A Hierarchical Region-Based Static Single Assignment Form

Jisheng Zhao and Vivek Sarkar

Department of Computer Science, Rice University
{jisheng.zhao,vsarkar}@rice.edu

Abstract. Modern compilation systems face the challenge of incrementally re-analyzing a program's intermediate representation each time a code transformation is performed. Current approaches typically either re-analyze the entire program after an individual transformation or limit the analysis information that is available after a transformation. To address both efficiency and precision goals in an optimizing compiler, we introduce a hierarchical static single-assignment form called Region Static Single-Assignment (Region-SSA) form. Static single assignment (SSA) form is an efficient intermediate representation that is well suited for solving many data flow analysis and optimization problems. By partitioning the program into hierarchical regions, Region-SSA form maintains a local SSA form for each region. Region-SSA supports a demand-driven re-computation of SSA form after a transformation is performed, since only the updated region's SSA form needs to be reconstructed along with a potential propagation of exposed defs and uses.

In this paper, we introduce the Region-SSA data structure, and present algorithms for construction and incremental reconstruction of Region-SSA form. The Region-SSA data structure includes a tree based region hierarchy, a region based control flow graph, and region-based SSA forms. We have implemented in Region-SSA form in the Habanero-Java (HJ) research compiler. Our experimental results show significant improvements in compile-time compared to traditional approaches that recompute the entire procedure's SSA form exhaustively after transformation. For loop unrolling transformations, compile-time speedups up to 35.8× were observed using Region-SSA form relative to standard SSA form. For loop interchange transformations, compile-time speedups up to 205.6× were observed. We believe that Region-SSA form is an attractive foundation for future compiler frameworks that need to incorporate sophisticated incremental program analyses and transformations.

1 Introduction

Recent processor architectures increasingly rely on multiple levels of locality and parallelism to improve performance. To realize this performance, it has become critical for compilers to incorporate an increasing number of sophisticated analyses and transformations. Examples include procedure-level transformations such as inline expansion of calls, loop-level transformations such as interchange, reversal, skewing, distribution, fusion, and unrolling [16, 3, 25, 20], and classical optimizations such as common subexpression elimination, partial redundancy elimination and constant propagation [1]. In

some cases, search and machine learning techniques are used to explore a large space of transformations [4, 17], further exacerbating the re-analysis problem.

The challenge that arises here is: how to build a flexible compiler infrastructure to enable sophisticated and incremental program analyses and optimizations to be performed efficiently? To address this problem, there are two important issues that need to be considered: *efficiency* and *precision*.

When considering the efficiency challenges, we observe that the time taken by a compiler to update its intermediate language (IL) when implementing a transformation is usually proportional to the size of the *region* being transformed, e.g., it just takes constant time to remove a "dead store" [1] from a list of IL instructions, and the compile-time required to unroll a loop is proportional to the size of the unrolled loop body. But the time taken to recompute analysis information exhaustively after each transformation is (at least) linear in the size of the entire procedure being compiled and thus more expensive than performing the transformation. In general, the cost of repeatedly performing exhaustive analysis after each transformation becomes highly prohibitive. Many compilers deal with this problem by assuming a fixed ordering for optimizations/transformations, and performing exhaustive reanalysis only after predefined batches/phases of transformations are completed (thus amortizing the overhead of exhaustive reanalysis over multiple transformations). This approach is restrictive because it becomes difficult to adapt the optimizer to different orderings of optimizations. In addition, this approach is error-prone because the analysis information is usually not entirely current when multiple transformations are processed without reanalysis.

We believe that the design of the internal representation is a critical issue in determining a compiler's ability and effectiveness in dealing with incremental reanalysis. It is of paramount importance for the internal representation to be flexible enough to accommodate incremental updates of analysis information in a specified region, as well as changes in the orderings of transformations and optimizations on different regions. Many internal program representations have been proposed in the past, including the Control Flow Graph [1], Interval Structure [2, 21], Program Dependence Graph [12], Static Single Assignment form [7], Forward Control Dependence Graph [11, 9, 19], Hierarchical Structured Control-flow Graph (HSCG) [13] and the Hierarchical Task Graph (HTG) [18]. These representations are convenient for certain aspects of program analysis and code generation, but updating these representations after transformation usually requires exhaustive reanalysis. The overhead of exhaustive reanalysis becomes even more burdensome when compilers need to simultaneously use more than one of these representations, because many of these representations are incomplete in capturing all the control and data information of interest to optimizing compilers.

Given the reasons listed above, we chose to use a hierarchical approach in the design of a new internal representation for incremental reanalysis. We also decide to use Static Single-Assignment (SSA) form [8] as the foundation for our IR, since it is an efficient intermediate representation that is well suited to a large number of compiler analyses and optimizations (e.g. constant propagation [24], partial redundancy elimination [14]). SSA form has also been implemented in a large number of research and production compilers. Thus, a hierarchical SSA form is a promising approach for supporting the efficiency and precision goals of modern optimizing compilers.

To address the efficiency and precision goals, we introduce a hierarchical static single-assignment form called Region Static Single-Assignment (Region-SSA) form. Region-SSA form is based on partitioning the program into the hierarchical regions which is tree structure named as Region Structure Tree (RST). For each region R in the RST, it maintains an individual Region Control Flow Graph (RCFG). Together, the RST and RCFG's form the Region Structure Graph (RSG). To achieve a hierarchical SSA form, we construct a local SSA form for each region's CFG in the RSG.

In this paper, we introduce algorithms for constructing Region-SSA form, and demonstrate its use in incremental re-analysis after selected transformations are performed. Our experimental results show significant improvements in compile-time compared to traditional approaches that recompute the entire procedure's SSA form exhaustively after transformation. For loop unrolling transformations, compile-time speedups up to 35.8× were observed using Region-SSA form relative to standard SSA form. For loop interchange transformations, compile-time speedups up to 205.6× were observed.

The rest of the paper is organized as follows. Section 2 describes the components of the Region Structure Graph (RSG) and their construction algorithms. Section 3 describes how a hierarchical Region-SSA can be built upon the RSG. Section 4 contains our experimental results. Section 5 discusses related work, and Section 6 contains our conclusions.

