Code Compaction of Matching Single-Entry Multiple-Exit Regions *

Wen-Ke Chen, Bengu Li, and Rajiv Gupta

Dept. of Computer Science, The University of Arizona, Tucson, Arizona 85721

Abstract. With the proliferation of embedded devices and systems, there is renewed interest in the generation of compact binaries. Code compaction techniques identify code sequences that repeatedly appear in a program and replace them by a single copy of the recurring sequence. In existing techniques such sequences are typically restricted to single-entry single-exit regions in the control flow graph. We have observed that in many applications recurring code sequences form single-entry multiple-exit (SEME) regions. In this paper we propose a generalized algorithm for code compaction that first decomposes a control flow graph into a hierarchy of SEME regions, computes signatures of SEME regions, and then uses the signatures to find pairs of matching SEME regions. Maximal sized matching SEME regions are found and transformed to achieve code compaction. Our transformation is able to compact matching SEME regions whose exits may lead to a combination of identical and differing targets. Our experiments show that this transformation can lead to substantial reduction in code size for many embedded applications.

Keywords - code compaction, single-entry-multiple-exit regions, control flow signature, predicated execution.

1 Introduction

In the embedded domain, often applications are required to fit in a limited amount of memory and their execution is required to be energy efficient. One avenue of reducing the memory needs of an application program is through the use of code size reduction techniques. Code size reduction can also lead to better instruction cache performance and hence yield energy savings during execution. There are two broad categories of code size reduction techniques: *code compaction* techniques that produce directly executable code and are implemented entirely in software (e.g., [2, 4]); and *code compression* techniques that produce code that must be decompressed prior to execution (e.g., [3]). Techniques that rely on hardware support to generate compact binaries also exist [15, 10, 13, 6]. This paper addresses the problem of *code compaction*.

* Supported by grants from IBM, Intel and NSF grants CCR-0105355, CCR-0208756, CCR-0220334, and EIA-0080123 to the Univ. of Arizona.

Code compaction techniques. A commonly used approach to *code compaction* is the application of compiler based transformations which find recurring code sequences in an application and replace them by a single shared copy of the code sequence [16, 7, 17, 1, 2]. There are two commonly used transformations for removing repeated occurrences of a code sequence: *tail merging* [16, 7, 1] and *procedural abstraction* [17, 2]. If the recurring sequences appear along alternate paths immediately prior to a merge point in the control flow graph, *tail merging* is applied to replace multiple occurrences of the code sequence immediately before the merge point by a single occurrence of the code sequence following the merge point. If the control flow does not merge following the recurring code sequences, *procedural abstraction* is used. A procedure is created to contain the code sequence and this procedure is called from each point where the sequence occurs. Therefore, *procedural abstraction* introduces runtime overhead due to call and return while *tail merging* does not require this additional overhead.

An important aspect of the above code compaction transformations is the scope of code sequences to which the transformations are applied. Early works consider code sequences made up of a single basic block or a part (suffix) of a single basic block. More recent techniques have extended the transformations to consider code sequences that form *single-entry single-exit* (SESE) regions in the control flow graph. Moreover the code sequences need not be identical – as long as they can be made identical through renaming of variables (registers), the transformations can be applied. We will refer to such code sequences as *similar code sequences* and their corresponding control flow graph regions as *matching regions*. It should be noted that finding multiple instances of a single large matching region is better than finding several smaller matching regions. This is because while tail merging or a single application of procedural abstraction will be sufficient to compact the large regions, to individually compact the smaller subregions it contains will require multiple applications of procedural abstraction. The latter yields less compact code and introduces greater amount of call-return overhead.

Our contribution. By studying several embedded benchmarks we have observed that often *similar* code sequences that appear in applications are larger than a basic block but they do not form matching SESE regions. Instead we have observed frequent presence of *similar single-entry multiple-exit* (SEME) regions. Moreover, the exits of the matching regions may lead to a combination of same and different program points. To take advantage of the above situations we require a generalized algorithm for identifying matching regions and a generalized transformation to handle the complexity of the exits associated with these regions.

In this paper we present a generalized algorithm for detecting matching SEME regions and a generalized tail merging transformation to compact such regions. The two key components of our algorithm are as follows:

- *Finding matching SEME regions.* There are several problems that we solve to find matching regions. First we construct a *region hierarchy graph* (RHG)

which represents a decomposition of the program into a hierarchy of SEME regions. Second we compute *signatures* of these regions. Finally using a combination of *signatures* and the *RHG* we search for SEME regions that match. Our approach to matching signatures naturally allows small differences to be present between the matched regions which can be effectively handled during the compaction transformation.
- *Generalized tail merging transformation.* The key to developing this transformation is an ability to compact matching SEME regions such that their exits lead to a combination of same and different target points in the control flow graph. By introducing a *distinguishing predicate*, which identifies the context in which a compacted region is being executed, we are able to efficiently handle transfer of control to different targets from corresponding exits of the matching regions. Thus, without using procedural abstraction, we are able to compact matching SEME regions with a combination of exits with same and different targets.

