td>-6</td>
<td>0</td>
<td>-6</td>
</tr>
<tr>
<td>( \text{addx}.l\ d0, d0 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( \text{negx}.l\ d0 )</td>
<td>0</td>
<td>-6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( \text{addx}.l\ d1, d1 )</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

\( d1 \rightarrow \text{sign}(n) \)
How Does it Work?

\[ d0 \leftarrow n \]

\[ \text{add.l} \ d0, \ d0 \]

\[ \text{subx.l} \ d1, \ d1 \]

\[ \text{negx.l} \ d0 \]

\[ \text{addx.l} \ d1, \ d1 \]

\[ d1 \rightarrow \text{sign}(n) \]
How Does it Work?

d0 ← 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>
</tbody>
</table>

add.l d0, d0

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

subx.l d1, d1

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

negx.l d0

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

addx.l d1, d1

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

d1 → sign(n)

<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>
</tbody>
</table>
**How Does it Work?**

\[ d0 \leftarrow n \]

1. \( \text{add.l} \ d0, \ d0 \)
2. \( \text{subx.l} \ d1, \ d1 \)
3. \( \text{negx.l} \ d0 \)
4. \( \text{addx.l} \ d1, \ d1 \)

\( d1 \rightarrow \text{sign}(n) \)

<table>
<thead>
<tr>
<th></th>
<th>( x )</th>
<th>( d0 )</th>
<th>( d1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>-6</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>-6</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>6</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>-4</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Copyright © 2013 Embecosm. Freely available under a Creative Commons license
Generating the sequences of instructions
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
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

How to select the sequences of instructions?
Instruction set
Not all instruction sequences are valid.

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?

Register renaming

\[ \text{add } r0, r1 = \text{add } r2, r3 \]
Not all instruction sequences are valid. How do we quickly ignore bad sequences?

Instruction set

Register renaming
\texttt{add r0, r1} = \texttt{add r2, r3}

Redundant computation
\texttt{move r0, r0}

Copyright © 2013 Embecosm. Freely available under a Creative Commons license
Not all instruction sequences are valid.

How do we quickly ignore bad sequences?

- Register renaming
  \[
  \text{add } r_0, r_1 = \text{add } r_2, r_3
  \]

- Redundant computation
  \[
  \text{move } r_0, r_0
  \]

- Commutativity
  \[
  A + B = B + A
  \]
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
Is the sequence correct?
Is the sequence correct?

Testing
(simulation)
Is the sequence correct?

Testing (simulation)

Mathematical proof (symbolic solving)
1. Choose some input
2. Run/simulate
3. Check output
Is the sequence correct?

Testing (simulation)

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

Mathematical proof (symbolic solving)

Formal verification
Proves the sequence correct
Slow
Is the sequence correct?

Testing (simulation)

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

Mathematical proof (symbolic solving)

Formal verification
Proves the sequence correct
Slow

Use Both
Which sequence is the best?
Which sequence is the best?

Execution time
Which sequence is the best?

Execution time

Code size
Which sequence is the best?

Execution time  
Code size  
Energy consumption
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.
Restrict parameters
Restrict parameters

- Registers
  - 50% of instruction sequences of length 8 use less than 4 registers
- Immediate constants
  - Frequently used constants: -16 to +16, $2^n$, $2^n-1$
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`
State of the Art
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
mov r1, r0 has many equivalent versions

Rename each register so they appear in sequence:

\[
\begin{align*}
\text{mov } r1, r0 \\
\text{mov } r4, r2 \\
\text{mov } r2, r8 \\
\text{mov } r1, r0
\end{align*}
\]
mov r1, r0 has many equivalent versions

Rename each register so they appear in sequence:

mov r1, r0
mov r4, r2
mov r2, r8

With 16 registers this replaces 16*15 equivalent versions
add r4, r8, r1
orr r8, r4, #1
sub r1, r2, #8

add r2, r1, r0
orr r1, r2, #1
sub r0, r3, #8
State of the Art
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

add r0, r0, r0
add r0, r0, r1
add r0, r1, r0
add r0, r1, r1
add r0, r1, r2
Data processing instructions

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

- 16 ops, each using 3 of 16 possible registers.
- E.g.  
  
  ```
  add r0, r1, r2
  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 } \texttt{r0, r1, r2} \\
  \texttt{sub } \texttt{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
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)
Characteristics of the instruction set affect how well a superoptimizer will perform.
Characteristics of the instruction set affect how well a superoptimizer will perform.

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

Smaller instruction set → fewer optimal sequences (?)
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 → fewer optimal sequences (?)

Large instruction set

Many short sequences

Hard for standard compilers

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: Many short sequences $\rightarrow$ Hard for standard compilers
- Small instruction set: Few longer sequences $\rightarrow$ Easier for standard compilers
State of the Art
Peephole Superoptimizers

Training Programs

Harvester → Canonicalizer → Fingerprinter

Test case matcher

Fingerprinter → Fingerprint Hasetable
State of the Art

Peephole Superoptimizers

Training Programs

Harvester → Canonicalizer → Fingerprinter

Check for match

Fingerprinter

Fingerprint Hashtable

Test case matcher

Enumerator
State of the Art

Peephole Superoptimizers

**Test case matcher**

1. **Training Programs**
   - Harvester
   - Canonicalizer
   - Fingerprinter

2. **Enumerator**
   - Fingerprinter
   - Check for match
   - Fingerprint Hashtable

3. **Match?**
   - Yes
     - Boolean Equivalence Test
     - Pass
     - Optimization Database
   - No
State of the Art
Peephole Superoptimizers

Your Program

Harvester → Canonicalizer → Fingerprinter

Optimization Database

Better sequence found

Test case matcher

Replace Input sequence
A different approach to instruction sequence enumeration
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
A different approach to instruction sequence enumeration

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

State of the Art
Discovering New Algorithms

Correct programs

Space of all programs

Algorithmically distinct programs

Superoptimized

gcc -O3

llvm -O0
Stochastic superoptimization's longer sequences make this more likely
GSO 2.0: A Superoptimizer Toolkit

- Machine state
- Instructions
- Slots
- Bruteforce iterator
- Canonical form iterator
- Constants iterator
- Stochastic iterator
- Parallelisation
- Instruction sequence testing
- Instruction equivalence checking
- Peephole superoptimizer testing

Copyright © 2013 Embecosm. Freely available under a Creative Commons license
Thank You

www.embecosm.com