2 Region Structure Graph

2.1 Region Structure Tree

The Region Structure Tree (RST) represents the region nesting structure of the method being compiled. Each region node (R-node) of the RST represents a *single-entry region* of the original (flat) control flow graph (CFG) for the input method, and each leaf node (L-node) of the RST corresponds to a node in that CFG. Hierarchical nesting of regions is captured by the parent-child relation in the RST. The RST can accommodate any partitioning of the CFG into hierarchical single-entry regions.

The root node of the RST represents the entire method being compiled. A non-root R-node represents a subregion of its parent's region. An L-node represents a CFG node (basic block) that is contained within the region corresponding to the parent of the L-node. The choice of basic block granularity for leaf nodes is made by the RSG user (compiler writer) when constructing the CFG graph from which the RSG is initialized. For example, if the RSG user is interested in statement-level transformations, each statement should be mapped to a distinct node in the CFG. In this paper, we will illustrate the RST using statement-level basic blocks.

We impose three key constraints on legal region structures in a RST:

1. **Tree Structure** The nesting structure for regions must form a single connected tree (specifically, the RST), and there must be a one-to-one correspondence between leaf nodes in this tree and nodes in the input CFG. This constraint implies that if two regions $r1$ and $r2$ in the RST have a non-empty intersection, then it must be the case that either $r1 \subseteq r2$ or $r2 \subseteq r1$.

2. **Proper Containment** Each R-node must have at least one child in the RST that is an L-node. This implies that the region corresponding to a non-root R-node r must be properly contained within the region of its parent's node *parent*(r) (because it will not contain at least one L-node that is a child of *parent*(r)). Another consequence of this constraint is that there can be at most as many region nodes as there are nodes in the input CFG.
3. **Single-entry Regions** Each region must be a single-entry subgraph of the input CFG. In the (rare) event that the input CFG contains an irreducible subgraph (a strongly connected subgraph with multiple entries), then the entire irreducible subgraph must be included in a containing single-entry region.

Thus, there may be several legal RST's for the same input CFG (though they will all have the same set of L-nodes). A trivial RST is a single-level tree consisting of a single R-node that is the parent of all the L-nodes. This is the coarsest possible partitioning into regions because there is only a single region that includes the entire input method. In this paper, we evaluate the approach of making each *single-entry loop* into a *separate* region (see section 2.2), but this approach can be used with any region structure that satisfies the three legality constraints listed above (e.g. an inlined method body can be a region, or multi-level nested loops). However, as we will see, finer-grained decomposition of regions will be useful for improved incremental performance.

An R-node serves as a useful anchor for all information related to the region corresponding to the R-node. All of the region local data structures are stored in the R-node for the region, including references to the Region Control Flow Graph (RCFG), and the Region-SSA built upon the RCFG.

```
    Java Source
x = 1;
if (y > M)
  for (int i=1; i<N;  i++)
  {
     if (i==M) break
     y = sqrt(i);
     arr[i] = arr[i-1]/y;
  }
y = x + 1;
```

```
     Intermediate Representation
S1:  x = 1;
S2:  if (y > M) goto S12;
S3:  i = 1;
S4:  if (i >= M) goto S12;
S5:    if (i == M) goto S12;
S6:    y = invoke sqrt(i);
S7:    $d0 = arr[i-1];
S8:    $d1 = $d0 / y;
S9:    arr[i] = $d1;
S10:   i = i + 1;
S11:   goto S4;
S12: y = x + 1;
```

Fig. 1. Example Program and its Intermediate Representation

As an example, Figure 1 shows a sample Java program and its intermediate representation[1] (right side). The original (flat) control flow graph (corresponding to IR ex-

[1] The IR used in this paper is the PIR in the HJ compiler which is based on Soot's JIMPLE IR [23]. Details will be introduced in Section 4.

ample) is shown in Figure 2. This CFG depicts the following: $S4$ as performs the test of index variable i for the loop; $S10$ as performing the increment of index variable i in each iteration (following a standard translation approach for for-loops); and the edge from $S11$ to $S4$ is the loop back-edge. Let us assume that the compiler selects a two-level hierarchical region partition of the CFG in which $R2 = \{S4, S5, S6, S7, S8, S9, S10, S11\}$ is the inner region and $R1 = \{S1, S2, S3, R2, S12\}$ is the outer region. The RST for this region partition is shown in Figure 3.

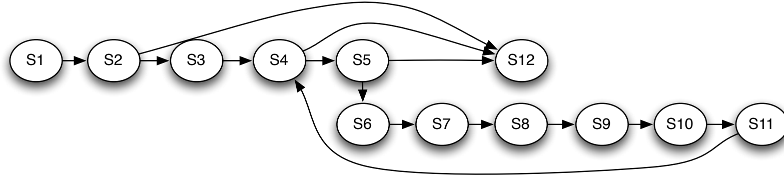

Fig. 2. Original Control Flow Graph

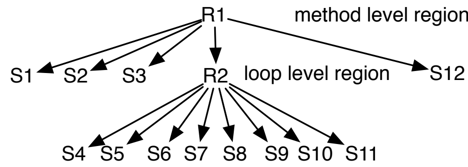

Fig. 3. Region Structure Tree

Figure 4 presents the algorithm for constructing the RST for a given hierarchical region partition, $R(G)$, of control flow graph G. The algorithm's time and space complexity is linear in the size of $R(G)$, and hence linear in the size of G.

For annotating a region at the IR level, we add two statement labels (pseudo-instructions) into the IR code list — *LoopRegionEntry* and *LoopRegionExit*. Figure 5 shows the IR code with these new region labels (see $S4$ and $S13$). The *LoopRegionEntry* statement contains two sets, *Use* and *Def*, which maintain the *SummaryReferences* of current region $R2$ (the detail of *SummaryReferences* will be discussed in next section). The *Use* and *Def* sets for array variables can be made more precise by using Array SSA form [15] instead of SSA form.