Before we present the details of the above algorithm in the subsequent sections, we illustrate the power of our approach in handling situations taken from benchmarks. The example in Fig. 1 is taken from one of the `spec2000` benchmarks. It illustrates a situation in which two matching SEME regions, with three exits each, have all their exits leading to the same program point. Small amount of renaming is needed to make these regions identical and carrying out a more general form of tail merging for compaction as shown in Fig. 1. Thus not only is a large sequence of code reused, the overhead introduced by code compaction is minimal. What is important to note is that existing techniques will not handle this situation well – they will perform less compaction and introduce additional overhead. Since existing algorithms are based upon SESE regions, and the only SESE subregions in the SEME region are individual basic blocks, they will transform each pair of corresponding basic blocks separately. While the three basic blocks from which we exit the SEME region (3, 4, and 5) can be transformed using tail merging, the other two basic blocks (1 and 2) will be transformed using procedural abstraction which will introduce additional call-return overhead. Finally since the control structures of the SEME regions will stay in place, the conditional branches in the two regions at which basic blocks 1 and 2 terminate are not compacted at all. Thus, existing algorithms will achieve less compaction at greater overhead cost while our algorithm will perform much better by compacting the entire SEME region in a single application of a generalized tail merging transformation.

While in the above example the SEME regions shared a single identical target for all the exits, in general this may not be the case. Two such situations that we observed in benchmarks and are handled by us are illustrated in Fig. 2 and Fig. 3. In Fig. 2 after exiting region SEME(y), an extra statement S is executed and then the same program point is reached that is also the target of exits of region SEME(x). By introducing a conditional execution of S we carry out compaction as shown in the figure. In Fig. 3 we can see that pair of corresponding exits from matching regions lead to different targets (T_1 and T_2). In this case we

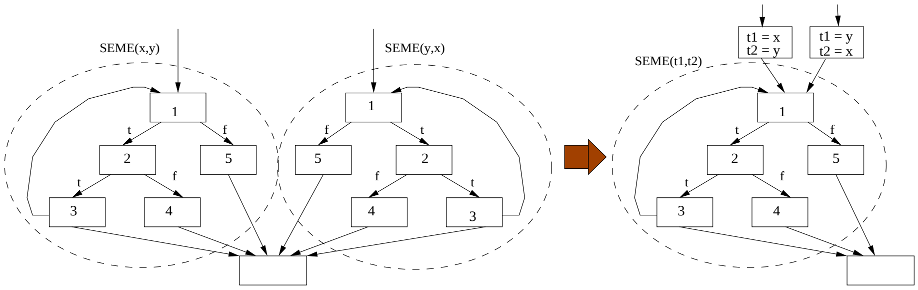


Fig. 1. spec2000::parser::parse.c::match()

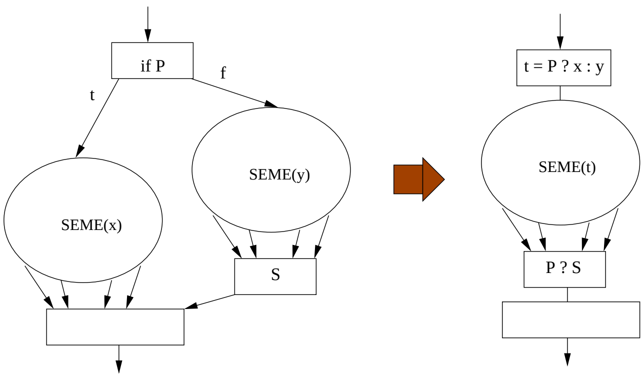


Fig. 2. spec2000::twolf::unetseg.c::unetseg()

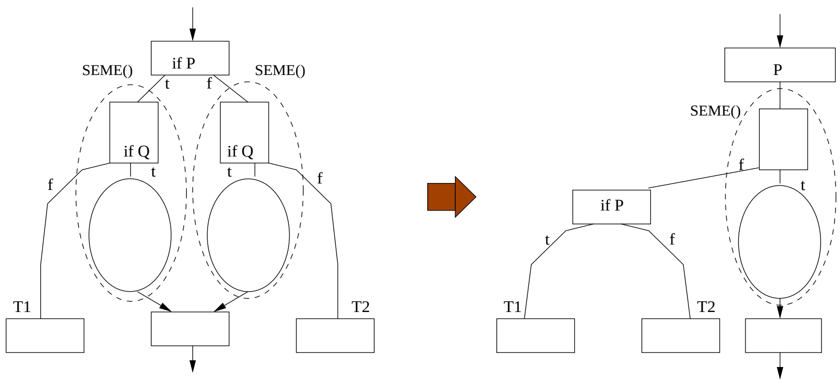


Fig. 3. MediaBench::mesa::clip.c::gl_viewclip_line()

introduce a conditional selection of the target at the corresponding exit following compaction. Both of the above examples are being essentially compacted using a common approach for handling differences between the matching SEME regions. We find a *distinguishing predicate* (denoted by P in our examples) which essentially will capture the context in which the compacted code sequence is being executed (i.e., if SEME_1 and SEME_2 are replaced by SEME_{12}, during an execution of SEME_{12}, P will indicate whether the execution corresponds to an execution of SEME_1 or SEME_2). Thus, P can be used to perform conditional execution of statements in the matching SEME regions that need to be handled differently. While in our examples a distinguishing predicate was already present in the program, if one is not present it is explicitly introduced by our transformation. These examples also illustrate that the transformation we use is essentially a generalized form of tail merging which incurs lower overhead than procedural abstraction. Clearly the large SEME regions present in the above examples will not be handled effectively by existing techniques as existing techniques can at best transform smaller SESE subregions using, in many cases, the more expensive procedural abstraction transformation.

