Register Allocation Sensitive Region Scheduling *

Cindy Norris
norris@cis.udel.edu

Lori. L. Pollock
pollock@cis.udel.edu

Department of Computer and Information Sciences
University of Delaware
Newark, DE, 19716
(302) 831-1953, fax (302) 831-8458

Abstract

Because of the interdependences between instruction scheduling and register allocation, it is not clear which of these two phases should run first. In this paper, we describe how we modified a global *instruction scheduling technique to make it cooperate with a subsequent register allocation phase. In particular, our cooperative global instruction scheduler performs region scheduling transformations on the program dependence graph representation of a program while attempting to prevent an increase in the amount of spill code which will be introduced in the subsequent register allocation phase. Our experimental findings indicate that the cooperative technique does indeed produce more efficient code than non-cooperative global instruction scheduling in programs in which an allocation can not be performed without the insertion of spill code.*

1 Introduction

In order to effectively exploit the fine-grained parallelism in pipelined, superscalar and VLIW machines, various strategies for carefully scheduling instructions at compile time have been developed [22, 33, 18, 31]. A compile-time scheduling phase increases run-time performance by rearranging the code to overlap the execution of low level machine instructions such as memory loads and stores, and integer and floating point operations to hide latencies and reduce possible run-time delays. A scheduler that rearranges code within each basic block in isolation is called a local scheduler, while a scheduler that moves instructions across basic block boundaries by considering the effects of code movement on a global level is called a global scheduler. Register allocation, which allocates registers to values so as to minimize the number of memory accesses at run-time [13, 12, 11, 29, 10, 21, 26], often interferes with the goals of instruction scheduling, creating a dilemma for the compiler writer in deciding which order to perform these two important optimization phases.

Performing register allocation before instruction scheduling may introduce false dependences by the reuse of the same physical register in otherwise independent instructions. This can prevent the instruction scheduler from reordering instructions or scheduling them to be executed in parallel. However, when instruction scheduling is performed before register allocation, the instruction scheduler, in its attempt to reorder instructions to maximize fine-grain parallelism, may lengthen the lifetime of values and thus increase the contention for registers and potential for register spilling.

There have been several strategies developed to introduce some communication of requirements between the local scheduler and the register allocator such that the two phases can cooperate to generate better code [19, 8, 28, 25, 6]. Experimental results from these groups indicate that the cooperative schemes indeed generate more efficient code than a conventional code generator that treats register allocation and instruction scheduling in isolation. This leads one to believe that similar increases in code quality could be attained by developing cooperative strategies between a global scheduler and the register allocator.

In this paper, we describe a strategy for modifying a global instruction scheduling technique known as *region scheduling* [20, 23] to take into consideration the requirements of a subsequent register allocation phase. We report on our experimental findings based on our implementation and comparison of standard region scheduling and our **R**egister **A**llocation **Se**nsitive **R**egion scheduling (RASER). Region scheduling performs scheduling over a program dependence graph (PDG) attempting to create regions in the PDG that contain equal amounts of fine-grain parallelism. RASER performs region scheduling transformations while attempting to prevent an increase in the amount of spill

*This work was partially supported by NSF under grant CCR-9300212.

code which will be introduced in the subsequent register allocation phase. RASER is easily created from modifying an existing region scheduler because it also uses the PDG, and does the same kinds of legality tests. We also introduce a new scheduling technique known as *live value reduction* which can be used to reduce the amount of spill code introduced. Live value reduction is a simple technique which is also easily integrated into region scheduling because it requires information which is readily available in the PDG.

We chose to focus on region scheduling to study the potential for cooperation between a global instruction scheduler and register allocation for several reasons. Foremost, the region scheduling algorithm uses the PDG. This representation is particularly useful in modifying the region scheduler to consider the problem of register allocation since the PDG includes data dependence edges. The same representation can be used by other phases of the compiler as the PDG has been used successfully as the basis for various scalar optimizations [14, 27, 4] as well as for detecting and improving parallelization for vector machines [30, 4], multiple processor machines [32, 3], and architectures that exhibit instruction level parallelism [20, 5, 2]. Although the global scheduling algorithm of Bernstein and Rodeh [5] also operates on the PDG, their approach only achieved modest improvements over local instruction scheduling. Other global scheduling techniques such as trace scheduling [16] and percolation scheduling [1] utilize the control flow graph rather than a program dependence graph to rearrange code. In addition, trace scheduling does not consider the underlying parallel architecture in order to determine how much parallelism should be exploited, requires the programmer to obtain profiling information, and is mainly applicable to scientific codes due to its reliance on execution frequency estimates.