2.2 Region Control Flow Graph

For each R-node, R, in the RST, we have a region-level control flow graph, $RCFG(R)$, that defines the control flow for R's immediate children in the RST (the immediate children may be L-nodes or R-nodes). $RCFG(R)$ must contain a node corresponding

ProcessRegionPartition (R(G), Parent) /* Let $R(G) = \{G_1, ..., G_k\}$ */
for i := 1 to k do

if $|G_i| = 1$ then
/* Let $G_i = \{n_j\}$ where n_j is a node in the CFG */
1. L := Create an RST leaf node corresponding to n_j
2. Make L a child of Parent

else /* Let $R(G_i)$ be the region partition for G_i */
1. T := create an RST region node corresponding to G_i
2. Make T a child of parent
3. Call ProcessRegionPartition($R(G_i), T$) /* recursive call */

end if

end for
end procedure;

begin /* main */

1. Root := create root node of RST, corresponding to G /* G is the control flow graph of input program */
2. ProcessRegionPartition (R(G), Root)

end;

Fig. 4. Procedure ProcessRegionPartition

to each node that is a child of R. $RCFG(R)$ also contains two pseudo nodes: START and EXIT. The pseudo nodes have the following interpretations:

- The START node is the destination of all region entry branches. Since R must be a single-entry region, there is no loss in precision in using a single START node. All LCFG edges from START have the same destination: the region's entry node.
- The EXIT node is the target of all region exit branches.

In the example shown in Figure 5, the START and EXIT nodes correspond to statements $S4$ and $S13$.

An edge e from X to Y within $RCFG(R)$ represents control flow in region R. If edge e is a conditional branch (i.e. if there are multiple outgoing edges from X in $RCFG(R)$), then it also carries a *label* of the form (S, C) identifying condition C in IR statement S as the *branch condition* that enabled execution to flow along e. See $(S5, T)$ in Figure 6 as an example. (Other labels have been omitted so as to reduce clutter.)

For each control flow exit from region R, an edge is inserted in R's RCFG with target EXIT, and another edge is inserted in the RCFG of R's RST parent from R to the exit destination. (If multiple regions are being exited, then additional outer-region edges need to be inserted in enclosing regions).

As an example, Figure 6 (corresponding to the IR example shown in Figure 5) shows the RCFG's for regions $R1$ and $R2$ from the RST in Figure 3. Note that $R2$ appears as a node in $RCFG(R1)$. In addition, note that the *break* statement is modeled as a premature loop exit edge from $S5$ in $RCFG(R2)$ with destination = EXIT; there is also an

```
S1:  x = 1;
S2:  if (y > M) goto S12;
S3:    i = 1;
S4:    LoopRegionEntry:
          Use (i, arr, y) Def (i, y)
S5:    if (i >= M) goto S13;
S6:      if (i == M) goto S13;
S7:      y = invoke sqrt(i);
S8:      $d0 = arr[i-1];
S9:      $d1 = $d0 / y;
S10:     arr[i] = $d1;
S11:     i = i + 1;
S12:   goto S5;
S13:   LoopRegionExit
S14: y = x + 1;
```

Fig. 5. RSG based IR Example

edge in $RCFG(R1)$ from node $R2$ to node $S14$. These two edges in $RCFG(R2)$ and $RCFG(R1)$ represent continuation of the same control flow through multiple levels of the RST when the *if* condition in IR statement $S5$ evaluates to true.

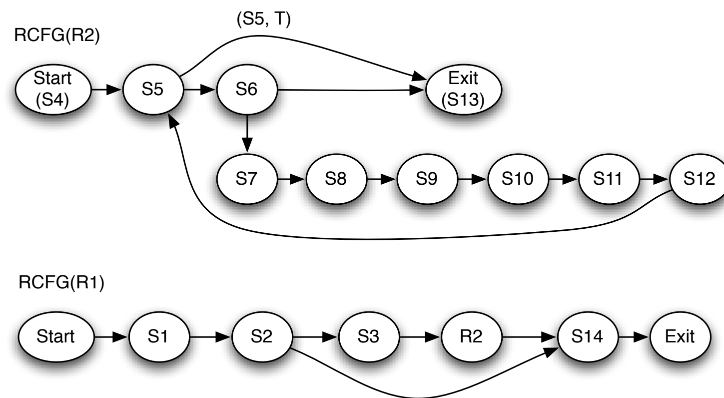

Fig. 6. Region Control Flow Graph

Figure 7 presents the algorithm for constructing the RCFG's for a given RST. The procedure *InitRCFG* is the algorithm's entry which starts from the root node of the given RST, and builds the RCFG for each R-node recursively. The RCFG contains $k+2$ nodes, where $k =$ number of children of the R-node.

A tricky part of the algorithm is in propagating control flow edges to ancestor regions. If T_i and T_j are siblings in the RST, then exactly one edge is created in their parent's RCFG. Otherwise, additional edges are inserted in all ancestor RCFG's up to

Procedure InitRCFG(R)

1. /* Let R's RST children be $T_1 ... T_k$ */
   Create RCFG(R), initialized with $k + 2$ nodes and zero edges. k of the RCFG nodes correspond to $T_1, ..., T_k$. The remaining two nodes are special: START, EXIT.
2. for each region node R_s in R's sub node list
   call InitRCFG(R_s);
   end for
3. call CreateRCFG(R, RCFG(R));

end procedure

Procedure CreateRCFG(R, RCFG(R))

1. create the normal CFG edges among all of the nodes in region R, and add them into RCFG(R);
2. for each edge $e_i = < T_i, T_j >$ whose destination node T_j is outside of R
   (a) LCA = RST least common ancestor of T_i and T_j;
   (b) $temp_i = T_i$;
   while (RST parent of $temp_i \neq LCA$) do
      i. Create an edge from $temp_i$ to EXIT in the RCFG for $temp_i$'s RST parent;
      ii. $temp_i =$ RST parent of $temp_i$;
   end while
   (c) $temp_j = T_j$;
   while (RST parent of $temp_j \neq LCA$) do
      i. Create an edge from START to $temp_j$ in $temp_j$'s RST parent;
      ii. $temp_j$ = RST parent of $temp_j$;
   end while
   (d) Create an edge from $temp_i$ to $temp_j$ in LCA's RCFG;
   end for

end procedure.