Outline. The remainder of the paper is organized as follows. In section 2 we develop the *region hierarchy graph* that decomposes a program into a hierarchy of SEME regions. In section 3 we describe our algorithm for computing *signatures* and *finding matching SEME regions.* In section 4 we present the *generalized tail merging* transformation. Experimental results are presented in section 5 and concluding remarks are given in section 6.

2 Hierarchy of SEME Regions

Given a control flow graph (CFG), SEME regions can be formed in a number of ways. Therefore as a first step it is important that we find a systematic method for forming SEME regions. The approach we take is to develop a *region hierarchy graph* (RHG) which decomposes the program into a hierarchy of SEME regions. A hierarchical representation is needed because larger SEME regions are formed by combining smaller SEME regions. Using this graph we will be able to explore all possible SEME regions that can be formed for a given control flow graph.

While an algorithm for decomposing a program into a hierarchy of SESE regions exists [9], no such algorithm has been developed for SEME regions. Next we present an algorithm that we have developed for building a RHG for SEME regions. For this purpose we first construct the *control dependence graph* (CDG) [5] corresponding to the given CFG. Let us briefly review the structure of the CDG and then see why CDG is an appropriate choice for building the RHG.

A CDG partitions the program into *control dependence regions.* Corresponding to each control dependence region, it creates a special region node which directly points to all the basic blocks within that region. In addition, it points to other nested control dependence regions. We assume that nodes within a control dependence region are ordered from left to right according to the order in which they appear in the program with an earlier node always appearing to the left of a later node. The set of basic blocks that belong to the same control dependence region are *control equivalent*, i.e. the conditions under which they are executed are identical. Finally the internal nodes of a CDG are made up of two types of nodes: those created to represent control dependence regions and those that correspond to basic blocks ending with conditionals.

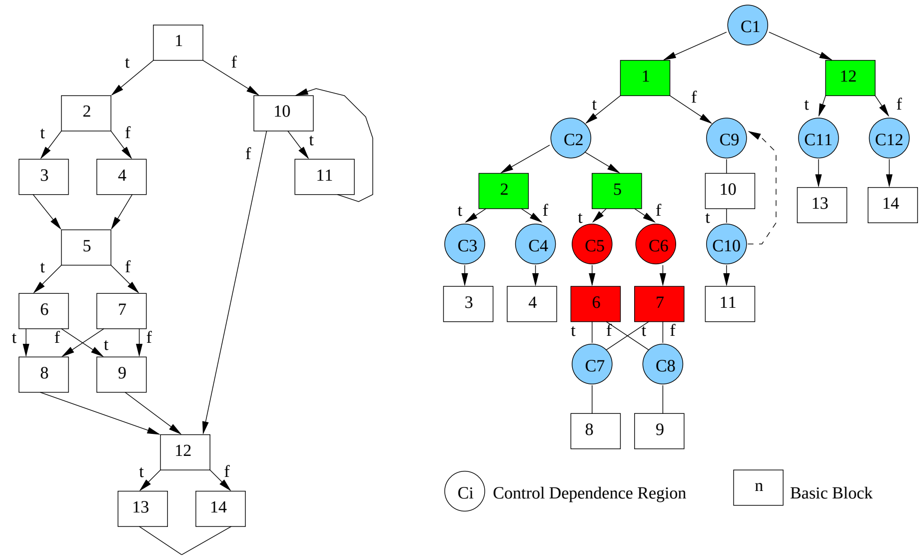


Fig. 4. Example Control Flow Graph and its Control Dependence Graph.

Consider an example CFG and its corresponding CDG shown in Fig. 4. As we can see, nodes 1 and 12 belong to the same control dependence region C_1 because they are control equivalent. Internal nodes include control dependence region nodes C_1 through C_{12} and basic blocks ending at conditionals (i.e., nodes 1, 2, 5, 6, 7, 10, and 12).

Our motivation for using the CDG as the basis for computing the RHG is as follows. First CDG is already a hierarchical decomposition of the program such that each internal node of this graph represents a program region. Second, while each such region corresponds to a subgraph of the CFG which may have single or multiple entries and exits, it is easy to distinguish single entry regions from multiple entry regions. In particular if there does not exist any edge from outside the region to a node within the region (see Fig. 5) then the region has only one entry node which is the first basic block encountered during the *in order* traversal of the region subgraph in the CDG. For example, in Fig. 4 the regions C_5 and C_6 are multi-entry regions.

Given that we can classify each internal node of the CDG as representing either a multi-entry or single-entry region, we can further construct a region hierarchy graph whose nodes represent single-entry regions where each region may have one or more exits. Since each node in the RHG represents a single

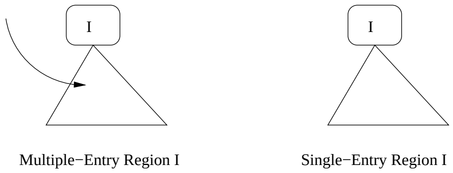


Fig. 5. Distinguishing Single-Entry Regions from Multiple-Entry Regions.

entry region, the RHG is a tree. Next we define the RHG formally and present an algorithm for constructing it.

A *SEME region R* is denoted as $R = (n_{entry}, N)$ where n_{entry} is the *single entry node* of the region in the CFG formed by the nodes in N. Each control flow graph edge whose source node is in N, but the destination node does not belong to N, represents an *exit* from region R. If R has a single exit, then it represents a SESE region, i.e., by definition SEME regions subsume SESE regions.