The remainder of this paper is organized as follows. Section 2 gives a more in depth description of previous cooperative strategies between register allocation and global instruction scheduling. Sections 3 and 4 present brief background on the PDG and standard region scheduling. The details of RASER are presented in Section 5, followed by a description of our experimental study in Section 6. We summarize the research and discuss future directions in Section 7.

2 Previous Cooperative Strategies

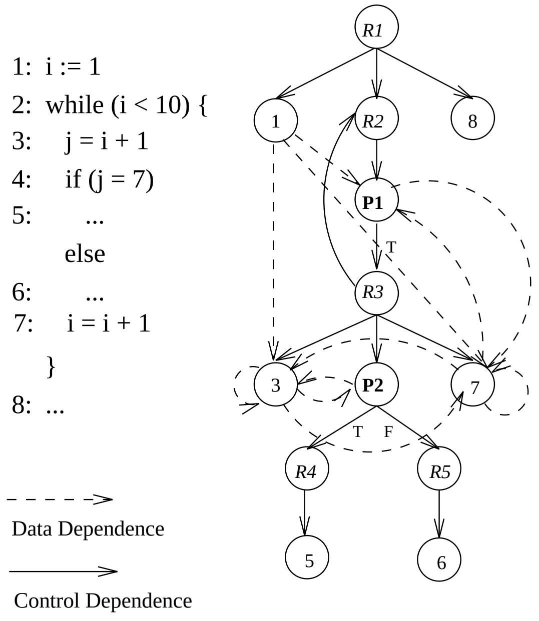
Cooperative strategies can be categorized as those which cooperate with local instruction scheduling and those which cooperate with a global instruction scheduling technique. In this section, we focus on the prior work in conjunction with global instruction scheduling.

Freudenberger [17] developed a technique for integrating register assignment with trace scheduling in the MultiFlow compilers. The scheduler drives the register assignment to assign the values in the heavily used traces to registers. The scheduler takes as many registers as it needs as it schedules the crucial traces first. Information about the register assignment is stored at the entry and exit points of traces, and used to hook up less crucial traces to the earlier scheduled traces, to minimize the amount of data movement code.

Moon and Ebcioglu [24] presented a resource-constrained code scheduling technique for VLIW and superscalar machines. The scheduling algorithm precomputes the set of available operations that are schedulable and can reach the root VLIW instruction. Global code motion is performed by choosing the best operation that can move to a point using the precomputed sets, moving the operation to the point, creating bookkeeping copies for edges that join the path, but are not on the path, and finally updating the available operations information at each basic block on the moving path. A search is made for finding a suitable destination register for an operation that has been identified as movable; however, if a suitable register can not be found among the available registers, the next best movable operation is examined.

Berson, Gupta and Soffa [6] developed a technique called URSA for allocating both functional units and registers in VLIW machines. In attempt to create phases with minimal interaction, register allocation and instruction scheduling are partitioned into a different sequence of phases, namely, the computation of resource requirements and identification of regions with excess requirements, code motion to reduce requirements to the level supported by the target machine, and finally resource assignment. URSA uses a dependence DAG to determine register and functional unit requirements. The dependence DAG is modified to reflect scheduling and allocation decisions by adding sequence edges and load and store instructions to spill registers. URSA relies on a program trace to build the DAG thus making it mainly applicable to scientific applications.

In a later paper [7], Berson, Gupta and Soffa describe the application of this framework to the problem of integrating register allocation with local instruction scheduling, and then with global instruction scheduling. The authors note that the integration of global instruction scheduling is much more complicated than local instruction scheduling under this framework, because benefits of moving instructions must be determined, while there is no choice but to reduce excess resource demands in the local scheduling. The paper mentions that the code motion algorithms must consider the effects of code duplication on the critical paths, and must determine whether the critical path length of the source block is decreased when loads of moved values are inserted in it. The paper does not go into detail on how these problems are handled; no detailed algorithm for resource spackling with global in-



```
1:  i := 1
2:  while (i < 10) {
3:     j = i + 1
4:     if (j = 7)
5:        ...
       else
6:        ...
7:     i = i + 1
    }
8:  ...
```

Figure 1: Program Dependence Graph

struction scheduling is given. Thus far, no experimental results have been obtained to judge the effectiveness of the URSA approach.

