Code Size Reduction by Difference Classification and Customized Lookup Table Generation

Subrato K. De (sde@qualcomm.com)
Kun Zhang (zhangk@qualcomm.com)
Tony Linthicum (tlinth@qualcomm.com)

80-VB419-71 Rev. A

QUALCOMM is a registered trademark of QUALCOMM Incorporated in the United States and may be registered in other countries. Other product and brand names may be trademarks or registered trademarks of their respective owners.
Terms and conditions of usage

This document may contain information regarding parts and products whose manufacture, use, sale, offer for sale, or importation into the United States is subject to restrictions under one or more U.S. injunctions against QUALCOMM Incorporated. This document is not to be construed as an offer to sell such parts or products for use or importation into the U.S. This document is intended solely to provide technical information regarding technical recommendations and/or requirements regarding uses and configurations of those parts or products inside and outside the United States that are permitted by such injunctions. Recipient’s download and/or use of the information in this document constitutes agreement with these terms.
Importance of code size reduction for mobile devices

- Benefits of smaller text code size
  - Less memory usage, enabling more functionality to reside simultaneously in on-chip RAM
  - Improved instruction cache performance
  - Reduced instruction bus traffic

- Undesired side effects
  - Some optimizations can degrade performance
  - Apply these to less time-critical portions of code

- Extensive control code and switch-case constructs in wireless networking, modem, and protocol code
  - Reduce code size by detecting similarities and replacing “compile time constant” differences as dictionary lookups
extern func1(int x, int y);
extern func2(int x, int y);
extern func3(int x, int y);
extern func4(int x, int y);
extern func5(int x, int y);
extern func6(int x, int y);
extern int g_array[];
int test(int i, int x)
{
    int a;
    switch(i)
    {
        case 1: a = 27 + x; break;
        case 2: a = 55 + x; break;
        case 3: a = 1024 + x; break;
        case 4: a = 23 + x; break;
        case 5: a = 129 + x; break;
        case 6: a = 256 + x; break;
        case 7: a = g_array[1]; break;
        case 8: a = g_array[6]; break;
        case 9: a = g_array[2]; break;
        case 10: a = g_array[9]; break;
        case 11: a = g_array[5]; break;
        case 12: a = g_array[4]; break;
        case 13: a = func1(x,1); break;
        case 14: a = func2(x,1); break;
        case 15: a = func3(x,1); break;
        case 16: a = func4(x,1); break;
        case 17: a = func5(x,1); break;
        case 18: a = func6(x,1); break;
        default: a = i; break;
    }
    return(a);
}
test:
  r8=#(.rodata)
  r6=r0
if (r0 >= #19)
  jump .Lt_0_2
r8=add(r8,r0<<#2)
  r9=loadw(r8+#0)
njmp r9
.Lt_0_2:
  r0=r6
  return;
.Lt_0_19:
  r0=r1
  r1=#1
call func6
  r6=r0
  jump .Lt_0_2
.Lt_0_18:
  r0=r1
  r1=#1
call func5
  r6=r0
  jump .Lt_0_2
.Lt_0_17:
  r0=r1
  r1=#1
call func4
  r6=r0
  jump .Lt_0_2
.Lt_0_16:
  r0=r1
  r1=#1
call func3
  r6=r0
  jump .Lt_0_2
.Lt_0_15:
  r0=r1
  r1=#1
call func2
  r6=r0
  jump .Lt_0_2
.Lt_0_14:
  r0=r1
  r1=#1
call func1
  r6=r0
  jump .Lt_0_2
.Lt_0_13:
  r6=#(g_array+16)
  r6=loadw(r6+#0)
  jump .Lt_0_2
.Lt_0_12:
  r6=#(g_array+20)
  r6=loadw(r6+#0)
  jump .Lt_0_2
.Lt_0_11:
  r6=#(g_array+36)
  r6=loadw(r6+#0)
  jump .Lt_0_2
.Lt_0_10:
  r6=#(g_array+8)
  r6=loadw(r6+#0)
  jump .Lt_0_2
.Lt_0_9:
  r6=#(g_array+24)
  r6=loadw(r6+#0)
  jump .Lt_0_2
.Lt_0_8:
  r6=#(g_array+4)
  r6=loadw(r6+#0)
  jump .Lt_0_2
.Lt_0_7:
  r6=add(r1,#256)
  jump .Lt_0_2
.Lt_0_6:
  r6=add(r1,#129)
  jump .Lt_0_2
.Lt_0_5:
  r6=add(r1,#23)
  jump .Lt_0_2
.Lt_0_4:
  r6=add(r1,#1024)
  jump .Lt_0_2
.Lt_0_3:
  r6=add(r1,#55)
  jump .Lt_0_2
.Lt_0_1:
  r6=add(r1,#27)
  jump .Lt_0_2