Definition 1. A *region hierarchy graph* (RHG) is a tree in which each node represents a SEME region corresponding to the subgraph rooted at a single-entry node identified in the CDG. The children of a node SEME_i in the RHG are the largest distinct SEME regions entirely contained within SEME_i.

The algorithm for constructing the RHG is given Fig. 6. The algorithm examines each internal node and if it represents a single entry region it constructs a region and adds it to the region set $\mathcal{RS}$. If the internal node is a basic block, it is the entry node for the region. If the internal node is a control dependence region node, then we keep taking the leftmost link till the first basic block is encountered. This basic block is the entry node. All basic blocks that are descendants of the internal node belong to the region. The construction of the RHG can be implemented efficiently through a single bottom-up traversal of the CDG and thus the runtime cost of building the RHG is proportional to the size of the CDG. The RHG corresponding to the CDG in Fig. 4 is shown in Fig. 6.

3 Searching for Matching SEME Regions

There are two problems that are addressed in this section: *searching* and *matching*. While there exist some previous work [11, 12] on identifying matching fragments, these techniques can not be applied directly to SEME matching. Given two SEME regions, matching is the process that finds out whether the two regions are *sufficiently similar* while searching is the process that selects pairs of SEME regions for matching and if they match, it iteratively expands and matches them to find larger matching regions. To carry out matching we define a method of computing *signatures* of SEME regions which are used as the basis of matching. This method is described first in the section. Following that we will show how RHG can be used to conveniently implement the search of matching SEME regions.

Input: A control flow graph (CFG) and its control dependence graph (CDG).
Output: A set of SEME regions, $\mathcal{RS}$.
definitions:
  backedges, $\mathcal{BE} = \{\ (s,d) : (s,d) \in$ CDG is a loop back edge $\}$
  descendant set of an internal node $c \in$ CDG,
    $\mathcal{DESCENDANT}(c) = \{\ n : \exists$ a path $c \rightarrow n$ in CDG which contains no edge from $\mathcal{BE}\ \}$
  reaching set of an internal node $c \in$ CDG,
    $\mathcal{REACH}(c) = \{n : n$ is a basic block; $n \in \{c\} \cup \mathcal{DESCENDANT}(c)\ \}$
initialization:
  $\mathcal{RS} = \phi$
Algorithm:
  foreach internal node $c \in$ CDG in reverse topological order **do**
    if $\exists$ an edge (s,d) st $d \in \mathcal{DESCENDANT}(c)$ and $s \notin \{c\} \cup \mathcal{DESCENDANT}(c)$
    then– do nothing; c represents a multiple-entry region
    else – add region for c to $\mathcal{RS}$
      if c is a basic block **then**
        $\mathcal{RS} = \mathcal{RS} \cup \{(n_{entry} = c, \mathcal{REACH}(c))\}$
      else – c is a control dependence region node
        Let l be the first basic block reached by following leftmost links starting at $c \in$ CDG
        $\mathcal{RS} = \mathcal{RS} \cup \{(n_{entry} = l, \mathcal{REACH}(l))\}$
      endif
    endif
  endfor

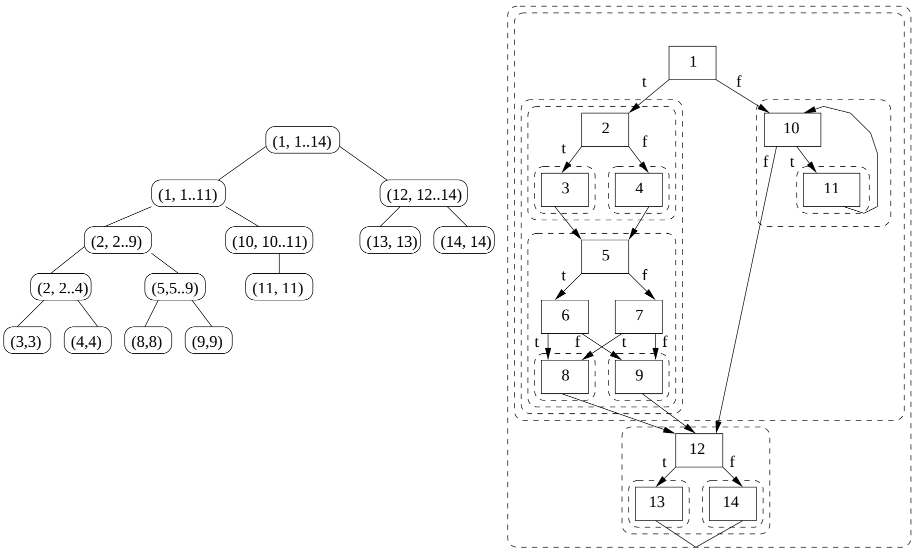


Fig. 6. SEME Region Hierarchy: Construction Algorithm and Example.

Signatures. We use a pair of signatures to characterize a SEME region. The first level is the *control flow signature* (CFS) which simply captures the shape of the SEME region. If two SEME regions have the same control flow signature, then they must have the same shape, i.e. the regions contain identical number of nodes connected with each other by identical edges and the single entry as well as the exits are in corresponding positions. Once the control flow signatures of two regions match, a correspondence is established between the basic blocks of the two regions. At this point the second level signatures, which are essentially the *data flow signatures* (DFS) of each of the basic blocks, are compared. In the remainder of this discussion we will mainly concentrate on control flow signatures because data flow signatures of various kinds have already been developed by other researchers to match code sequences that extend across a complete basic block or that form a suffix of a basic block [7, 2]. These existing techniques also allow for differences that can be overcome using variable renaming.