Fig. 7. Procedure InitRCFG and CreateRCFG

and including $LCA(T_i, T_j)$. Figure 7 outlines the steps involved in creating these multiple RCFG edges for a single edge.

Each R-node R maintain a *Summary Use* set and *Summary Def* set (shown in Figure 5) which includes *summary references* for use by its parent region R_p as follows:

– If R contains at least one def of a variable V, then include a single *summary def* of variable V in R's *Summary Def* set.
– If R includes at least one use of variable V, then include a single *summary use* of variable V in R's *Summary Use* set.

Summary references can be initially computed by a bottom-up traversal of the RST, and can then be updated incrementally. More precise summaries can be obtained by suppressing the inclusion of a summary use of variable V in R when there is no *upwards expose* and *downwards expose* [1] of V in R_p, and by suppressing the inclusion of a

summary def of variable V in R_p when V can be renamed [6] to a local/private variable in region R.

3 Region based Single Static Assignment Form

This section presents the the Region-SSA form, including algorithms for initialization, transformation out of Region-SSA form, and for incremental updates.

3.1 Region-SSA Construction

A Region-SSA form structure is built for each RCFG. As mentioned in Section 2.2, the pseudo-instructions *LoopRegionEntry* and *LoopRegionExit* are used to annotate the R-node in its parent region. To enable the interaction between parent region R_p and child region R_c, each child region R_c stores summary *Use/Def* sets. These sets enable R_c to be treated uniformly like other statements in R_p's scope (except that R_c may potentially contain more defs and uses than normal statements). R_c also maintains a *Use* map and a *Def* map which maintain the map between the *Summary References* and corresponding local variables in R_c.

Figure 8 presents the steps for constructing Region-SSA form for a given RST. The procedure *InitRCFG* is the algorithms entry which starts from the root node of the given RST. The building process is accomplished by a bottom up traversal of the RST that ensures that the child R-node in the RST is processed prior to its parent. For each R-node, the Region-SSA construction algorithm (extended from standard SSA) is applied. Within a region, we use the standard SSA construction algorithm [8], so we omit the details of building the dominator tree, dominance frontiers and variable renaming. Since we perform variable renaming locally in each region, the same variable may have multiple rename stacks in different regions. To avoid any name duplication during renaming process, each region is assigned a global region ID, and this ID is mangled into the new variable name. For example, given a variable *x*, it new name will be of the form: *x_region ID_assignment count*.

The extension of standard SSA comes from Φ-function creation. To maintain the correct use/def chain information, every *summary reference* variable should be treated as a definition at the entry of the region (see step 4 in procedure *RSSAConstruct*). Another extension is building the variable map between the *summary reference* variables and their corresponding region local variables (see step 6 and 7 in procedure *RSSAConstruct*).

Figure 9 and 10 show the translated Region-SSA form based on the example shown in Figure 5. To demonstrate the region, we split the IR code into two parts corresponding to region *R1* (method level region) and *R2* (loop level region). The *Use/Def* variable maps are listed after the region entry label. Compared with standard SSA form, Region-SSA also maintains a variable mapping between the parent and child regions. Thus, Region-SSA construction for the parent region does not need to go through the child region's code.

InitRSSA(R) /* Initialize the Region-SSA for a given R-node and all of its sub R-nodes*/

1. for each region node R_i in R's sub node list
   - call InitRSSA(R_i)

   end for
2. call RSSAConstruct(R);

end /* InitRSSA */

RSSAConstruct(R)

1. initialize RCFG(R) and call CreateRCFG(R, RCFG(R)) to build it;
2. build R's dominator tree: $DT(R)$
3. build R's dominance frontier: $DF(R)$
4. insert Φ-functions for R based on that each variable in Use/Def set is treated as an implicit definition at the entry of the region
5. rename variables in R
6. for each variable u_p in R's Use set
   - get u_p's corresponding variable u_c in u_p's name stack S in R, u_c should satisifies that its subscript has the minimum assignment count in S;
   - add the map pair: $u_c \mapsto u_p$ into R's Use map ;

   end for
7. for each variable d_p in R's Def set
   - get d_p's corresponding variable d_c in d_p's name stack S in R, d_c should satisifies that its subscript has the maximum assignment count among those varaibles that are in same rename stack S and their corresponding assignment statements dominate the EXIT code;
   - add the map pair: $d_c \mapsto d_p$ into R's Def map;

   end for

end procedure /* RSSAConstruct */

Fig. 8. Region-SSA Construction Algorithm

```
S1:  x_1_0 = 1;
S2:  if y_1_0 <= M goto S17;
S3:  i_1_0 = 1;
S4:  Loop Region Entry
     Use Set (y_1_0, arr_1_0, i_1_0)
     Def Set(y_1_1, i_1_1)
S16: Loop Region Exit
S17: y_1_2 = Phi(y_1_0 #S2, y_1_1 #S16);
S18: y_1_3 = x_1_1 + 1;
```

Fig. 9. Region-SSA(R1)

```
S4:  Loop Region Entry:
     Use Set (y_1_0, arr_1_0, i_1_0)
     Def Set (y_1_1, i_1_1)
     Use Map: (y_2_0 ↦ y_1_0, arr_2_0 ↦ arr_1_0, i_2_0 ↦ i_1_0)
     Def Map: (y_2_1 ↦ y_1_1, i_2_1 ↦ i_1_1)
  S5:    i_2_1 = Phi(i_2_0 #S4, i_2_2 #S15);
  S6:    y_2_1 = Phi(y_2_0 #S4, y_2_2 #S15);
  S7:    if i_2_1 >= M goto S16;
  S8:      if i_2_1 == M goto S16;
  S9:      y_2_2 = invoke sqrt(i_2_1);
  S10:     $i0 = i_2_1 - 1;
  S11:     $i1 = arr_2_0[$i0];
  S12:     $i2 = $i1 / y_2_2;
  S13:     arr_2_0[i_2_1] = $i2;
  S14:     i_2_2 = i_2_1 + 1;
  S15:   goto S5;
S16: Loop Region Exit
```

Fig. 10. Region-SSA(R2)

3.2 Out-of-Region-SSA Transformation

Figure 11 shows the algorithm for transforming out of Region-SSA. The Φ function elimination is same as the standard SSA algorithm [8], so we omit details here. To maintain the correctness of the value exchange between an R-node *R* and its parent, a prologue and an epilogue need to be inserted into *R*'s region. For prologue creation, the assignment operations are created from all of the variables in *R*'s *Use* set to their corresponding variables in *R*'s *Use* map, and inserted in front of the region entry node. Similarly, all of the variables in *R*'s *Def* set should be assigned by their corresponding variables in *R*'s *Def* map for creating the epilogue.