ORIGINAL
JUMP TABLE

.section
.rodata
.org 0x0

.word .Lt_0_2
.word .Lt_0_1
.word .Lt_0_3
.word .Lt_0_4
.word .Lt_0_5
.word .Lt_0_6
.word .Lt_0_7
.word .Lt_0_8
.word .Lt_0_9
.word .Lt_0_10
.word .Lt_0_11
.word .Lt_0_12
.word .Lt_0_13
.word .Lt_0_14
.word .Lt_0_15
.word .Lt_0_16
.word .Lt_0_17
.word .Lt_0_18
.word .Lt_0_19
Improved assembly code after detecting similarity and replacing differences using LUT

test:
  r8=#(.rodata)
  r6=r0

  if (r0 >= #19)
    jump .Lt_0_2

  r8=add(r8,r0<<#2)
  r9=loadw(r8+#0)

.Lt_Unchanged:
  if (r0 <= #0)
    jump r9

.Lt_SingleConst:
  if (r0 <= #6)
    jump .Lt_0_1

.Lt_MemOffset:
  if (r0 <= #12)
    jump .Lt_0_8

.Lt_SameCallSig:
  if (r0 <= #18)
    jump .Lt_0_14

.JUMP TABLE
NOW PARTLY BECOMES LUT

.section .rodata
.org 0x0
.word .Lt_0_2
.word 27
.word 55
.word 1024
.word 23
.word 129
.word 256
.word 4
.word 24
.word 8
.word 36
.word 20
.word 16
.word &func1
.word &func2
.word &func3
.word &func4
.word &func5
.word &func6
Difference items, types, and classes: encoding and decoding for LUT

**EXAMPLE ENCODING:**

<table>
<thead>
<tr>
<th>Memory offset</th>
<th>Logical-or operand</th>
<th>Extract bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>31st bit</td>
<td>15th bit</td>
<td>7th bit</td>
</tr>
</tbody>
</table>

**.Lt_10_13:**
- r17 = loaduh(r30+#-584)
- r17 = or(r17,#16)
- r17 = extract(r17,#10,#6)
- storeh(r30+#-584)=r17
- r9 = loaduh(r29+#16)
- r10 = loadw(r29+#600)
- jump .Lt_10_294

**.Lt_10_12:**
- r20 = loaduh(r30+#-584)
- r20 = or(r20,#8)
- r20 = extract(r20,#12,#6)
- storeh(r30+#-584)=r20
- r9 = loaduh(r29+#16)
- r10 = loadw(r29+#680)
- jump .Lt_10_294

**.Lt_10_11:**
- r21 = loaduh(r30+#-584)
- r21 = or(r21,#4)
- r21 = extract(r21,#14,#6)
- storeh(r30+#-584)=r21
- r9 = loaduh(r29+#16)
- r10 = loadw(r29+#720)
- jump .Lt_10_294

**DIFFERENCE CLASS**

**Difference Item1: Constant Operand**
- Difference Type 1: constant operand in “logical-or operation”
- Difference Type 2: bit width in extract operation

**Difference Item2: Memory Offset**
- Difference Type 3: memory offsets in load word, i.e., loadw
### Example where encoding requires additional LUT

#### .Lt_30_1:
- `r0` = `loadub(r25 + #124)`
- `r1` = `#33`
- `r2` = `loadw(r24 + #1020)`
- `r3` = `#125`
- `r4` = `#75`
- `r5` = `#85`
- `call Callee1`
- `jump .Lt_30_2`

#### .Lt_30_3:
- `r0` = `loadub(r25 + #248)`
- `r1` = `#66`
- `r2` = `loadw(r24 + #2040)`
- `r3` = `#10`
- `r4` = `#95`
- `r5` = `#51`
- `call Callee2`
- `jump .Lt_30_2`

#### .Lt_30_7:
- `r0` = `loadub(r25 + #492)`
- `r1` = `#55`
- `r2` = `loadw(r24 + #4088)`
- `r3` = `#114`
- `r4` = `#15`
- `r5` = `#49`
- `call Callee3`
- `jump .Lt_30_2`