Our algorithm for generating the control flow signature generates a serialization of the control flow graph in which all nodes, edges, label on edges, entry, and exits are included. The nodes are renamed before inclusion in this serialization. Thus, if two SEME regions have the same shape, they produce the same signature when processed by our signature generating algorithm.

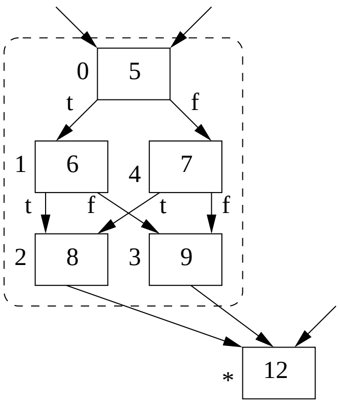
Fig. 7 presents a function CFS(n, R) which when called with n being the entry node of region R, returns the CFS for region R. It carries out an *in order*

```
CFS (n, R = (n_e, N)) {
    if (AN(n) ≠ null) then
      if (n ∈ N) then
        return( (AN(n), in) )
      else
        return( (AN(n), out) )
      endif
    else
      if (n ∈ N) then
        AN(n) = count; count++;
        if (Succ(n) = {n_s}) then
          return( (AN(n), in) o (−) o CFS(n_s, R) )
        elseif (Succ(n) = {n_t, n_f}) then
          return( (AN(n), in) o(T) o CFS(n_t, R) o (F) o CFS(n_f, R) )
        endif
      else
        AN(n) = ∗; return( (AN(n), out) )
      endif
    endif
}
main () {
    count = 0; CFS_R = CFS(n_entry, R = (n_entry, N));
}
```

Fig. 7. Computing the Control Flow Signature of a Region.

traversal of the region R starting at the entry node such that no edge is visited twice. Each node in the region is assigned a new unique id/name $\mathcal{AN}(n)$ when it is visited for the first time prior to which all nodes are assigned a *null* id. All nodes and edges contained within the region are visited once. In addition all exit edges for the regions as well as the nodes outside the region to which the exit edges lead are also visited once. These outside nodes are assigned an $\mathcal{AN}(n)$ value of *. This is because, for our compaction transformation to be applicable, while the positions of exits of matching SEME regions must be the same, their targets need not be the same. The signature is made up of sequence of nodes and edges that are visited, in the order that they are visited. A visit to node n adds $(\mathcal{AN}(n), in)$ to the signature if n belongs to the region and $(\mathcal{AN}(n), out)$ if it is out of the region (i.e., it is a target of an exit edge). The traversal of an edge is indicated by adding $(-)$, (T), and (F) to the signature if the edge is not labeled, labeled true, and labeled false respectively. When two SEME regions generate the same signature, the $\mathcal{AN}$ values assigned to the nodes establish a correspondence between the nodes, edges and exits of the two regions. This information can then be used to match the data flow signatures of the corresponding nodes and carrying out the compaction transformation itself. The runtime cost of computing the signature is proportional to the size of the SEME region.

Fig. 8 illustrates the computation of the control flow signature of a region taken from the flow graph of Fig. 4. As we can see, the signature contains eight edges since the region contains six internal edges and two exit edges. The nodes that connect these edges are also a part of the signature. The signature constructed in this manner uniquely characterizes the SEME region.



n	$\mathcal{AN}(n)$
5	0
6	1
7	4
8	2
9	3
12	*

$(0, in)$
$\quad o(T)o(1, in)$
$\qquad o(T)o(2, in)o(-)o(*, out)$
$\qquad o(F)o(3, in)o(-)o(*, out)$
$\quad o(F)o(4, in)$
$\qquad o(T)o(2, in)$
$\qquad o(F)o(3, in)$

Fig. 8. CFS($R = (5, \{5, 6, 7, 8, 9\})$).

Search strategy. Given the above method we can generate the signatures of all the SEME regions represented explicitly in the region hierarchy graph. However, if we simply find matching regions by comparing the signatures of these regions, we may only find small matching regions. This is because all SEME regions are not represented explicitly in the RHG. Consider the RHG of Fig. 9. Node 4 contains three regions 2, 1 and 3. While these contained regions represent disjoint SEME regions that together form region 4, they can be used to form other valid SEME regions composed of (2,1) and (1,3). Thus to find maximal matching regions we must consider both SEME regions that are explicitly and implicitly represented by the RHG.

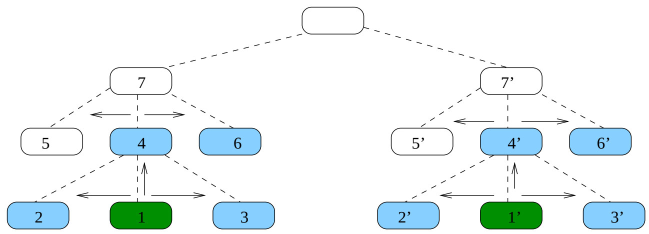


Fig. 9. Searching the RHG.