Figure 12 shows the translated IR code after Out-of-Region-SSA process (the prologue code are: *S5*, *S6*, *S7* and epilogue code are: *S20*, *S21*). After the Out-of-Region-SSA process, the *Use*/*Def* map are eliminated. As the out Region-SSA introduces some redundant definition operations (i.e. the statements in prologue and epilogues), some post Region-SSA optimizations (e.g. copy propagation, coalesces) are necessary.

3.3 Incremental Update Algorithm

When a program transformation modifies a region, the Region-SSA form must be reconstructed to maintain precision and correctness. Here we identify the regions that need to be involved in the reconstruction.

Given a region R, a compiler transformation can insert/delete statements and insert/delete uses/defs of variables. For region-local (private) variables, only the current region needs to be reconstructed. For inserted/deleted variables in ancestor regions, we enumerate the scenarios and rules for identifying the regions that need to be reconstructed as follows (the identified regions are put into a *reconstruction list*):

OutRSSA(R)

1. get the list of R's RST children nodes: L_{rs}
2. for each region node T_i in L_{rs}
   - call OutRSSA(T_i)

   end for
3. eliminate the Φ-functions in local region
4. for each variable u_c in R's Use set
   (a) get its corresponding variable u_p in parent region from R's Use map
   (b) generate assignment instruction: $u_c \leftarrow u_p$ and insert this statement after the START node

   end for
5. for each variable d_c in R's Def set
   (a) get its corresponding variable d_p in parent region from R's Def map
   (b) generate assignment instruction: $d_p \leftarrow d_c$ and insert this statement before the EXIT node

   end for

end procedure; /* OutRSSA */

Fig. 11. Algorithm for Out-of-Region-SSA Transformation

- Insert use variable u_p (u_p is a variable in parent region R_p and $u_p \notin R$'s Use set):
  1. add u_p into the R's Use set;
  2. create a corresponding variable u_c for using in R;
  3. add the pair $u_c \mapsto u_p$ into R's Use map;
  4. add R into the *reconstruction list*;
- Remove a use variable u_p (u_p is a variable in parent region and $u_p \in R$'s Use set):
  1. remove u_p from Use set;
  2. remove u_p related pair from Use map;
  3. add R into the *reconstruction list*;
- Insert a def variable d_p at statement S (i.e. d_p is a variable in parent region R_p and $d_a \notin R$'s Def set):
  1. add d_p into R's Def set;
  2. create a corresponding variable d_c for using in R;
  3. add $d_c \mapsto d_p$ into R's Def map;
  4. if $d_p \notin Use$ set and S does not dominate R's entry node, then
     (a) add d_p into R's Use set
     (b) create a corresponding variable d_{init} and add $d_{init} \mapsto d_p$ into R's Use map;
  5. add both R and R_p into the *reconstruction list*;
- Remove a def variable: d_p (i.e. d_p is a variable in parent region R_p and $d_p \in R$'s Def set):
  1. remove the variable from the Def set;
  2. remove this variable related pair from Def map;
  3. add both R and R_p into the *reconstruction list*;

```
S1:  x_1_0 = 1;
S2:  if y_1_0 <= M goto S23;
S3:  i_1_0 = 1;
S4:  Loop Region Entry:
     Use Set (y_1_0, arr_1_0, i_1_0)
     Def Set (y_1_1, i_1_1)
S5:    y_2_0 = y_1_0;
S6:    i_2_0 = i_1_0;
S7:    arr_2_0 = arr_1_0;
S7:    i_2_1 = i_2_0;
S8:    y_2_1 = y_2_0;
S9:    if i_2_1 >= M goto S22;
S10:     if i_2_1 == M goto S22;
S11:     y_2_2 = invoke sqrt(i_2_1);
S12:     $i0 = i_2_1 - 1;
S13:     $i1 = arr_2_0[$i0];
S14:     $i2 = $i1 / y_2_2;
S15:     arr_2_0[i_2_1] = $i2;
S16:     i_2_2 = i_2_1 + 1;
S17      y_2_1 = y_2_2;
S18      i_2_1 = i_2_2;
S19:   goto S9;
S20:   y_1_1 = y_2_1;
S21:   i_1_1 = i_2_1;
S22: Loop Region Exit
S23: y_1_2 = y_1_1;
S24: y_1_3 = x_1_1 + 1;
```

Fig. 12. IR after Out-of-Region-SSA Transformation

The scenarios and rules list above handle the updating between the parent and child regions. These rules can also be applied recursively for handling the ancester/child case. Given those regions in the *reconstruction list*, *RSSAConstruct* is called to reconstruct Region-SSA form.

When the program transformations modified the regions (i.e. shrinking or extending the region), the modified region and its parent region need to be reconstructed. When the transformations add or delete regions in RST, the added or deleted regions and their parent region need to be reconstructed.

4 Experimental Results

In this section, we present experimental results for the Region-SSA approach. We first summarize the experimental setup. Then we use standard loop transformations (unrolling and interchange) as examples to demonstrate the benefits of Region-SSA form by comparing the compile-time for recomputing SSA form in our approach with that of recomputing standard SSA form exhaustively. As mentioned earlier, the Region-

SSA approach can also be used with other region decompositions (beyond loops), even though the results in this section are for loop transformations.

