Code Size reduction by difference classification and customized look-up table generation

Subrato K. De, Kun Zhang, Tony Linthicum

Qualcomm Incorporated, San Diego & Austin, USA.

{sde, zhangk, tlinth }@qualcomm.com

ABSTRACT

Code size has become an important constraint for applications on mobile devices. Not only should the applications be very responsive and execute fast, they should also consume low power, and be reasonably compact in size to fit in the limited memory of the mobile devices. In this paper we focus on a specific opportunity of code size optimization that is detected in software for mobile devices like wireless networking protocols, modems, etc. These applications have large amount of code-regions across the control-flow graph that are inherently similar, but with slight differences that are compile time constants. Examples of such similar code regions are: case statements labeled by "jump tables", tail regions that merge to a common point, cascaded sequence of similar code-regions with unique predecessor-successor relationship, etc. This paper presents a novel approach where similar code-regions across the control flow graph are classified into disjoint sets based on their differences, and then replaced by a single representative code-region with the differences encoded in customized look-up tables resulting in reduced code size. The methodology shows remarkable reduction in code size for large switch-case constructs that have very similar case statements. Implementation is done in the Open64 compiler, even though it could be integrated with other compilers (e.g., GCC). We also provide some results of code size reduction, which are in the range of 30% to 80% for pure text (instructions only), and 5% to 60% for total text (instructions and read only data), achieved by the algorithm on some of the functions in software applications for mobile devices.

1. Introduction

Open64 is an open source C/C++/Fortran77/90 compiler that is currently used in various industry and academic research projects. It originates from the SGI Pro64(TM) compiler suite that was released under the GNU General Public License. Open64 was originally intended to be a compiler generating high performance code that is extremely focused on execution speed of the generated code. Optimization for code size of the generated assembly, to the extent demanded by application developers for mobile devices, was not the focus. However, the use of Open64 in Qualcomm as a compiler for embedded/DSP processors used in mobile devices created the urgent need for enhancing the code size optimization infrastructure within Open64. We will briefly summarize the different features of Open64 and then discuss some generic improvements for code size optimization. Then we illustrate the details of code size reduction by difference classification and customized look-up table generation, which is the main theme of this paper.

1.1 Open64 as a compiler for generating high performance code

Open64 uses an intermediate representation (IR) called WHIRL that has multiple levels of representation and serves as the common interface for the compiler phases. The optimizations are mainly geared towards cycle performance, as evidenced by the exhaustive set of loop optimizations and transformations, vectorization, aggressive unrolling and inlining, function cloning, hyperblock scheduling, and predication. The important phases of Open64 are:

- The very high level optimizer (VHO) lowers aggregates, flattens nested calls, etc.
- The inter-procedural analysis (IPA) first gathers data flow analysis information from each procedure locally. It then generates the call graph, performs inter-procedural analysis and transformations. It performs global variable optimization, dead function elimination, inter-procedural alias analysis, function cloning, constant propagation, function inlining, etc.
- The loop nest optimization (LNO) phase calculates dependence graph for array accesses and performs loop transformations, and automatic vectorization.
- The global optimizer (WOPT) computes the control flow graph, the dominator tree, dominance frontier, control dependence set, and then converts the IR to a hashed SSA form. It performs def-use analysis, alias classification, pointer analysis, induction variable recognition/elimination, copy propagation, dead code elimination, partial redundancy elimination, register variable identification, bitwise dead-code elimination.
- The code generator (CG) performs target specific optimizations, instruction selection, scheduling,

software pipelining, hyper-block scheduling, register allocation and emits the assembly code.

The details of these phases can be found in [7, 8, and 9].

1.2 General modifications to Open64 for generating size aware assembly code

Our goal is to enhance Open64 to generate assembly code that gives reasonable cycle performance and is also size optimized. One of our efforts is to leverage interprocedural analysis (IPA) using -Os for code size optimization. The function inlining heuristic is made conservative when compiling with –Os with IPA. The loop unroller heuristic is enhanced to estimate the benefit of unrolling the loop by a certain factor. Unrolling is not performed if the estimated payoff in cycle gain is below a certain limit. The clustering algorithm for switch lowering in VHO is tuned to generate denser clusters, leading to more clusters that could be differently lowered i.e., by jump-table, by linear if-else, by binary search if-else, based on a heuristic. Changes are been done in VHO to facilitate aggregate copies, register promotion of structures and unions, which reduce both code size and stack usage [10]. Finally, a novel mechanism for code size optimization, by creating a customized look-up table (LUT) for the differences found in similar code regions across the control flow graph (CFG,) is incorporated in the Open64 compiler. This is the main focus of this paper and is discussed in details in sections 2 and 3.

1.3 Organization of this paper

This paper is organized as follows. Section 2 illustrates the difference classification, the encoding/decoding mechanism, and the generation of the LUT for differences in similar code regions found across the CFG. Section 3 compares some of the existing approaches and describes the generalized infrastructure that uses the difference classification results and the encoded LUTs, and replaces the similar code-regions by a single representative. Section 4 compares the improvement in code size and the performance impact over some of our internal applications when using the novel methodology. Finally, we present our conclusions in section 5, followed by acknowledgements.