3 Program Dependence Graphs

The program dependence graph (PDG) [14] for a program is a directed graph that represents the relevant control and data dependences between statements in the program. The nodes of the graph are statements and predicate expressions that occur in the program. An edge represents either a control dependence or a data dependence among program components. A data dependence edge from x to y denotes the fact that executing y before x could alter the program's semantics because x and y may reference the same memory location with at least one of them writing to that location. A control dependence edge from p to n denotes the fact that predicate p immediately controls the execution of the statement n. The source of a control dependence edge is either the entry node of the program or a predicate. A control dependence edge from a predicate node is labeled with either *true* or *false*, indicating the value of the predicate under which the statement at the sink of the edge will be executed.

Special nodes called region nodes are inserted into the graph to summarize the set of control conditions for a node and to group all of the nodes that are executed under the same control conditions together as successors of the same region node. For a given node n, each subset of its control dependences that is common with control dependences of another node, say m, is factored out, and a region node R is created to represent this control dependence subset. The common control dependence edges to n and m are replaced by edges $R \rightarrow n$ and $R \rightarrow m$. Each region node represents a set of control conditions, and after region nodes are inserted, each predicate node has at most one *true* outgoing edge and one *false* outgoing edge.

Figure 1 shows a program segment and its PDG representation. The data dependences are represented by the dotted edges. For example, the data dependence due to the definition of i in instruction 1 and the use of i in instruction 3 is represented by a directed edge from node 1 to node 3. The WHILE loop condition and IF statement are represented by predicate nodes P1 and P2, respectively. The region node R1 represents the control conditions on program entry, region node R2 represents the conditions on entering the loop or looping back to possibly execute another iteration of the loop, R3 represents the conditions under which the body of the loop is executed, R4 represents the THEN branch, and R5 represents the ELSE branch.

4 Region Scheduling

Region scheduling [20] is a global instruction scheduling technique which operates on the PDG attempting to create regions of code in the procedure containing equal amounts of fine-grain parallelism. The scheduler estimates the number of instructions that can be performed in parallel in a region and uses this estimate to guide the movement of instructions between regions. The goal is to match the estimated parallelism in each region to the amount of parallelism exploitable by the target architecture. Thus, the region scheduler uses information about the underlying parallel architecture, that is, the maximum number of instructions that can be executed in parallel, to guide the scheduling. Although region scheduling can be used to distribute coarse-grain operations, this research focuses on the use of region scheduling to increase and distribute available fine-grain parallelism.

In order to perform region scheduling, a PDG must be extended as follows:

1. Each region node in the PDG is labeled with an estimate of the amount of available parallelism in that region. The parallelism in a region R_i is defined as the ratio, O_i/D_i where O_i is the number of operations in the region and D_i is the length of the longest data dependence path in the data dependence subgraph of the region.
2. Loops are marked as suitable or unsuitable for unrolling. A suitable loop contains only statement nodes (i.e., the loop contains no conditionals).
3. The order of statement nodes in a region from left to right corresponds to the order of the source program. This enables the statements considered for motion to be examined in the desired order.

This is a subset of the extensions proposed by [20]. In order to simplify our implementation,

```
Procedure Region_Scheduling(entry_node){
Input: entry node of PDG
Output: transformed PDG

  Regions = list of regions in the PDG
  Loop {
    for ($i = 0$; $i < |Regions|$; $i++$)
      est_parallelism($Regions_i$)
    list = need_parallelism(Regions)
    $i = 0$
    moved = false
    while (i < |list| && moved == false) {
      moved = peel($list_i$)
      if (moved == false)
        moved = invariant_motion ($list_i$)
      if (moved == false)
        moved = forward_motion($list_i$)
      if (moved == false)
        moved = backward_motion($list_i$)
      i++
    }
    if (moved) cleanup_pdg()
  } while (moved)
}
```

Figure 2: Region Scheduling

our test cases consist of programs in which all loops are single-entry, and our assumed architecture does not allow the evaluation of a predicate to be carried out in parallel with operations succeeding the predicate.