4.1 Experimental Setup

We implemented the Region-SSA support in the Habanero-Java (HJ) compiler, which translates HJ source code to Java bytecode. Figure 13 presents the basic structure of the HJ compiler.

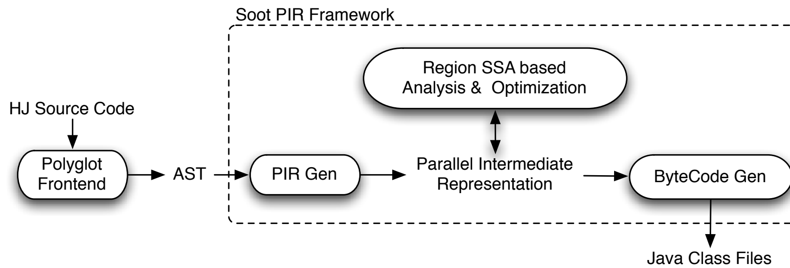

Fig. 13. Compilation Framework for Region SSA

We modified the Polyglot-based front-end for X10 [26] to emit a new Parallel Intermediate Representation (PIR) extension to the Jimple intermediate representation in the SOOT bytecode analysis and transformation framework [23]. In addition to standard Java operators, the PIR includes explicit constructs for parallel operations [22] such as async and finish. However, no HJ parallel constructs were used for the results reported in this paper.

We choose the Java version of the NAS Parallel Benchmarks (NPB) as our test suite. NPB is a high performance computation benchmark suite which contains multiple levels of nested loops. Table 1 summarizes the RSG statistics for Java versions of six NPB benchmarks[2].

Benchmark	FT	IS	CG	MG	SP	BT
Method Region	17	12	17	27	30	24
Loop Region	27	15	51	80	202	157

Table 1. The Number of Regions Per Benchmark.

All results were obtained on a Quad-Core Intel E7330 Xeon processor running at 2.40GHz with 8GB memory. The JDK used was Sun JDK 1.6.0 64-bit, and the heap size was set to 4GB.

[2] LU was omitted because Soot was unable to process the Java version of LU.

4.2 Loop Unrolling

Loop unrolling [25] replicates a loop body multiple times, thereby reducing the number of branch instructions executed and enabling the compiler to perform aggressive optimizations. For the results reported in this paper, we chose inner-most loops that contain less than 500 IR instructions as the targets for our unrolling experiments.

Table 2 shows the benchmark methods' compile-time [3] obtained from initializing the standard SSA/Region-SSA (see the columns "init"), and reconstructing standard SSA/Region-SSA after unrolling the inner most loop by the unroll factor 2 and 4 respectively (see the columns "trans"). For standard SSA, reconstruction rebuilds SSA form exhaustively for the given method. For Region-SSA, only updated regions are reconstructed. Finally, we give the speedups, including the ratio from initializing standard SSA to Region-SSA (see column "Init Speedup"); and the ratio from reconstructing standard SSA to Region-SSA (i.e. demand-driven) after unrolling by factor 2 and 4 (see columns "Trans Speedup (factor 2)" and "Trans Speedup (factor 4)").

As each R-node of the method's RST has its own CFG, dominance frontier, dominator tree and other SSA form related data structures, this introduces more initialization overhead than standard SSA which only need to maintain one instance for each type of data structure. So the initialization of Region-SSA spent more compile-time than standard SSA in most of the methods. On another hand, the Region-SSA simplifies the control flow graph for each region and reduces the complexity of SSA form computation. That is why some of methods' Region-SSA initialization is more efficient than standard SSA.

As the reconstruction of Region-SSA is demand-driven, it is more efficient than the standard SSA which need recompute SSA form exhaustively. The speedup varies from $1.18\times$ to $35.81\times$. The speedup is small for smaller methods such as IS.partial_verify and large for larger methods such as BT.z_solve. The fact that the biggest speedups were observed for the largest methods is an important testimonial to the value of Region-SSA form.

4.3 Loop Interchange

Loop interchange [25] is the process of exchanging the order of two loop iteration variables. In this paper, we setup such constraints for selecting the loops that are the targets for interchanging:

- the two inner most perfectly nested loops.
- the loop bounds must be compile-time constant or loop invariants for both loops involved in interchange.
- the array index must be analyzable, i.e. the array index must be an affine expression with factors which are compile-time constants; and there should be no data dependence across loop iterations.

[3] Due to the space limitation, we only chose those methods whose compile-time for constructing standard SSA is larger than 10 msec.