2. Difference classification and customized look-up table generation

This section discusses core concept of the novel methodology where similar code-regions spread across the CFG are grouped into disjoint sets based on their differences. Each set is then is replaced by a single representative code-region with the differences encoded in a compact fashion in LUTs.

2.1 Overview of the methodology through a motivating example

Figure 1(a) shows a motivating C-code where the impact of code size reduction is phenomenal using the methodology described in this paper. It shows a switch-case statement that is commonly found in wireless networking protocol and modem software [11]. In the example in figure 1(a), the case statements can be grouped into three distinct clusters, such that the case statements in a cluster are very similar to each other except for some differences that are compile time constants. It is to be noted that some of these redundancies can be even removed by a suitable strategies while developing the C-code. However, the importance of this work is because of the fact that given a not-so well organized C/C++ source code, a compiler could generate code efficient enough in terms of size and cycles needed for software on mobile devices [12]. Also, there exists vast amount of C/C++ source code that isn't originally written for embedded systems keeping in mind the stringent memory requirements, but need to be quickly incorporated into the product. This makes a strong case for the need of the work described in this paper.

```
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);
}
```

Figure 1(a): C-code for the motivating example.

The fundamental idea is to replace a set of similar code regions (in this example the case statements), by a representative code region, and encode the difference into a LUT. While, the idea of similar code identification has been used in various researches [4, 13], the novelty of the work in this paper comes from the following:

- Classification based on the types of differences so that they can be suitably encoded and decoded.
- Increased scope of the similar regions by a compact encoding of as many difference items as possible.
- Use of heuristics to trade-off between code size increases due to the overhead (addition read-only data in LUT, additional decoding instructions) v.s. code

size reduction due to removal of multiple instances of similar code regions by a single representative.

- Lightweight mechanism to detect similar code-regions by using the CFG as the guiding factor.

Figure 1(b) shows the code generated by Open64 compiler using 68 instructions, as well as GNU C/C++ compiler versions 3.4.6, and 4.3.2, without using the methodology described in this paper. All three compilers generated a jump table whose entries are the labels for the case statements.

```
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
```

Figure 1(b): Assembly code for the motivating example without the optimization discussed in this paper

Figure 2 shows the assembly code using 22 instructions when the methodology described in this paper is used. The jump table entries are replaced with the difference items that are compile time constants and the similar case statements that are replaced by a single representative. The jump table now partly behaves like a look-up table. However, some additional code is needed for decoding the LUT entries and for the conditional jumps that leads the program control to execute the particular representative case statement. A reduction of 68 -22 = 46 is achieved, while the read only data (rodata) size for the jump table remains unchanged.

2.2 Profitability of doing the transformation

The example in the previous section uses a jump table to implement the switch-case statements. Entries of the LUT are formed by replacing the corresponding entry of the jump table. However, there is some additional control code needed to make the execution reach the specific case statement that is a representative for the set of similar case statements.

```
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
```

ASSEMBLY CODE AND LUT
WITH THE NOVEL METHODOLOGY

Figure 2: Assembly code generated for the motivating example using the optimization discussed in this paper. It shows much reduced code size.

There is also some additional code in the representative case statement for decoding and extracting the desired value from the LUT entry. These give rise to the overhead components of doing this optimization, both for code size and cycles. If the code size of the overhead becomes more than the memory saving obtained by replacing multiple case statements by a single representative, the optimizing won't be beneficial. Hence we use a profitability function to decide for or against invoking the transformation for code size benefit:

- CODE SIZE REDUCED = {(SIZE OF A SIMILAR CODE REGION) * (# OF SIMILAR REGIONS DETECTED -1) - (SIZE OF CODE TO ACCESS LUT + SIZE OF DECODE CODE + SIZE OF GLUE CONTROL CODE + ADDITIONAL LUT SIZE)}
- % CODE SIZE REDUCED = "CODE SIZE REDUCED" * 100 / {(SIZE OF A SIMILAR CODE REGION) * (# OF SIMILAR REGIONS DETECTED)}

The optimization is useful if "CODE SIZE REDUCED" > 0. However, cycle performance is reduced slightly when compared to the performance of a pure jump-table based implementation, particularly if there is some discontinuity in the case numbers and there are multiple groups of similar case statements. In the code size optimized example in Figure 2, there is some additional control flow overhead to determine the particular representative code region (out of the three groups) to jump to, based on the switch index. However, this method can be applied for code size

optimization of non time-critical portions of the application by using the profitability function discussed in this section.

2.3 Difference types and difference classes

The types of differences that can be considered must be a compile time constant so that it can be encoded in a LUT. As an example, we can consider differences in two constant operands (say 123 v.s. 99), but cannot easily consider two different operations (say addition v.s. subtraction). If we really want to incorporate such difference, it will add control code overhead to decide on between "addition" v.s. "subtraction" at runtime based on a particular value (say "1" for addition, and "2" for subtraction) encoded in the LUT entry. Certainly, capturing such difference might be profitable under certain conditions, but is considered for future enhancement.