---

**Difference Class**

<table>
<thead>
<tr>
<th>Item: Constant Operands</th>
<th>Item: Memory Offset</th>
<th>Item: Function with same signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Types: constants loaded in <code>r1</code>, <code>r3</code>, <code>r4</code>, <code>r5</code></td>
<td>• Types: memory offsets for <code>loadub</code> and <code>loadw</code></td>
<td>• Types: the function called</td>
</tr>
</tbody>
</table>

---

**EXAMPLE_ENCODING REQUIRES AT LEAST THREE WORDS (12 bytes):**

<table>
<thead>
<tr>
<th>Constant for “r5”</th>
<th>Constant for “r4”</th>
<th>Constant for “r3”</th>
<th>Constant for “r1”</th>
</tr>
</thead>
<tbody>
<tr>
<td>31st bit</td>
<td>23rd bit</td>
<td>15th bit</td>
<td>7th bit</td>
</tr>
<tr>
<td>unused</td>
<td>offset for “loadw to r2”</td>
<td>offset for “loadub to r0”</td>
<td></td>
</tr>
<tr>
<td>31st bit</td>
<td>24th bit</td>
<td>9th bit</td>
<td>0 bit</td>
</tr>
</tbody>
</table>

**Callee Function Address**

<table>
<thead>
<tr>
<th>31st bit</th>
<th>0 bit</th>
</tr>
</thead>
</table>
Highlights of difference classification algorithm

- Input is pair of code regions for detecting similarity
  - Detected hierarchically: control flow graph → basic blocks → individual operations → operands
  - TN’s (operands)
    - Globals
    - Locals – use-def chains determined and compared
- Differences detected at operands, functions with same signature called
- A list of pair-wise code regions is formed, arranged in descending order of “the number of difference types in difference class of pair”
- Code regions having the same difference class clustered together
  - All code regions in a cluster can be replaced by single representative
- Code regions that belong to a difference class that is a subset of another difference class can sometimes be clustered together (if possible and profitable)
- Opportunities exist to improve the algorithm
Venn diagram representation of difference classification and LUT generation

• \{1,2,3,\ldots,30,31,32\} indicate code regions that are compared
• A, B, C, D, E, F indicate “difference type”
• Each distinct region indicates “difference class”
  \{1,2,3,4\}, \{5,6,7\}, \{8,9\}, \ldots, \{27,28,29,30,31,32\}
Overall framework

Similar Instances of a particular control flow region type

Similar case statements determined from Jump table

Determine similar tail regions that merge at the same point

Cascaded sequence of similar BBs

Similar regions without specific control flow type

Difference Classification and Lookup Table Generation

Replace multiple similar case statements by a single representative

Common Interface

Merge similar tail regions that merge at the same point

Procedural Extraction

Generate loop using a similar BB as it’s body, count = # of BBs

Post Processing
Tail merging example

BB1
r13=add(r25,r26)
r14=r26
r15=r13+r14
r0=loadw(r6+#0)
r1=r10
r2=r11>>#2
r3=r2
r4=r13-r14
r5=#11
r6=r15+1
call msg_send_3

BB5
r12=#0
r13= #1
jump .LBB7_agc_init

BB2
r13=sub(r25,r26)
r14=r26
r15=r13+r14
r0=loadw(r6+#8)
r1=r10
r2=r11>>#2
r3=r2
r4=r13-r14
r5=#91
r6=r15+1
call msg_send_3

BB6
r13= #1
jump .LBB7_agc_init

BB4
r12=#0
r13= #1
jump .LBB7_agc_init

BB7
.LBB7_agc_init:
new BB
r14=r26
r15=r13+r14
Ry=#LUT_base_address;
Ry=Ry+Rz;
Rx=loadh(Ry+#0)
ro=extract(Rx,8,8); r0=r0+r6
r0=loadw(r0+0)
r1=r10
r2=r11>>#2
r3=r2
r4=r13-r14
r5=extract(Rx,8,0)
r6=r15+1
call msg_send_3

BB1
Rz=#0
r13=add(r25,r26)

BB2
Rz=#2
R13=sub(r25,r26)

BB4
r12=#0

BB9
.LBB9_agc_init:
r13= #1

BB7
.LBB7_agc_init:

BB6
jump .LBB9_agc_init

Look Up Table

<table>
<thead>
<tr>
<th></th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>91</td>
</tr>
</tbody>
</table>
Cascaded regions replaced by loop with differences in LUT