	Function Name	Num of Unrolled Loops	Standard SSA (msec)			Region SSA (msec)			Init Speedup	Trans Speedup (factor 2)	Trans Speedup (factor 4)
			Init	Trans (factor 2)	Trans (factor 4)	Init	Trans (factor 2)	Trans (factor 4)			
FT	fftXYZ	5	56	65	139	63	42	81	0.89×	1.55×	1.72×
	appft_serial	1	58	94	262	61	78	166	0.95×	1.21×	1.58×
IS	runBenchMark	2	15	30	38	23	7	11	0.65×	4.29×	3.46×
	full_verify	1	11	12	16	15	4	9	0.73×	3×	1.78×
	partial_verify	1	15	47	59	17	32	50	0.88×	1.47×	1.18×
CG	runBenchMark	11	91	185	269	88	79	155	1.03×	2.34×	1.74×
	makea	3	14	64	77	20	31	54	0.7×	2.07×	1.43×
	sparse	8	20	93	102	23	20	41	0.87×	4.65×	2.49×
	conj_grad	9	24	61	98	28	23	38	0.86×	2.65×	2.58×
MG	setup	4	22	39	70	26	15	27	0.85×	2.6×	2.59×
	zran3	7	70	199	456	63	105	182	1.11×	1.90×	2.51×
	bubble	2	17	67	133	17	44	77	1×	1.52×	1.73×
	resid	2	21	77	156	26	41	74	0.81×	1.88×	2.11×
	mg3p	2	15	37	71	15	21	42	1×	1.76×	1.69×
	mg3pMaster	2	14	21	45	14	12	32	1×	1.75×	1.41×
	rprj3	2	34	180	470	28	64	136	1.21×	2.81×	3.46×
	interp	13	61	615	1259	83	250	431	0.74×	2.46×	2.92×
	psinv	2	10	49	118	14	20	41	0.71×	2.45×	2.88×
	runBenchmark	1	65	57	91	89	33	52	0.73×	1.73×	1.75×
SP	printTimers	2	19	57	83	17	41	67	1.12×	1.39×	1.24×
	error_norm	3	12	17	25	18	8	14	0.67×	2.13×	1.79×
	exact_rhs	20	832	15074	49000	782	4876	9928	1.06×	3.09×	4.94×
	compute_rhs	18	1649	5002	8490	1724	1730	3030	0.96×	2.89×	2.80×
	txinvr	1	15	186	761	15	90	285	1×	2.07×	2.67×
	tzetar	1	14	181	1133	15	90	189	0.93×	2.01×	6.0×
	x_solve	13	1044	6583	17688	1388	2299	5260	0.75×	2.86×	3.36×
	y_solve	13	1010	6559	18677	1348	2347	4602	0.75×	2.80×	4.06×
	z_solve	14	588	3672	9353	668	1346	2508	0.88×	2.73×	3.73×
BT	printTimers	2	22	57	85	24	41	56	0.92×	1.39×	1.52×
	rhs_norm	3	18	18	22	17	8	11	1.06×	2.25×	2×
	error_norm	3	20	18	26	20	9	13	1×	2×	2×
	verify	3	20	22	26	19	6	10	1.05×	3.67×	2.6×
	exact_rhs	20	803	14678	55026	812	5060	11283	0.99×	2.90×	4.88×
	x_solve	2	2154	2678	3242	2287	80	137	0.94×	33.48×	23.66×
	compute_rhs	18	2132	5151	8270	1789	1700	4163	1.19×	3.0×	1.99×
	y_solve	2	2077	2683	2686	2290	79	136	0.91×	33.96×	19.75×
	z_solve	2	2101	2578	2675	2261	72	122	0.93×	35.81×	21.93×

Table 2. Compile-time(msec) Result for Loop Unrolling with Factor 2 and 4.

As loop interchange does not add or delete any reference and definition for the variable that are outside of current loops, the Region-SSA reconstruction process only need to update the two loop regions that are involved in the interchange. Table 3 shows the experimental result for measuring compile-time for building each benchmark method. Like the loop unrolling case, this table shows the compile-time for the two SSA mechanism and the speedup ratios for both standard SSA form/Region-SSA form initialization and reconstruction after loop interchange.

Based on the constraint listed above, there are less loops that can be involved in interchange. Also those interchanged loops use to have small size. So we got better speedup for SSA reconstruction (e.g. from 1.33× to 206.54×).

	Function Name	Num of Interchanged Loops	Standard SSA (msec)		Region SSA (msec)		Init Speedup	Trans Speedup
			Init	Trans	Init	Trans		
FT	fftXYZ	8	56	73	63	19	0.89×	3.84×
	appft_serial	2	58	95	61	22	0.95×	4.32×
CG	runBenchMark	2	91	95	88	4	1.03×	23.75×
	conj_grad	4	24	28	28	6	0.86×	4.67×
	makea	2	14	27	20	10	0.7×	2.7×
MG	setup	4	22	23	26	6	0.85×	3.83×
	zran3	6	70	84	63	4	1.11×	21×
SP	error_norm	2	12	22	18	6	0.67×	3.67×
	exact_rhs	10	918	1414	782	92	1.17×	15.37×
	compute_rhs	32	1649	3335	1724	1764	0.95×	1.89×
	txinvr	2	15	20	15	15	1×	1.33×
	tzetar	2	14	20	15	15	0.93×	1.33×
	x_solve	2	1044	1700	1388	223	0.75×	7.62×
	y_solve	2	1010	1847	1344	223	0.76×	8.28×
	z_solve	2	588	1064	668	66	0.88×	16.12×
BT	rhs_norm	2	18	21	17	5	1.06×	4.2×
	error_norm	2	20	12	20	3	1×	4×
	exact_rhs	10	803	1384	812	89	0.99×	15.55 ×
	x_solve	2	2154	2446	2287	16	0.94×	152.88×
	compute_rhs	18	2132	3386	1789	2082	1.19×	1.63×
	y_solve	2	2077	2685	2290	15	0.91×	179×
	z_solve	2	2101	2685	2261	13	0.93×	206.54×

Table 3. Compile-time(msec) for Standard SSA and Region SSA in Loop Interchange.

5 Related Work

Our source of inspiration for the region based hierarchical program partitioning originated from the Forward Control Dependence Graph (FCDG) used in the PTRAN system to represent interval structure and statement parallelism [11, 9, 19]. The FCDG is a variant of the PDG in which the program's loop structure is made evident by control

dependences that are derived from pseudo-control-flow edges connecting interval preheader and post-exit nodes. There are two limitations in using the FCDG representation for incremental reanalysis:

1. There is no known way to incrementally update the FCDG (and accompanying data dependence edges) after a small program transformation.
2. The FCDG is not well-defined for a program with an irreducible control flow graph; at the very least, irreducibility causes the FCDG to be cyclic whereas all the published algorithms that use the FCDG assume that it is acyclic.

In our approach, the Region Structure Graph remedies the above problems as follows:

1. The region node builds all of CFG related data structure locally, e.g. RCFG, dominator tree based on the RCFG.
2. An irreducible region can be isolated from the other regions in the RST which remain eligible for all transformations and optimizations.

The Hierarchical Structured Control-flow Graph (HSCG) [13] and the Hierarchical Task Graph (HTG) [18] were proposed as hierarchical program representations with a loop hierarchy at the outer level and individual loop control flow graphs at the inner level. Whereas this hierarchical structure is similar to that of the RSG, the HSCG and the HTG are only defined for structured programs and also suffer from other limitations of the FCDG.