Based on our analysis of certain mobile software code, the current scope of this paper considered the following frequently occurring difference items: (i) constant operands, (ii) memory offset in the "base+offset" addressing mode for loads and stores, (iii) constants used in address computation, and (iv) functions with the same signature. When comparing two or more similar code regions, different instances of one or more of the above difference items are actually associated with distinct operations in the code region making them uniquely identifiable, and are called difference types. The list of all the difference types among two or more comparable code regions forms a "difference class". When all instructions of the comparable code regions, except those in the difference class, are exactly same the code regions can be replaced by a common representative with the difference class encoded in a LUT. Thus in Figure 3, the difference items are "constant operand" and "memory offset". There three difference types – constant operand in logical-or operation, constant operand for the width of extract operation, and memory offset in the loadw operation. These three difference types form the difference class that uniquely clusters the three code regions (basic blocks in this case) with labels .Lt_10_13, .Lt_10_12, and .Lt_10_11 into a single group.

2.4 Encoding and decoding the differences

Encoding and decoding are needed when a difference class has multiple differences types. Compact encoding of the differences types are needed for efficient memory usage. A greedy approach for compaction is currently used, where the maximum bit requirement to represent the particular difference, among all the code-regions being compared, is determined. The difference types are then arranged in ascending order of "maximum bit requirements" per type. Figure 3 shows three code-regions (basic blocks with labels .Lt_10_13, .Lt_10_12, and .Lt_10_11) that are similar with three difference types: constant operand for the logical-or operation (the 3 values in order are: 16, 8, and 4), constant representing the bit-width of the extract operation (the 3 values in order are: 10, 12 and 14), and the memory offset for the "loadw" operation (the 3 values in order are: 600, 680 and 720). The maximum bits required to represent the three difference types are:

- constant operand for the logical-or operation = BinaryBits(MAX[16, 8,4]) + 1 = 6 bits,
- constant for the bit-width of the extract operation = BinaryBits (MAX[10, 12,14]) + 1 = 5 bits,
- memory offset for the "loadw" operation = BinaryBits (MAX[600, 680,720]) + 1 = 11 bits.

Where, BinaryBits() is a function that computes the number of bits in the binary representation of the magnitude of the number, e.g., 16 decimal = 10000 binary (needs 5-bits), 14 decimal = 1110 binary (needs 4-bits). Thus the difference class for the code regions can be implemented using 6 + 5 + 11 = 22 bits. Since there is already 32 bits available (1 word) from the existing jump table, we have a relaxed constraint. Hence, the encoding done is: 0^{th} to 7^{th} bits for representing "bit-width" for the extract operation, 8^{th} to 15^{th} bit for representing the constant operand for the logical-or operation, 16^{th} to 31^{st} bit for representing the memory offset of the "loadw" operation.

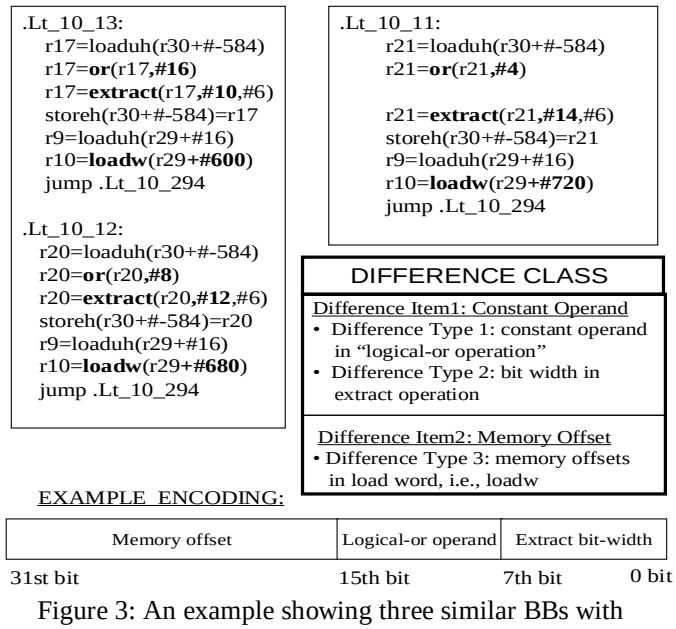


Figure 3: An example showing three similar BBs with multiple classes of difference that can be successfully encoded into the look-up table without increasing its size.

Figure 4 shows an example where the number of difference types is large enough to require additional LUT. In this example the code-regions being compared are three basic blocks with labels .Lt_30_1, .Lt_30_3, and .Lt_30_7. There are four differences types of difference item "constant

operand", each of which would require at least 8 bits. There are two difference types of difference item "memory offset", requiring 10 and 15 bits respectively. Finally the last difference type is of difference item "function with same signature", requiring 32 bits for 4-byte addresses. This example needs 12 bytes to encode the difference class, thus requiring extra 8 bytes per existing jump-table entry.