BB1
r0=loadw(r6+#0)
r1=r10
r2=#106
r3=r11
call msg_send_3

BB2
r0=loadw(r6+#48)
r1=r10
r2=#135
r3=r11
call msg_send_3

BB3
r0=loadw(r6+#36)
r1=r10
r2=#224
r3=r11
call msg_send_3

BB4
r0=loadw(r6+#64)
r1=r10
r2=#298
r3=r11
call msg_send_3

BB5
r0=loadw(r6+#72)
r1=r10
r2=#234
r3=r11
call msg_send_3

New BB 6: \( R_y = \#LUT\_base\_address \)
LoopCounter=#5

LoopStart:
\( R_x=loadw(R_y++) \)
\( R_z=extract(R_x,16,16) \)
\( R_z=r6+R_z; \)
\( r0=loadw(R_z); \)
\( r1=r10 \)
\( r2=extract(R_x,16,0) \)
\( r3=r11 \)
call msg_send_3
LoopEnd

New BB 7

LOOK UP TABLE:
Upper half-word is for load offset
Lower half-word is for constant loaded in r2

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>106</td>
</tr>
<tr>
<td>48</td>
<td>135</td>
</tr>
<tr>
<td>36</td>
<td>224</td>
</tr>
<tr>
<td>64</td>
<td>298</td>
</tr>
<tr>
<td>72</td>
<td>234</td>
</tr>
</tbody>
</table>
Procedural abstraction with differences in LUT

Look Up Table

<table>
<thead>
<tr>
<th></th>
<th>12</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>
### Code size comparison: Open64 and GCC (-Os)

Test cases are some of the functions in software for mobile device – networking protocol, modem, etc.

<table>
<thead>
<tr>
<th>Test cases</th>
<th>GCC 4.3.2 (size in bytes)</th>
<th>GCC 3.4.6 (size in bytes)</th>
<th>Original Open64 (size in bytes)</th>
<th>Methodology in Open64 (size in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pure text</td>
<td>rodata</td>
<td>Total text</td>
<td>Pure text</td>
</tr>
<tr>
<td>1</td>
<td>3368</td>
<td>6164</td>
<td>9532</td>
<td>3364</td>
</tr>
<tr>
<td>2</td>
<td>1152</td>
<td>308</td>
<td>1460</td>
<td>1224</td>
</tr>
<tr>
<td>3</td>
<td>1032</td>
<td>352</td>
<td>1384</td>
<td>1036</td>
</tr>
<tr>
<td>4</td>
<td>896</td>
<td>0</td>
<td>896</td>
<td>876</td>
</tr>
<tr>
<td>5</td>
<td>2204</td>
<td>9204</td>
<td>11408</td>
<td>2204</td>
</tr>
<tr>
<td>6</td>
<td>1052</td>
<td>1128</td>
<td>2180</td>
<td>1088</td>
</tr>
</tbody>
</table>

- **30% to 80% improvement for pure text (instructions only),**
- **5% to 60% improvement for total text (instructions and read only data)**

© Qualcomm Incorporated

80-VB419-71 Rev. A
## Code size and performance impact: Open64 (-Os)

<table>
<thead>
<tr>
<th>Test</th>
<th>Original Open64</th>
<th>Methodology in Open64</th>
<th>Percentage improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total text size</td>
<td>Kilo cycles</td>
<td>Total text size</td>
</tr>
<tr>
<td>7</td>
<td>1884</td>
<td>3.89</td>
<td>1168</td>
</tr>
<tr>
<td>8</td>
<td>2448</td>
<td>34.1</td>
<td>1896</td>
</tr>
<tr>
<td>9</td>
<td>3744</td>
<td>1161</td>
<td>3672</td>
</tr>
<tr>
<td>10</td>
<td>11860</td>
<td>17358</td>
<td>8372</td>
</tr>
<tr>
<td>11</td>
<td>7392</td>
<td>20996</td>
<td>4580</td>
</tr>
</tbody>
</table>

- **Code size always improves**
- **Slight degradation in performance: can be blindly used for non-time critical portions of code**
Other code size improvement efforts

- Register promotion of small structures (and members in big structures) to reduce unwanted loads/stores
- Better heuristics for unrolling (which estimate cycle benefit for unroll factor, and prevent unnecessary unrolling)
- Use –Os –ipa for code size optimization
- Better clustering of VHO switch lowering
- Generalized tail merging
- Recognizing loops out of straight line code
Acknowledgements

- Thanks to other members of the Qualcomm DSP compiler team for their valuable feedback on the work
  - Anshuman Dasgupta
  - Raja Venkateswaran
  - Sundeep Kushwaha
  - Sergei Larin

- Special thanks to Taylor Simpson, Director of the Qualcomm DSP system software team, for supporting this effort