Figure 2 contains an overview of our implementation of region scheduling. The *est_parallelism* function attaches an estimate of the number of instructions which can be executed in parallel in the region. See [20] or [23] for a more detailed discussion on how this estimate can be calculated. Note that although not indicated in this algorithm, the estimate need not be recalculated for every region of the PDG after each transformation. The next function, *need_parallelism*, builds a list of regions with estimates of parallelism that are less than that exploitable by the underlying architecture. Region scheduling will attempt to increase the parallelism in the regions in that list by moving instructions from regions with excess parallelism into regions with insufficient parallelism. A region contains an excess amount of parallelism if the estimate of parallelism in the region is greater than the amount exploitable by the underlying architecture.

Like[20], the region scheduling transformations are applied in order of increasing difficulty. $List_i$ is the current region being examined for code movement because it was found to have insufficient parallelism. The first transformation, *peel*, will peel off one iteration of a loop and move it into region $list_i$ if $list_i$ is the region to be executed immediately before an unrollable loop. If this transformation is not possible, the next transformation, *invariant_motion*, is attempted. In particular, if the region to be executed immediately after $list_i$ is a loop region containing excess parallelism, *invariant_motion* moves loop invariant computations from this loop region to region $list_i$. If neither of these first two transformations is possible, then the scheduler attempts to move code forward from a region, R_j, in the PDG to the region $list_i$ where region R_j contains an excess amount of parallelism. If none of the first three transformations is possible, the scheduler will attempt to move code backward from a region R_k to the region $list_i$ where R_k contains an excess amount of parallelism. If a transformation has been applied to $list_i$ then the inner loop is exited and function *cleanup_pdg* is called to remove a completely unrolled loop from the PDG and eliminate region nodes without descendants. Otherwise, the next region with insufficient parallelism is examined for potential code motion. Region scheduling continues to apply transformations to the PDG until all regions contain sufficient parallelism or until no transformations can be applied.

5 RASER

Figure 3 contains an overview of RASER. As in standard region scheduling, the first FOR loop estimates the available parallelism in each region in the PDG. The second FOR loop performs *Integrated Prepass Scheduling* (IPS) [19, 8] on each region. Instead of simply building the list of regions that need more parallelism, RASER also builds a list, *max_list*, of regions in which the number of live variables in that region exceeds the number of physical registers in the target architecture. This involves computing what we call *maxlive*, the maximum number of live variables in a region. Since the number of live variables is greater than the number of physical registers, the variables that are live in that region will be candidates for spilling by the subsequent register allocation phase. RASER attempts to reduce the amount of spill code to be generated by the register allocator, by reducing the amount of live variables in these regions. The transformation, *live value reduction*, is applied to each region in *max_list*.

After live value reduction, RASER applies transformations to increase parallelism in regions with an insufficient amount. These transformations are essentially the same as those in region scheduling except that they have been modified to be *sensitive* to register allocation. Finally, RASER calls *cleanup_pdg* to eliminate any completely unrolled loops or region nodes with no descendants. The remainder of this section describes the procedures of RASER in more detail.

5.1 Integrated Prepass Scheduling

The procedure *ips_sched*, based on IPS [19] and improved IPS [8], performs local scheduling on each region while keeping track of the number of live variables. By incorporating integrated prepass scheduling into RASER, we are able to obtain a better estimate of the maximum number of live variables than if the scheduling was

```
Procedure RASER(entry) {
Input: entry node of PDG
Output: transformed PDG

  $Regions$ = list of regions in the pdg
  Loop {
    for ($i = 0$; $i < |Regions|$;$i++$)
      est_parallelism($Regions_i$)
    for ($i = 0$; $i < |Regions|$;$i++$)
      ips_sched($Regions_i$)
    for ($i = 0$; $i < |Regions|$;$i++$)
      calc_max_live($Regions_i$)
    $sch\_list$ = need_parallelism($Regions$)
    $max\_list$ = too_many_max_live($Regions$)
    $moved$ = false
    $i = 0$
    while ($i < |max\_list|$ && $moved$ == false) {
      $moved$ = live_value_reduction($max\_list_i$)
      $i++$
    }
    $i = 0$
    while ($i < |sch\_list|$ && $moved$ == false) {
      $moved$ = peel($sch\_list_i$)
      if ($moved$ == false)
        $moved$ = sen_invariant_motion($sch\_list_i$)
      if ($moved$ == false)
        $moved$ = sen_forward_motion($sch\_list_i$)
      if ($moved$ == false)
        $moved$ = sen_backward_motion($sch\_list_i$)
      $i++$
    }
    if ($moved$) cleanup_pdg()
  } while ($moved$)
}
```

Figure 3: RASER

performed after RASER. Since the data dependence dag must be built for each region for calculating the estimated parallelism, IPS can be performed without a significant increase in RASER execution time. IPS needs to be executed initially once on each region. If the set of variables live on entrance to a region R is changed because of code movement in the PDG, or if code is moved either into or out of R, then IPS is executed on that region again.