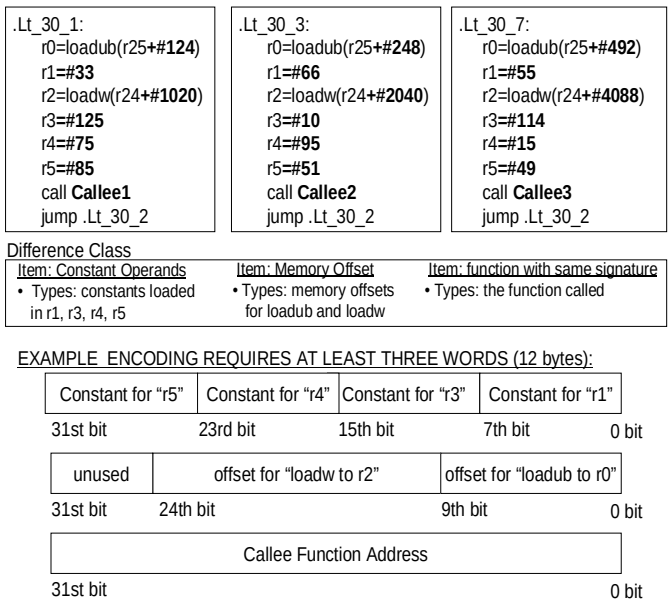


Figure 4: An example showing multiple similar BBs with multiple classes of difference that can only be encoded into the look-up table by increasing its size, or adding a new look-up table.

Decoding involves extracting the value for the particular difference type from the entries of the LUT. However, unlike encoding, where the encoded values are compile time constants, decoding involves introduction of assembly instructions that extracts a consecutive sequence of bits starting from a certain offset, at runtime. As an example, to extract the value for the logical-or operand in figure 3, a signed extract operation with the semantics "Rx=signed_extract(Ry=LUT_entry, width=8, offset=8)" is used. This operation extracts a contiguous set of 8 bits, at an offset of 8 bits from the least significant bit of the register Ry containing the LUT entry. The extracted value is signed extended and saved in register Rx. In most processors signed extract is implemented using an arithmetic shift-left by X, followed by arithmetic shift-right by Y, where, X and Y are:

- X = Right shift value = "no. of bits in register – (width+offset)", and
- Y = Left shift value = "no. of bits in register – width"

2.5 Description of the algorithm

The input to the algorithm is a pair of code-regions that are to be compared for similarity. For a particular pair of code-regions (example for two different code regions given by two different case statements), similarity is determined hierarchically, i.e., first the control flow pattern is compared, then the individual basic blocks, finally the operations in the basic-blocks and their operands. The differences that could be encoded in the LUT are at the operand level. Thus two regions are similar if they have the same control flow pattern and their basic blocks have the same operations with the same dependency graph, but the individual operations could have different constant values as their operands, as mentioned in section 2.3.

A pair of code-regions is rejected if there is any difference item that can't be handled by the scope of this paper. The list of pair-wise code regions, with each pair having its list of difference types, is then used to form disjoint sets of code-regions of a particular "difference class". As described in section 2.3, a "difference class" uniquely clusters a group of similar code regions. Each difference class contains a group of code-regions, whose difference items can be exactly encoded in the same way in the LUT and decoded with exactly the same set of operations. This makes the set of code regions replaceable by a single representative code region, with the difference class encoded in a LUT and difference types read back using the decoding instructions that are added in the representative code region. The original switch index becomes the LUT index, similar to the indexing mechanism of a jump table. The list of pair-wise code regions are first arranged in descending order of number of difference types found in the pair. Thus if a particular code region Z occurs multiple times in the list with other code regions W, X, Y, with each pair (Z,W), (Z,X), and (Z, Y) having different number of difference types, the pair with the higher number of difference types gets priority. Example, if the difference types in (Z, W) > difference types in (Z,X) > difference types in (Z Y) , the pair (Z, W) is first considered as a difference class. The pairs (Z,X) and (Z,Y) can also be included if the difference types present in them are a subset of the difference types found in (Z,W). However, if any of the difference types in (Z,X) and/or (Z,Y) is not a subset then X and/or Y cannot be included in the difference class that includes Z and W.

The disjoint sets of code-regions of specific difference classes can be pictorially represented as a Venn diagram, as shown in figure 5, which represents different code regions as numbers. The following disjoint sets representing different difference classes of code region are present: {1,2,3,4}, {5,6,7}, {8,9}, {13, 14}, {15,16,17}, {18,19,20}, {21,22}, {23,24,25,26}, {27,28,29,30,31,32}. Each set A, B, C, D, E, F represents a difference type (as discussed in section 2.3). Example: the set {1, 2, 3, 4} represents code regions whose difference class is composed of only a single difference type A, the set {5, 6, 7,} represents code regions whose difference class is composed of a combination of two difference types A

and B, while the set {8, 9} represents code regions whose difference class is composed of a combination of three difference types A, B, and C. In each of the disjoint region, the set of code regions can be replaced by a representative code-region and a LUT with number of entries equal to the number of similar code-regions in that set. Some additional control code is needed to allow control to jump to the representative code region, when the corresponding case statements are invoked.