Some previous work had been done for updating SSA from incrementally. Cytron and Gershbein [10] present techniques for incrementally building SSA form to accommodate $mayalias$. This approach only works for the inserted special definitions, called $may - defs$, generated by may-aliases. The incrementality is limited in the program whose structure and contents remain constant.

In [5], Choi, Sarkar and Schonberg presented the algorithm for recomputing SSA form incrementally. The algorithm handles the cases that insert/delete of a use/definition of a variable arbitrarily and update a single $interval$ arbitrarily. The $interval$ here is actually a region of CFG. To update this $interval$, the local dominator tree, and dominator frontier need to be built, then insert Φ function and perform rename. These steps are similar to procedure $RSSAConstruct$ discussed in Section 3. But the $interval$ is based on a procedural level CFG, and there is no hierarchical mechanism for updating of CFG locally.

They also demonstrated the customized SSA-update algorithm for loop transformations (e.g. loop fusion and loop interchange). In this approach, all of the SSA related compiler transformation algorithm need to be aware the updating rules to maintain the correctness of the SSA form. This increases the complexity, when user designs the transformation algorithms.

Compared with the related work listed above, the hierarchical based Regions-SSA leverages the efficiency of the hierarchy based control flow structure, also provides a general. As each region's local SSA form is same as standard SSA form, the SSA based compiler transformation algorithm can be supported transparently.

6 Conclusion and Future Work

In this paper, we addressed the problem of *incrementally rebuilding SSA form* i.e. of efficiently updating SSA form after program transformations are performed. We introduced Region SSA form, a hierarchical SSA form based on Region Structure Graph, a new compiler internal representation that maintains control and data information for a hierarchical region decomposition of the program. A local SSA form is built for each region. In addition to define all the components of the Region-SSA and RSG, we described how can the Region SSA be constructed/out for a given hierarchical region partition and how to update it incrementally. The incremental updating is region-based demand-driven policy, which means that only those updated regions need to reconstruct their Region-SSA form. Finally, we presented some loop transformations based on Region-SSA and related experimental results based on the compile-time got from incremental recomputing Region-SSA and recomputing standard SSA form exhaustively. By comparing the compile-time, the demand-driven recomputing Region-SSA shows significant advantage. For loop unrolling transformations, compile-time speedups up to 35.8$\times$ were observed using Region-SSA form relative to standard SSA form. For loop interchange transformations, compile-time speedups up to 205.6$\times$ were observed. All of these experimental results show that the Region-SSA is is an ideal infrastructure for building incremental program analysis and optimization system.

For future work, we are extending the current scalar Region-SSA to support Array SSA [15], and applying this model (i.e. RSG and Region-SSA) as the basic infrastructure for a new compilation framework which performs sophisticated incremental program analysis and optimizations (such as loop transformations, scalar replacement, procedure inlining, etc.).

References

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.
2. F. E. Allen and J. Cocke. A Program Data Flow Analysis Procedure. *Communications of the ACM*, 19(3):137–147, March 1976.
3. John R. Allen. *Dependence Analysis for Subscripted Variables and its Application to Program Transformation*. PhD thesis, Rice University, 1983.
4. John Cavazos, Grigori Fursin, Felix V. Agakov, Edwin V. Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO*, pages 185–197, 2007.
5. Jong-Deok Choi, Vivek Sarkar, and Edith Schonberg. Incremental computation of static single assignment form. In *CC*, pages 223–237, 1996.
6. Ron Cytron and Jeanne Ferrante. What's in a Name? Or the Value of Renaming for Parallelism Detection and Storage Allocation. *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, August 1987.
7. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An Efficient Method for Computing Static Single Assignment Form. *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, January 1989.
8. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

9. Ron Cytron, Jeanne Ferrante, and Vivek Sarkar. Experiences Using Control Dependence in PTRAN. *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, August 1989. In Languages and Compilers for Parallel Computing, edited by D. Gelernter, A. Nicolau, and D. Padua, MIT Press, 1990 (pages 186-212).
10. Ron Cytron and Reid Gershbein. Efficient accommodation of may-alias information in ssa form. *SIGPLAN Not.*, 28(6):36–45, 1993.
11. Ron Cytron, Michael Hind, and Wilson Hsieh. Automatic Generation of DAG Parallelism. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland, Oregon*, 24(7):54–68, June 1989.
12. J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
13. Francois Irigoin, Pierre Jouvelot, and Remi Triolet. Semantical Interprocedural Parallelization: An Overview of the PIPS Project. *Proceedings of the ACM 1991 International Conference on Supercomputing*, June 1991.
14. Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in ssa form. *ACM Trans. Program. Lang. Syst.*, 21(3):627–676, 1999.
15. Kathleen Knobe and Vivek Sarkar. Array ssa form and its use in parallelization. In *POPL*, pages 107–120, 1998.
16. L. Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, 17(2):83–93, February 1974.
17. David Parello, Olivier Temam, Albert Cohen, and Jean-Marie Verdun. Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *SC*, page 15, 2004.
18. Constantine Polychronopoulos. The Hierarchical Task Graph and its Use in Auto-Scheduling. *Proceedings of the ACM 1991 International Conference on Supercomputing*, June 1991.
19. Vivek Sarkar. The PTRAN Parallel Programming System. *Parallel Functional Programming Languages and Compilers*, pages 309–391, 1991.
20. Vivek Sarkar and Radhika Thekkath. A General Framework for Iteration-Reordering Loop Transformations. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 175–187, June 1992.
21. J. T. Schwartz and M. Sharir. Tarjan's fast interval finding algorithm. Technical report, Courant Institute, New York University, 1978. SETL Newsletter Number 204.
22. Jun Shirako, Jisheng Zhao, V.Kirshna Nandivada, and Vivek. Sarkar. Chunking parallel loops in the presence of synchronization. In *ICS '09*, New York, NY, USA, 2009. ACM.
23. R. Vallée-Rai et al. Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.
24. Mark Wegman and Ken Zadeck. Constant Propagation with Conditional Branches. *Conf. Rec. Twelfth ACM Symposium on Principles of Programming Languages*, pages 291–299, January 1985.
25. Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.
26. X10 release on SourceForge. http://x10.sf.net.