The *ips_sched* procedure works by oscillating in its heuristic for scheduling based on whether the current number of live variables has reached the register limit. If the number of live variables is less than the register limit, then *ips_sched* schedules instructions to increase the amount of fine-grain parallelism. Otherwise, *ips_sched* schedules an instruction in order to free registers. The register limit for the region is set to be the maximum of 3 and the value obtained by taking the number of physical registers minus the number of variables live on entrance to the region.

5.2 Max Live Calculation

The *calc_max_live* algorithm examines the statements in the region in reverse order keeping track of the number of live variables at each statement. If two variables are live at some point, then these variables can not be assigned to the same physical register. Thus, the set of variables in the set *live* must be assigned distinct registers and if the size of this set exceeds the number of physical registers, the register allocator will need to spill one or more of these variables.

Note that the maximum number of live variables is not equal to the number of physical registers needed to perform the allocation without spilling, and the number of physical registers needed may actually be more. Even if the maxlive of every region is less than the number of physical registers, spill code may still need to be inserted during register allocation. RASER uses the *calc_max_live* algorithm to identify a subset of the situations in which spilling would occur. We felt this was a reasonable method for measuring register demand without having to actually begin performing the allocation.

RASER calculates *maxlive* for each region in the PDG. If the *maxlive* of a region is greater than the number of physical registers, then the function *too_many_max_live* will add the region to *max_list* and RASER will call the *live_value_reduction* function on that region.

5.3 Live Value Reduction

Figure 4 contains an overview of the algorithm which performs live value reduction. In order to perform the reduction, each region must be tagged with the set of variables that are live on entrance and the set of definitions that reach that region. The statements in the region are examined in sequential execution order, and a list of the live variables is maintained with respect to the current point in the region. At any point p in the region, if the number of live variables exceeds the number of physical registers, then each variable, v_j, in the live set at p that has only one reaching definition at p is examined in order to decide whether the reaching definition of v_j can be moved before each of its uses. In the algorithm in figure 4, the four function calls in the expression of the IF statement determine the legality and desirability of the code motion.

The first call, *violated*, determines whether the movement would violate any of the existing data dependences thus making the motion illegal. The live value reduction code motion is essentially the same as the forward code motion applied by standard region scheduling. Thus, the legality tests applied in forward code motion are also applicable here.

The second call, *live_ops*, ensures that at least one of the operands of the reaching definition of v_j is live at p. Otherwise, the code motion may actually increase the number of live variables along the path from the definition to the use. Suppose, for example, RASER is considering to move the statement $a = b + c$ before each of the uses of this definition of a. If the operand b is not live at one of the uses then moving this definition of a forces b to be live at that use and along the path, (and thus at p), from the original definition $a = b + c$ to that use. The function

```
Algorithm live_value_reduction ($R$,$num\_regs$) {
Input: Region $R$
    number of physical registers
Output: Region $R$ with a reduced number of max live

  $live$ = set of variables live on entrance to region $R$
  $S$ = set of $n$ statements in region $R$
  for ($i = 0$; $i < n$; $i++$) {
    if ($|live| > num\_regs$) {
      for ($j = 0$; $j < |live|$; $j++$) {
        if (!violated($v_j$) && live_ops($v_j$)
          && (multiple_uses($v_j$) || diff_regns($v_j$))) {
          move_reaching_def($v_j$)
          $live = live - v_j$
        }
      }
    }
    if ($S_{arg1}$ dead) $live = live - S_{arg1}$
    if ($S_{arg2}$ dead) $live = live - S_{arg2}$
    $live = live \cup S_{result}$
  }
}
```

Figure 4: Live Value Reduction

live_ops returns true if one of the operands is currently live. Changing *live_ops* to return true only if both operands are currently live would ensure that the number of live variables at p is reduced by one, but we only require that the number of live variables is not increased by the movement. This may permit a later movement of the definition of the operand which causes the reduction in the number of live variables that we desire.