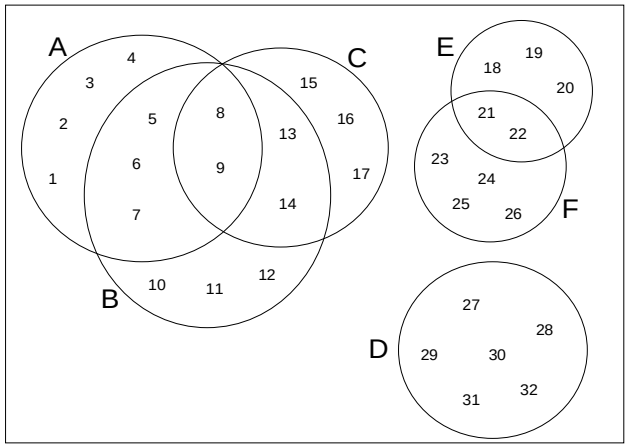


Figure 5: The disjoint sets of code-regions pictorially represented as a Venn diagram

3. Code size optimization across control-flow regions using LUT for differences

In this section we first briefly compare the different approaches of code size reduction for similar code regions that has been tried out in research. Many previous approaches of similarity extraction and replacement [1, 2, 5, and 6] for reducing the code size operated after register allocation and scheduling, where the code representation is very close to the final assembly. These approaches suffered from some serious problems. First, these approaches needed to extract the control flow information after the similar sequences are detected, and then decide if the two similar sequences are indeed equivalent w.r.t program control flow. As an example two otherwise similar sequences, one of which has an incoming control flow edge in between and the other doesn't, cannot be replaced by a single representative code region since it will break the control flow. The implementation in this paper is in the control flow optimization (CFLOW) sub-phase in the code generation (CG) phase of Open64 and uses the CFG. Second, many of the previous techniques are limited to the textual representation of a program, where the algorithms are sensitive to the use of different registers. Some works used register renaming to solve the problem. However, due to the limited number of registers, not all registers could be renamed with spare registers and some additional register to register move instructions have to be inserted. Thus, register renaming could not solve the problem completely. Third, working with a special order of instructions hinders the identification of similar basic blocks, or code regions. The CFLOW sub-phase in the CG-phase of Open64 occurs before register allocation and instruction scheduling. Hence the methodology described in this paper could avoid the register reuse and renaming problem, and also avoided the difficulty in determining similarity due to the reordering of the instruction sequence that comes with scheduling. However, the original source code ordering could still hamper similarity extraction. Hence the methodology in this paper is augmented to incorporate the associative, commutative and distributive nature of the operations. Also, use-def chains are created and used in conjunction to the dependency graph to determine the similarity of two code regions to expand the scope to cases where the original source code ordering can be different for the same functionality.

Figure 6 illustrates the overall infrastructure that uses the difference classification results and the encoded LUTs, for replacing code-regions by a single representative code-region across different types of control flow regions. First, the different control flow regions are identified according to the control flow features. Second, the common interface classifies the difference among the basic blocks in one region and generates the necessary LUTs, as discussed in Section 2. Finally, the control flow regions are processed, and transformed according to the needs of the specific control flow region type. Figure 6 illustrates three specific control flow region types to perform searching for similar among code regions: (i) case statements regions indicated by labels in a jump table, (ii) tail regions that merge at a common point, (iii) basic block (or code regions with limited control flow) that form a cascaded chain. The module for performing difference classification and LUT generation have generic APIs that enable plugging of newer control flow region types easy. However, there is a default procedural extraction phase that also uses the module for difference classification and LUT generation, when no specific control flow region type is determined. The transformation for the case-statement regions with jump tables have already been discussed in section 2. The following paragraphs briefly describe the application of the methodology to control flow regions of different types.

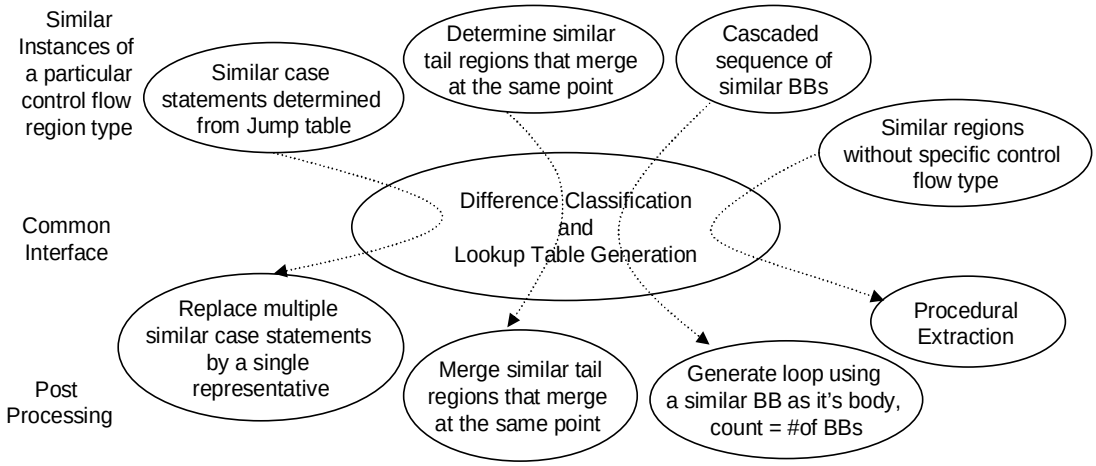


Figure 6: Overall infrastructure of code size optimization using difference classification and LUT generation.

