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 Incorporated
5775 Morehouse Drive
San Diego, CA 92121-1714
U.S.A.

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

Simplified example

```
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);
}
```

Assembly code by original Open64

```
test:
    r8=#(.rodata)
    r6=r0
    if (r0 >= #19)
     jump .Lt_0_2
    r8=add(r8,r0<<#2)
    r9=loadw(r8+#0)
    jumpr 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)
      jumpr 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
```

```
.Lt_0_2:
  r0=r6
  return;

.Lt_0_14:
  r0=r1
  r1=#1
  callr r9
  r6=r0
  jump .Lt_0_2

.Lt_0_8:
  r6=#(g_array)
  r6=r6+r9
  r6=loadw(r6+#0)
  jump.Lt_0_2

.Lt_0_1:
  r6=add(r1,r9)
  jump .Lt_0_2
```

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

```
.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 ENCODING:

Memory offset	Logical-or operand	Extract bit-width

31st bit — 15th bit — 7th bit — 0 bit

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

Item: Constant Operands
- Types: constants loaded in r1, r3, r4, r5

Item: Memory Offset
- Types: memory offsets for loadub and loadw

Item: function with same signature
- Types: the function called

EXAMPLE ENCODING REQUIRES AT LEAST THREE WORDS (12 bytes):

Constant for "r5"	Constant for "r4"	Constant for "r3"	Constant for "r1"

31st bit | 23rd bit | 15th bit | 7th bit | 0 bit

unused	offset for "loadw to r2"	offset for "loadub to r0"

31st bit | 24th bit | 9th bit | 0 bit

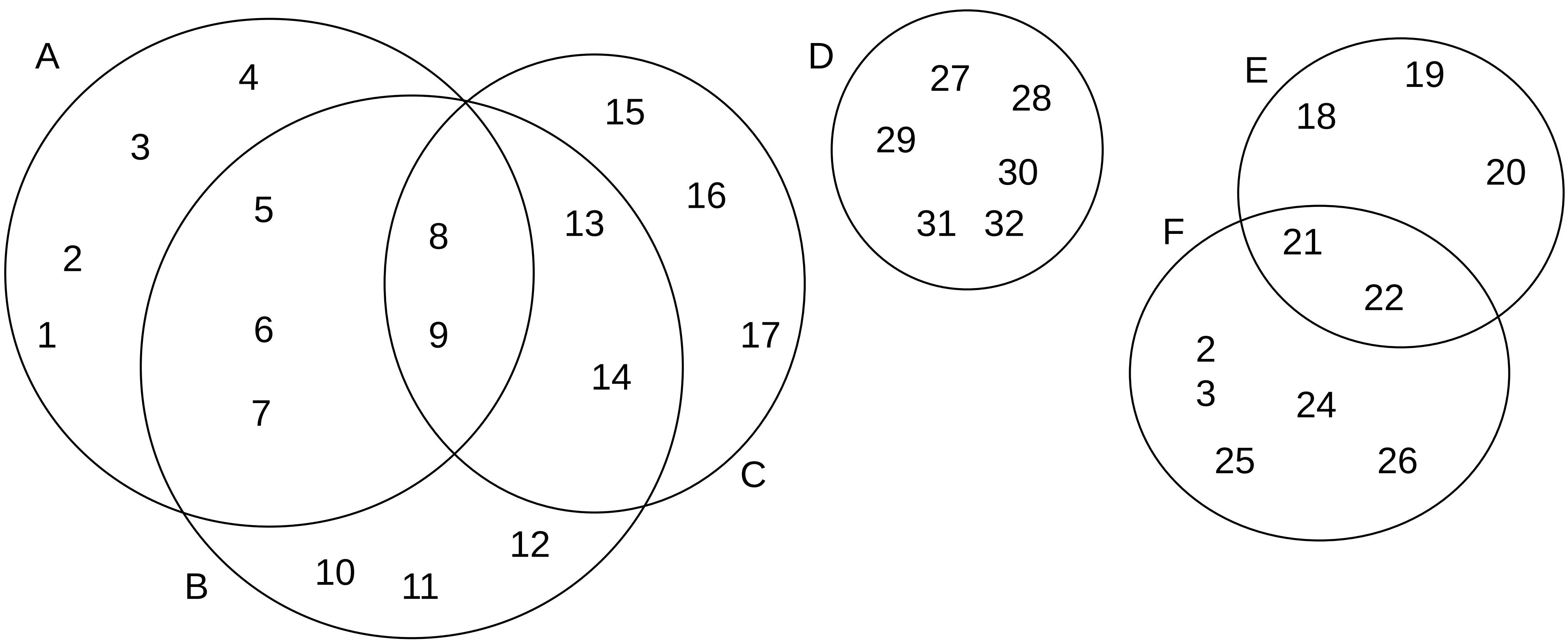
Callee Function Address

31st bit | 0 bit

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

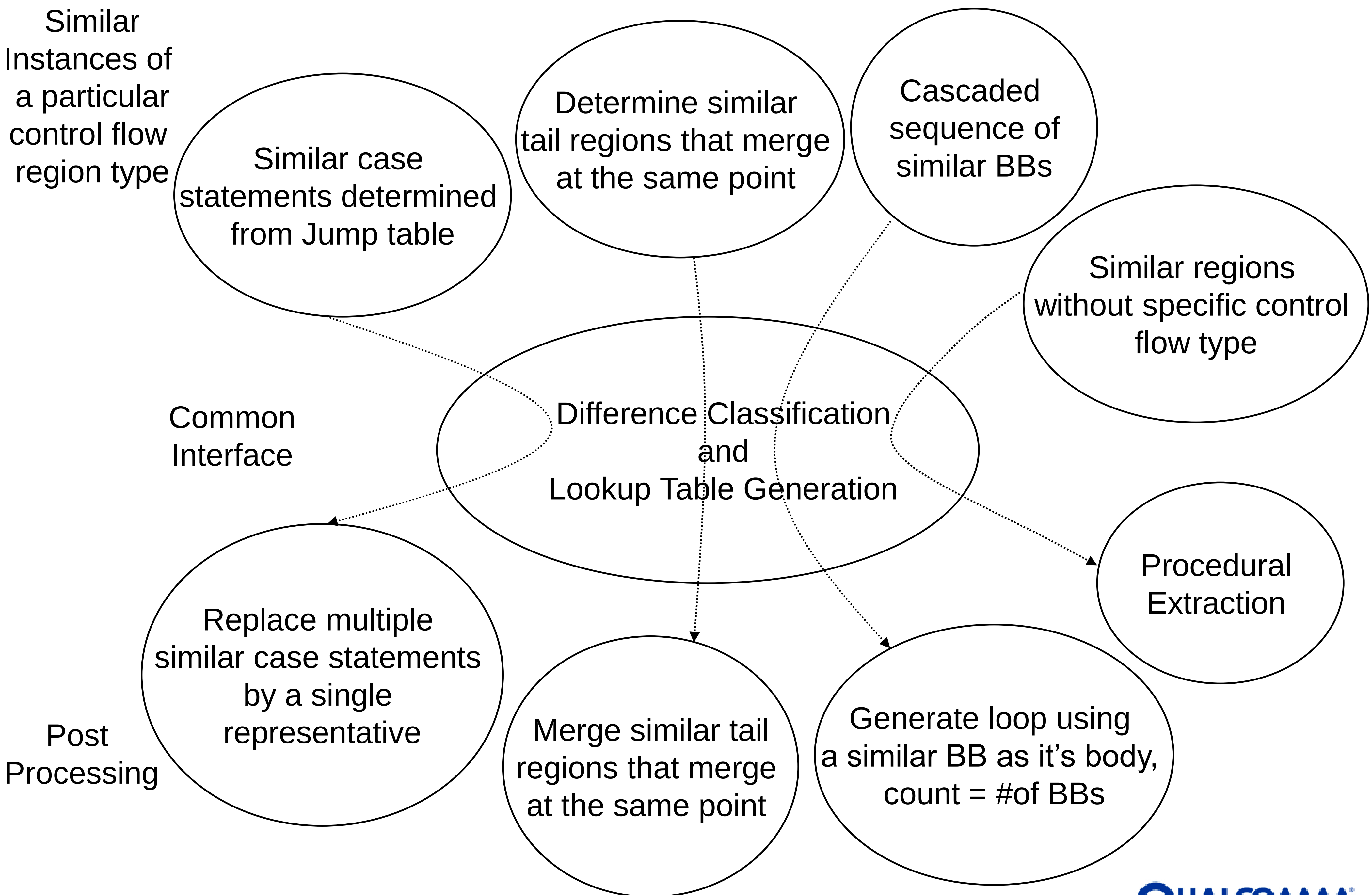