To find a pair of matching regions we first find a pair of matching leaf regions. Then we gradually grow these regions as long as both can be expanded such that they still continue to match. The leaf regions are expanded by first attempting to incorporate their siblings to the left and then siblings to the right. If all siblings get incorporated, then we go to the parent and start expansion again. Thus this approach explores both SEME regions that are explicitly and implicitly represented by the RHG. It is important to once again note that this simple search is made possible because the RHG was constructed using the CDG. Siblings belong to the same control dependence region and hence can be joined to form larger SEME regions.

The algorithm given in Fig. 10 presents in detail the search process and matching process. While the algorithm is based upon our earlier descriptions of this section, there are a few things that are worth pointing out. First in the search algorithm $Matchleft*$ and $Matchright*$ continue to expand a region by respectively including left and right siblings till no more can be matched or none exist. During this matching process while the positions of the exits from the SEME regions are matched, their targets are not matched. This is because our transformation is powerful enough to compact similar SEME regions even when the targets of their exits are not the same. The second thing worth noting is that a call to $Match(R_1, R_2)$ returns the ratio of the code size that will be introduced (i.e., code to set up variables introduced by renaming and to initialize the distinguishing predicate that may be needed to express conditional execution in the transformed code) and the size of code that will be removed. If this ratio is no larger than a preset threshold value, then we consider the regions to match and worthy of compaction. The number of matches performed by the above algorithm, which bounds its runtime cost, is $O(n^2)$.

$\mathcal{LRS} = \{L : L \in RHG,\ st\ it\ contains\ no\ nested\ regions\}$
while $\exists\ R_1,\ R_2 \in \mathcal{LRS}$ st CFS(R_1) =CFS(R_2) **do**
  if $Match(R_1, R_2) \leq Threshold$ **then**
    $\mathcal{R}_1 = \{R_1\};\ \mathcal{R}_2 = \{R_2\};$
    repeat
      if $\mathcal{R}_1 = Children(P_1)$ and $\mathcal{R}_2 = Children(P_2)$ **then**
        $\mathcal{R}_1 = \{P_1\};\ \mathcal{R}_2 = \{P_2\}$
      endif
      $\mathcal{OR}_1 = \mathcal{R}_1;\ \mathcal{OR}_2 = \mathcal{R}_2;$
      $(\mathcal{R}_1, \mathcal{R}_2) = Matchleft * (\mathcal{R}_1, \mathcal{R}_2);$
      $(\mathcal{R}_1, \mathcal{R}_2) = Matchright * (\mathcal{R}_1, \mathcal{R}_2);$
    until $\mathcal{OR}_1 = \mathcal{R}_1\ and\ \mathcal{OR}_2 = \mathcal{R}_2;$
    $\mathcal{MRS} = \mathcal{MRS} \cup \{(\mathcal{R}_1, \mathcal{R}_2)\}$
  endif
endwhile

Match (R_1, R_2) {
  if CFS(R_1) =CFS(R_2) **then**
    $Pred \leftarrow false;\ Rename \leftarrow \phi$
    foreach $b_1 \equiv b_2$ st $b_1 \in R_1$ and $b_2 \in R_2$ **do**
      if DFS(b_1) =DFS(R_2) **then**
        $Rename \leftarrow Rename \cup diff(b_1, b_2)$
      else $Pred \leftarrow true$ **endif**
    endfor
    $ExtraI = ExtraI(Pred) + ExtraI(Rename)$
    return ($ExtraI / MaxI(R_1, R_2)$)
  endif
}

Fig. 10. Searching for Matching Regions.

4 Generalized Tail Merging Transformation

Finally, given two matching SEME regions SEME(x) and SEME(y), we develop a transformation to compact the two regions. The transformation is pictorially illustrated in Fig. 11. There are three main components of this transformation. First a renamed version of the SEME region SEME(t) is constructed to replace SEME(x) and SEME(y). We do not discuss details of renaming as this is a known technique. Second the incoming edges of SEME(x) and SEME(y) are made the incoming edges of SEME(t). However, along these incoming edges additional code is introduced to carry out two functions: initializing variables introduced during *renaming* and initialization of the *distinguishing predicate* if one is needed. During execution of SEME(t), the value of the distinguishing predicate determines whether an execution of SEME(t) in the transformed code corresponds to an execution of SEME(x) or an execution of SEME(y) in the original code. Third the exits of the SEME(t) are connected.

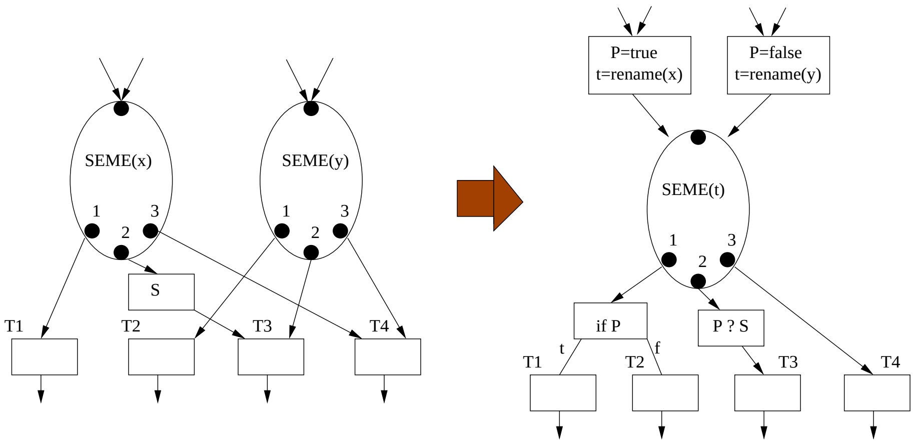


Fig. 11. Generalized Tail Merging.