3.1 Tail merging with LUT generation for differences

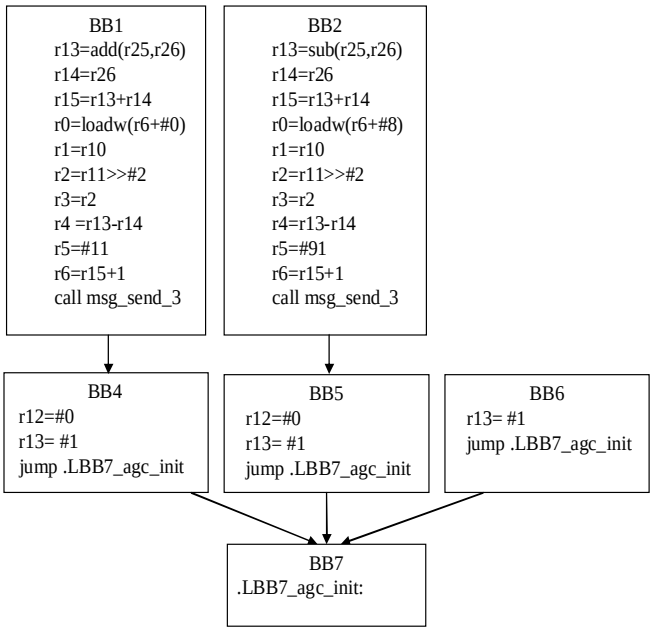


Figure 7(a) CFG before code size optimization across tail regions that merge at a common point.

Figure 7 (a) gives an example of tail regions that could potentially be merged. In this example, the basic blocks BB1 and BB2 are similar; BB4, BB5 are exactly same, while BB6 has one less operation than in BB4/BB5. The algorithm starts from the basic blocks (i.e. BB4, BB5, and BB6) with the same successor (i.e. BB7), and first determines the number of instructions in them. If two basic blocks are not completely the same (i.e. BB4 and BB6); then one basic block (i.e. BB4) is split. Next, the algorithm merges the similar basic blocks (i.e. BB4, BB5, and BB6). The algorithm optimizes the control flow graph by optimizing the branches and removing the unreachable basic blocks (i.e. BB5), making BB4 as the successor of both BB1 and BB2.

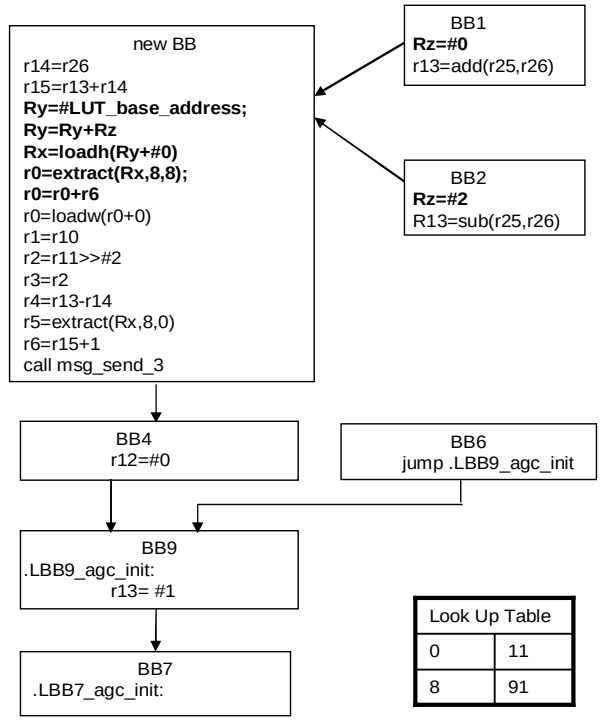


Figure 7(b) CFG after code size optimization across tail regions that merge at a common point.

The algorithm is further applied to BB1 and BB2. The algorithm detects instructions that have differences conforming to the acceptable difference categories, as discussed in section 2. The instructions (i.e. r0=loadw(r6+#0) in BB1 compares with r0=loadw(r6+#8) in BB2 with the memory offset difference. Similarly, the instruction r5=#11 in BB1, compares with instruction r5=#91 in BB2 with the difference in the constant operand. Representing these two difference sets in the LUT thus increases the scope of the code-region that could be tail-merged.