Venn diagram representation of difference classification and LUT generation



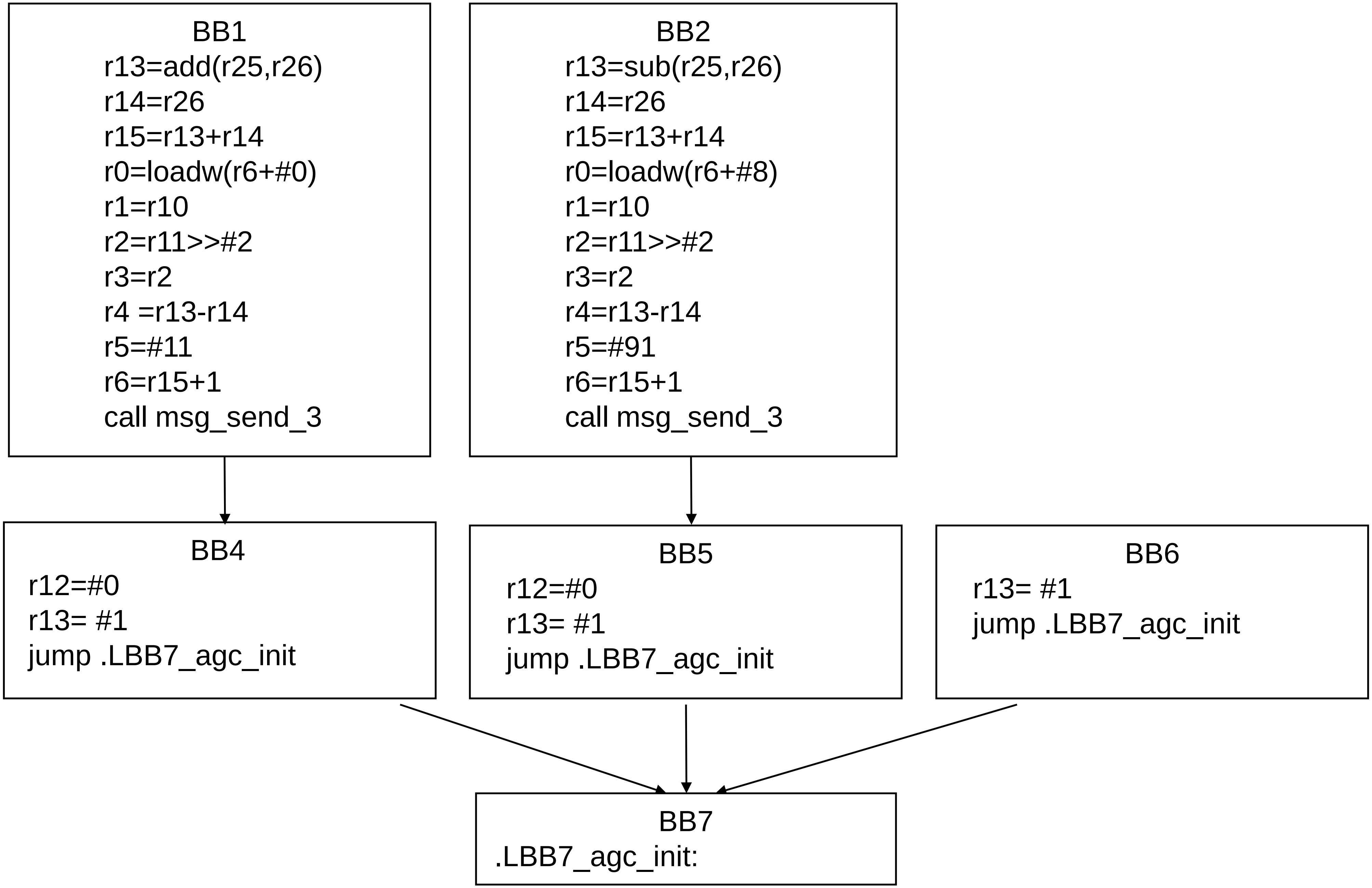
- {1,2,3,…..30,31,32} indicate code regions that are compared
- A, B, C, D, E, F indicate "difference type"
- Each distinct region indicates "difference class"

{1,2,3,4}, {5,6,7}, {8,9},….,{27,28,29,30,31,32}

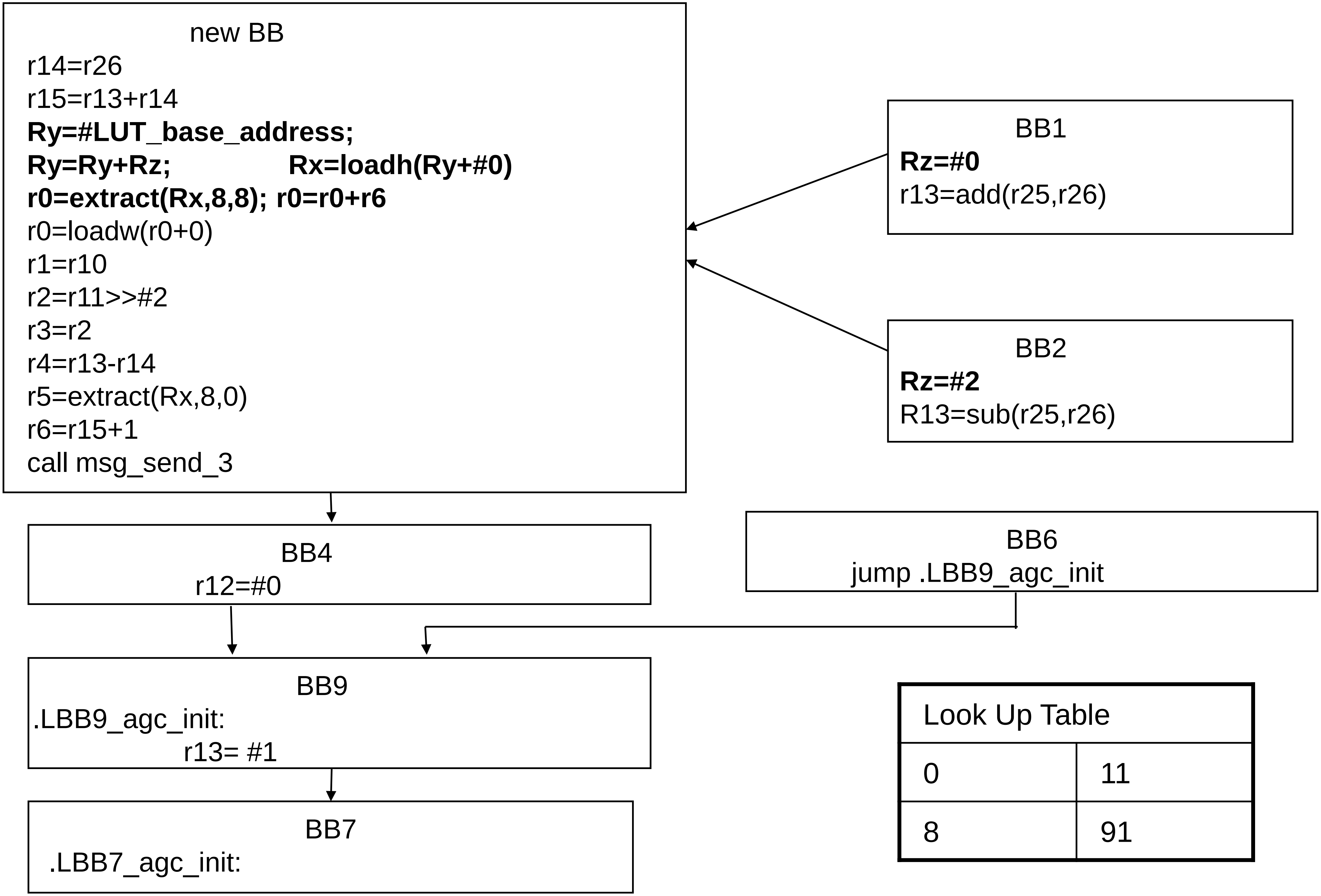
Overall framework



Tail merging example



Tail merging with differences in LUT



Look Up Table	
0	11
8	91

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: **Ry = #LUT_base_address**
LoopCounter=#5