Three different situations arise when handling the exits which are illustrated in Fig. 11 by corresponding exits marked 3, 2, and 1. The targets of the pair of corresponding exits marked 3 in SEME(x) and SEME(y) are identical (T_4) and thus exit marked 3 in SEME(t) is connected to T_4. The exit marked 2 leads to target T_3 but there is a small difference between SEME(x) and SEME(y). There is an extra statement S between 2 in SEME(x) and T_3. This statement's execution is made conditional upon the predicate P (denoted by $P?S$) and 2 in SEME(t) is connected to T_3 via statement $P?S$. The targets of the pair of exits marked 1 lead to entirely different points in the program, i.e. T_1 and T_2. Therefore at the exit marked 1 in SEME(t) we place a condition that tests predicate P and based upon the outcome appropriately transfers control to either T_1 or T_2 as shown in Fig. 11. Detailed algorithm for carrying out the transformation is presented in Fig. 12.

From the above transformation we can see that it is essentially a generalization of tail merging so that we never have to make use of procedural abstraction. Like tail merging, when corresponding exits of matched regions lead to the same target point, the corresponding exit in the transformed code is simply connected to the same target. Thus, no additional branching overhead is introduced for this exit. In other situations, where targets are not the same or an extra block is present in one case, the distinguishing predicate is used to allow selection of the target or conditional execution of the extra statement. Thus, by introducing the distinguishing predicate we enable conditional execution possible and eliminate the need for using procedural abstraction. It should also be noted that if any two corresponding basic blocks in SEME(x) and SEME(y) differ, then they can also be merged and appropriately executed by testing predicate P. This form of conditional execution does not require introduction of explicit branches as modern processors such as Intel's IA64 support predicated execution of instructions.

Renamed SEME Region:

Given matching regions SEME($x_1, \cdots, x_n$) and SEME($y_1, \cdots, y_n$)
  Create renamed region SEME($t_1, \cdots, t_n$).
  Introduce renaming code along entry corresponding to SEME($x_1, \cdots, x_n$)
    $(t_1, \cdots, t_n) \leftarrow rename(x_1, \cdots, x_n)$
  Introduce renaming code along entry corresponding to SEME($y_1, \cdots, y_n$)
    $(t_1, \cdots, t_n) \leftarrow rename(y_1, \cdots, y_n)$

Distinguishing Predicate:

Predicate P distinguishing an execution of SEME($x_1, \cdots, x_n$) from
  SEME($y_1, \cdots, y_n$) if needed iff:
  $\exists$ exits $E_1 \equiv E_2$ from SEME($x_1, \cdots, x_n$) and SEME($y_1, \cdots, y_n$)
    which do not lead to the same target.
  if P does not naturally occur in the program, create P by introducing:
    $P = true$ along entry to SEME($x_1, \cdots, x_n$); and
    $P = false$ along entry to SEME($y_1, \cdots, y_n$).

Handling Exits:

foreach pair of exits $E_x \equiv E_y$ from SEME($x_1, \cdots, x_n$) and SEME($y_1, \cdots, y_n$) **do**
  Let T_x and T_y be the targets of E_x and E_y;
  if $T_x = T_y$ **then**
    Connect corresponding exit E_t in SEME($t_1, \cdots, t_n$) to $T_x (= T_y)$
  elseif T_x *postdominates* T_y or T_y *postdominates* T_x via S **then**
    Connect corresponding exit E_t in SEME($t_1, \cdots, t_n$) via
      predicated execution of S, $P?S$ to the appropriate target (T_x or T_y)
  else
    Introduce the following test on P for selection of target:
      "$if\ P\ then\ goto\ T_x\ else\ goto\ T_y$"
  endif
endfor

Fig. 12. Code Compaction Transformation Algorithm.

5 Experimental Results

We looked for benefits of applying our approach to functions from a few benchmark programs taken from the `Mediabench` [14] and `MiBench` [8] embedded suites and `spec2000` suite. We found many instances of regions where our transformation is beneficial. Table 1 shows the characteristics of the matching regions that were compacted. It should be noted that in some functions SEME regions were found but since all exits went to same targets, distinguishing predicate was not needed. In other cases while the distinguishing predicate was needed, the regions were merely SESE regions. Finally in other cases both SEME regions were compacted and distinguishing predicate was also needed. Thus, the various elements of our generalized tail merging transformation plays an important role in the compaction of code for all these functions.

Table 1. Matching Region Characteristics.