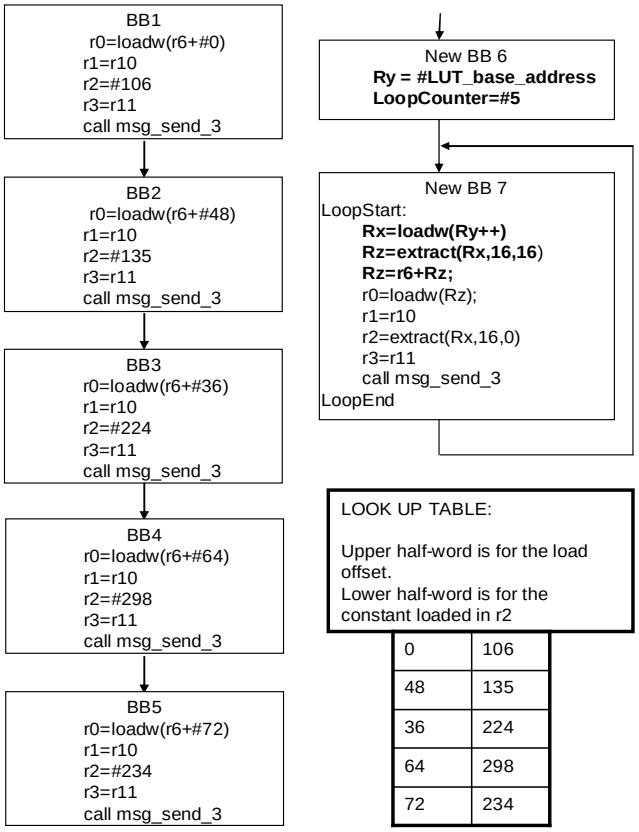
The final control flow graph after tail-merging the code-regions is shown in Figure 7 (b). The associated LUT is also shown. The operation Rz=#0 and Rz=#2 sets the LUT offset to be used for accessing the LUT entry for the two original basic blocks, BB1 and BB2 respectively. In this example each LUT entry is 2 bytes, using a byte each for the two difference categories.

3.2 Loop conversion for a sequence of cascaded code regions with LUT generation for the differences

Figure 8(a) shows a sequence of code regions (basic blocks BB1, BB2, BB3, BB4, and BB5) forming a cascaded chain. The sequences of similar code–regions have a unique predecessor and successor relationship and could be replaced by a loop. Figure 8(b) shows the transformed loop body. The loop body is a single representative code-region (shown as New BB 7), and the loop header (shows as New BB 6). A set of additional decoding operations on the LUT entry, which is indexed by the loop induction variable, is present in the loop body (i.e., New BB 7).

3.3 Procedural abstraction with LUT generation for the differences

Procedural abstraction can be augmented with difference classification and LUT generation, when no specific control flow region type can be determined. The methodology in this paper can improve the traditional procedural abstraction technique. In the traditional procedural abstraction technique, the different instructions between the two basic blocks are left in the original code region. While the approach in this paper can include the operations involved with the differences in the abstracted procedure by incorporating the differences in the LUT. The abstracted procedure has some additional instructions to load the particular LUT entry and operations to decode and generate the difference values. Figure 9 shows an example of procedure abstraction with LUT generation. Two similar code regions that are replaced by two calls to the abstracted procedure, named “NewProcedure”, are shown in Figure 9. The call site initializes the LUT index and passes it as an argument to the abstracted procedure (as shown by register r0 before the call to “NewProcedure” in Figure 9).



(a) CFG before code size optimization

(b) CFG after code size optimization

Figure 8: Code size optimization for a cascaded chain of similar BBs

4. Results

This section provides some of the results of using the new code size optimization method discussed in this paper. Implementation is done in Open64, but other compilers (e.g., GCC 3.4.6 and GCC 4.3.2) would also benefit by implementing this optimization. This is a work in progress. The current implementation focuses on the similar case statements in switch-case constructs. Additional LUTs are not added; hence the total size of the encoded value is limited to the address size in the target processor. Table 1 shows the comparison in text size (pure text and read only sections) with original Open64 (i.e., without the optimization), GCC 3.4.6 and GCC 4.3.2, all compiled using –Os optimization level.

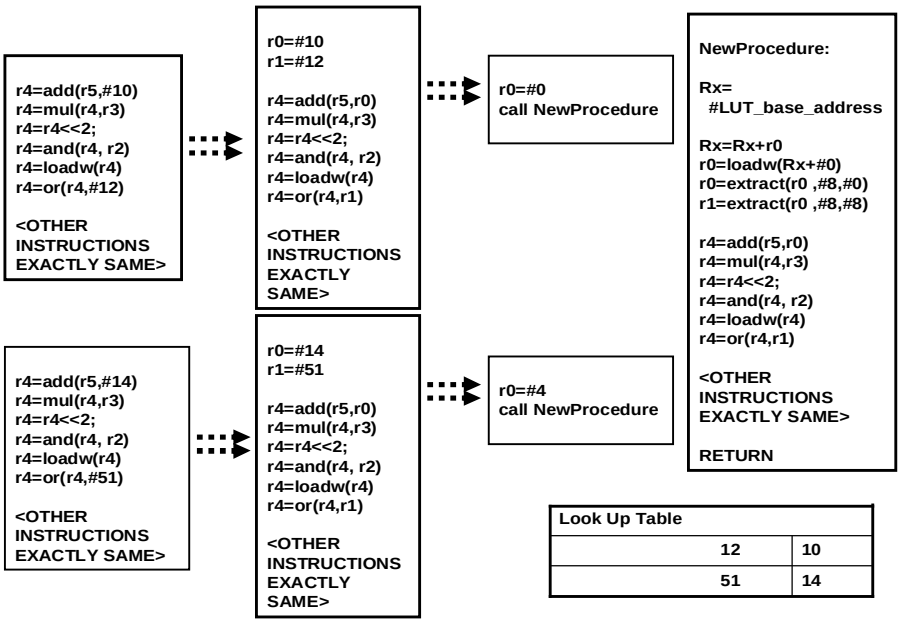


Figure 9: Procedural abstraction with difference encoding.