```
New BB 7
LoopStart:
        Rx=loadw(Ry++)
        Rz=extract(Rx,16,16)
        Rz=r6+Rz;
        r0=loadw(Rz);
        r1=r10
        r2=extract(Rx,16,0)
        r3=r11
        call msg_send_3
LoopEnd
```

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

0	106
48	135
36	224
64	298
72	234

Procedural abstraction with differences in LUT

```
r4=add(r5,#10)
r4=mul(r4,r3)
r4=r4<<2;
r4=and(r4, r2)
r4=loadw(r4)
r4=or(r4,#12)

<OTHER
INSTRUCTIONS
EXACTLY SAME>
```

```
r0=#10
r1=#12

r4=add(r5,r0)
r4=mul(r4,r3)
r4=r4<<2;
r4=and(r4, r2)
r4=loadw(r4)
r4=or(r4,r1)

<OTHER
INSTRUCTIONS
EXACTLY SAME>
```

```
r0=#0
call NewProcedure
```

```
r4=add(r5,#14)
r4=mul(r4,r3)
r4=r4<<2;
r4=and(r4, r2)
r4=loadw(r4)
r4=or(r4,#51)

<OTHER
INSTRUCTIONS
EXACTLY SAME>
```

```
r0=#14
r1=#51

r4=add(r5,r0)
r4=mul(r4,r3)
r4=r4<<2;
r4=and(r4, r2)
r4=loadw(r4)
r4=or(r4,r1)

<OTHER
INSTRUCTIONS
EXACTLY SAME>
```

```
r0=#4
call NewProcedure
```

```
NewProcedure:

Rx=
  #LUT_base_address

Rx=Rx+r0
r0=loadw(Rx+#0)
r0=extract(r0 ,#8,#0)
r1=extract(r0 ,#8,#8)

r4=add(r5,r0)
r4=mul(r4,r3)
r4=r4<<2;
r4=and(r4, r2)
r4=loadw(r4)
r4=or(r4,r1)

<OTHER
INSTRUCTIONS
EXACTLY SAME>

RETURN
```

Look Up Table	
12	10
51	14

Code size comparison: Open64 and GCC (-Os)

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

Test cases	GCC 4.3.2 (size in bytes)			GCC 3.4.6 (size in bytes)			Original Open64 (size in bytes)			Methodology in Open64 (size in bytes)		
	Pure text	rodata	**Total text**	Pure text	rodata	**Total text**	Pure text	rodata	**Total text**	Pure text	rodata	**Total text**
1	3368	6164	**9532**	3364	6164	**9528**	4040	7512	**11552**	1676	6568	**8244**
2	1152	308	**1460**	1224	312	**1536**	1352	352	**1704**	916	476	**1392**
3	1032	352	**1384**	1036	352	**1388**	1336	352	**1688**	572	416	**988**
4	896	0	**896**	876	0	**876**	1352	0	**1352**	116	224	**340**
5	2204	9204	**11408**	2204	9204	**11408**	2220	9208	**11428**	524	9208	**9732**
6	1052	1128	**2180**	1088	1128	**2216**	728	1952	**2680**	724	1392	**2116**

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

Code size and performance impact: Open64 (-Os)

Test	Original Open64		Methodology in Open64		Percentage improvement	
	Total text size	Kilo cycles	Total text size	Kilo cycles	**Total text size**	**Cycle Performance impact**
7	1884	3.89	1168	2.76	**+38**	**+27**
8	2448	34.1	1896	39.0	**+22**	**-14.3**
9	3744	1161	3672	1162	**+2**	**-0.09**
10	11860	17358	8372	17360	**+29**	**-0.05**
11	7392	20996	4580	21329	**+38**	**-1.5**

- *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