Benchmark:: Function	Characteristics
gsm:: LARp_to_rp	Distinguishing Predicate; SESE
gsm:: Long_term_analysis_filtering	Distinguishing Predicate; SESE
mesa:: update_pixel_logic	SEME
mesa:: gl_viewclip_line	Distinguishing Predicate; SEME
mesa:: gl_viewclip_polygon	Distibguishing Predicate; SEME
mesa:: gl_userclip_line	Distinguishing Predicate; SESE
mesa:: gl_userclip_polygon	Distinguishing Predicate; SESE
tiff-v3.5.4:: PackBitsEncode	SEME
sphinx:: sorted_id	SEME
twolf:: unetseg	Distinguishing Predicate; SEME
parse:: match	SEME
craft:: RepetitionDraw	Distinguishing Predicate; SESE
gzip:: inflate_codes	SEME
gzip:: inflate_dynamic	Distinguishing Predicate; SEME

We computed the code sizes of these functions by compiling the original and transformed (i.e., compacted) source codes into ARM instructions. All benchmarks were compiled using `gcc 2.95.2` with options `-O2 -march=armv4`. For comparison we also computed the code size reduction achieved by existing techniques that apply compaction transformations to SESE regions using branch and link instructions. Table 2 shows the resulting data for named functions taken from benchmarks. As we can see, the code size reductions are substantial in most cases. While our approach gave an average reduction of 24.96%, existing techniques provide an average reduction of 14.56% in code size.

Table 2. Code Size Reduction.

Benchmark:: Function	Original Size	SEME		SESE	
		Size	Reduction(%)	Size	Reduction(%)
gsm:: LARp_to_rp	74	45	39.19	49	33.78
gsm:: Long_term_analysis_filtering	160	64	60.00	76	52.50
mesa:: update_pixel_logic	65	47	27.69	61	6.15
mesa:: gl_viewclip_line	818	666	18.58	674	17.60
mesa:: gl_viewclip_polygon	1326	982	25.94	1126	15.08
mesa:: gl_userclip_line	306	262	14.38	283	7.52
mesa:: gl_userclip_polygon	395	341	13.67	350	11.39
tiff-v3.5.4:: PackBitsEncode	150	121	19.33	139	7.33
sphinx:: sorted_id	83	46	44.58	61	26.56
twolf:: unetseg	1627	1588	02.40	1627	0.00
parse:: match	120	94	21.67	118	1.67
craft:: RepetitionDraw	60	41	31.67	54	10.00
gzip:: inflate_codes	310	260	16.13	284	8.39
gzip:: inflate_dynamic	458	393	14.19	431	5.90
average			24.96%		14.56%

6 Conclusions

In this paper we presented a generalized tail merging transformation which achieves two goals. First it enables application of code compaction to large SEME regions which cannot be compacted by a single application of existing transformations that function only for SESE regions. Second the use of distinguishing predicate enables the application of our tail merging style transformation in the presence of small differences between the matching regions. Existing techniques cannot handle SEME regions and thus in contrast they will rely upon multiple applications of the procedural abstraction transformation to SESE subregions of the SEME region. Therefore they will achieve less compaction and introduce greater runtime overhead due to calls and returns. Thus we are able to carry out greater degree of compaction with reduced compaction overhead.

References

1. K. Cooper and N. McIntosh, "Enhanced Code Compression for Embedded RISC Processors," *SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), May 1999.
2. S. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler Techniques for Code Compaction," *ACM Transactions on Programming Languages and Systems* (TOPLAS), vol. 22, no. 2, pages 378-415, March 2000.
3. S. Debray and W. Evans, "Profile-Guided Code Compression," *SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 95-105, June 2002.

4. B. De Sutter, B. De Bus and K. De Bosschere, "Sifting out the Mud: low level C++ Code Reuse," *SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA), pages 275-291, Seattle, Wanshington, November 2002.
5. J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems* (TOPLAS), vol. 9, no. 3, pages 319-349, 1987.
6. C. Fraser, "An Instruction for Direct Intepretation of LZ77-compressed Programs," Technical Report, Microsoft Research, MSR-TR-2002-90, September 2002.
7. C. Fraser, E. Myers, and A. Wendt, "Analyzing and Compressing Assembly Code," *ACM SIGPLAN Symposium on Compiler Construction* (CC), 1984.
8. M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *IEEE 4th Annual Workshop on Workload Characterization* (WWC), Austin, TX, December 2001.
9. R. Johnson, D. Pearson, and K. Pingali, "The Program Structure Tree: Computing Control Regions in Linear Time," *SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 171-185, 1994.
10. D. Kirovski, J. Kin, and W. H. Mangione-Smith, "Procedure Based Program Compression," *30th Annual ACM/IEEE International Symposium on Microarchitecture* (MICRO), pages 204-217, 1997.
11. R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," *International Static Analysis Symposium* (SAS), pages 40-56, Paris, France, July 2001.
12. J. Krinke, "Identifying Similar Code with Program Dependence Graphs," *Working Conference on Reverse Engineering* (WCRE), pages 301-309, Stuttgart, Germany, October 2001.
13. A. Krishnaswamy and R. Gupta, "Profile Guided Selection of ARM and Thumb Instructions," *ACM SIGPLAN Joint Conference on Languages Compilers and Tools for Embedded Systems & Software and Compilers for Embedded Systems* (LCTES/SCOPES), pages 55-63, Berlin, Germany, June 2002.
14. C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *IEEE/ACM International Symposium on Microarchitecture* (MICRO), Research Triangle Park, North Carolina, December 1997.
15. C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge, "Improving Code Density Using Compression Techniques," *30th Annual ACM/IEEE International Symposium on Microarchitecture* (MICRO), pages 194-203, 1997.
16. W. Wulf, R. Johnson, C. Weinstock, S. Hobbs, and C. Geschke, "The Design of an Optimizing Compiler," American Elsevier, New York, 1975.
17. M. Zastre, "Compacting Object Code via Parameterized Procedural Abstraction," MS Thesis, University of Victoria, 1995.