The test cases in table 1 are obtained from Qualcomm's software applications on mobile devices and contain one or more functions. It is seen that the methodology in this paper consistently reduces code size: for pure text (assembly instructions) in the range of 30% to 80%, while the total text (pure text and read only data) in the range of 5% to 60%. For an entire application the total code size reduction would depend on how frequently functions with largely similar case statements are present in them.

Table 2 illustrates the total text size improvement and the impact on cycle performance of the new methodology on Open64. It compares the original Open64 and the modified Open64 (i.e., with the novel methodology in it). It is seen that code size consistently reduces (as large as 38%), with very small performance reduction in most cases. The slight performance reduction (for tests 9, 10, and 11) is attributed to the overhead for decoding the LUT and the control flow overhead to select the particular representative code region, when compared to a pure jump table based implementation. In some cases, like test example 8, cycles can degrade noticeably when there are multiple difference classes in the switch-case construct, or frequent gaps in the case values, leading to large control flow overhead for selecting the representative code regions. Test example 7 actually shows a remarkable improvement in performance because the original Open64 didn't use jump table to lower the switch-case statement, while the modified Open64 lowered to jump table and then performed the optimization described in this paper.

Table 1: Code size when using the methodology in Open64 and compared with other compilers, using –Os optimization level.

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**

Table 2: Comparing total text size and cycle performance of original and modified Open64 using –Os optimization level.

Test	Original Open64		Methodology in Open64		Percentage improvement	
	Text size	Kilo cycles	Text size	Kilo cycles	Text size	Cycle Perf.
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

5. Summary and conclusion

In this paper we described a code size optimization methodology based on difference classification and look-up table generation. The methodology can be consistently used for code size optimization of the non time-critical portions of the applications. The side benefits are improved instruction cache performance and increased availability of instruction bus bandwidth. The optimization has been implemented in the Open64 compiler, but could also be implemented in other compilers, e.g., GCC (versions 3.4.6. or 4.3.2).

The work to enhance Open64 for code size is ongoing, further enhancements are being looked into, as mentioned in section 1.2. We compared the enhanced Open64 compiler with the GNU 3.4.6 and 4.3.2 compiler versions for memory usage (i.e., code size) of the generated assembly code and compared the impact on cycle performance with and without the novel methodology in Open64. The results are promising and we are further enhancing the methodology for various types of control flow regions.

6. Acknowledgements

Thanks to other members of Qualcomm's DSP compiler team: Anshuman Dasgupta, Raja Venkateswaran, Sundeep Kushwaha, and Sergei Larin, for some of their valuable feedback on this work. Special thanks to Taylor Simpson, director of Qualcomm's DSP system software group, for supporting this work.

REFERENCES

[1] Saumya Debray, William Evans, Robert Muth, "Compiler Techniques for Code Compression".

[2] Keith D. Cooper and Nathaniel McIntosh, "Enhanced code compression for embedded RISC processors", Proc. SIGPLAN '99 Conference on Programming Language Design and Implementation, May 1999.

[3] Wen-Ke Chen, Bengu Li, Rajiv Gupta, "Code Compaction of Matching Single-entry Multiple-exit regions".

[4] J. Krinke, "Identifying similar code with program dependence graphs", WCRE, pages 301-309, Stuttgart, Germany, October 2001.

[5] John Gilbert, David M. Abrahamson. Adaptive object code compression. In proceedings of the 2006 international conference on compilers, architectures and synthesis for embedded systems. 2006.

[6] Warren Cheung, William Evans, Jeremy Moses. Predicated instructions for code compaction. Lecture Notes in Computer Science: Software and Compilers for Embedded Systems, 7th International 2826, 17-32. 2003.

[7] Open64, http://open64.sourceforge.net/

[8] WHIRL Intermediate Language Specification, whirl.pdf. http://open64.sourceforge.net

[9] WHIRL Symbol Table Specification, symtab_Pro64_SGI.pdf. http://open64.sourceforge.net

[10] Subrato De, Anshuman Dasgupta, Sundeep Kushwaha, Tony Linthicum, Susan Brownhill, Sergei Larin, Taylor Simpson, "Development of an efficient DSP compiler based on Open64," Open64 Workshop, 2008.

[11] Miroslav Popovic, "Communication Protocol Engineering", Edition: illustrated, Published by CRC Press, 2006 ISBN 0849398142, 9780849398148.

[12] Gábor Lóki, Ákos Kiss, Judit Jász, Árpád Beszédes, "Code Factoring in GCC", Proceedings of the GCC Developer Summit, June 2nd–4th, 2004, Ottawa, Ontario, Canada

[13] Brenda S. Baker, Udi Manber, "Deducing similarities in Java sources from bytecodes, " USENIX Technical Conference, 1998.