The last two calls ensure the termination of *live_value_reduction*. The function, *multiple_uses*, returns true if there is more than one use of the reaching definition of $v_j \in$ live. The function, *diff_regns* returns true if the *single* use of the reaching definition of v_j has a different control dependent parent than the definition. If either of these cases is true, then the code can be moved. This prevents a definition from "chasing" its single use around a region. Instead, we use a local scheduling technique to reorder the code within a region.

Outside of the loop in *live_value_reduction*, the live set is updated for the statement, S, currently under consideration at point p in order to obtain the live information with respect to the next point in the region. If there are no future uses of an operand of S, the operand is removed from the live set. The variable defined by S is added to the live set, and S then becomes the reaching definition of that variable.

Figure 5 contains an example of the application of live value reduction. The definition of a in statement $S1$ is live during the execution of statements $S2$ ... S_{n+2}. Live value reduction moves $S1$ before each use of the definition of a in $S1$ causing a to be not live until immediately prior to the use of a. In the view of the register allocator, this eliminates interferences which would have been added between a and the variables defined in statements $S2$ ... S_n.

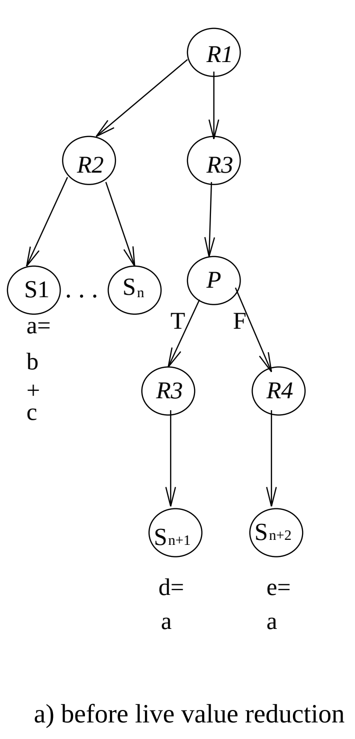
Live value reduction is similar to the scheduling to reduce the number of live variables in IPS. Unlike IPS, live value reduction moves instructions across regions and thus across basic blocks to reduce the number of live variables. Thus, IPS performs cooperative scheduling at a local level and live value reduction performs cooperative scheduling globally.

5.4 Allocation Sensitive Transformations

After RASER performs as many live value reductions as possible, it then begins to execute *register allocation sensitive* region scheduling transformations. These transformations are identical to the transformations of standard region scheduling except that a code motion is not performed if the *maxlive* of R increases due to the code motion to become greater than the number of physical registers. For example in figure 6, before code motion is applied, the live range of k begins at the statement that defines k and extends to the next statement that uses k. If the loop invariant transformation is applied to move the statement outside of the loop, then k would be live during the entire execution of the loop. This transformation could possibly increase the amount of spill code inserted by the register allocator. Before applying the loop invariant motion, the function *sen_invariant_motion* calculates the change, δ, in the number of live variables in the loop that would result because of the motion. The function *sen_invariant_motion* will only perform the code motion if (1) δ is nonpositive or (2) $\delta + maxlive$ is less than the number of physical registers.

The RASER function *sen_forward_motion* (figure 3) will only perform the forward code motion after calculating the increase in the number of live variables along the path from the source to the target. For example in figure 7, suppose an operand of the statement, S_1, in region R_2 is dead after the execution of S_1. If S_1 is moved to node R_4, the operand would then be live along all paths from the R_2 node to R_4. In contrast, the variable defined by statement S_1 would be dead along any path after the motion, thus the movement could possibly reduce the number of variables that are live along the path. Prior to moving the statement, *sen_forward_motion* calculates the change δ in the number of live variables that would occur due to the motion. The forward motion is performed only if (1) δ is nonpositive or (2) for all regions along a path from the source node to the target, $\delta + maxlive$ is less than the number of physical registers. Like *sen_forward_motion*, the function *sen_backward_motion* calculates the change δ in the number of live variables that would occur by a backward code motion, and performs the motion under the same conditions.

The *peel* routine in RASER is identical to the *peel* routine executed by the region scheduling. Loop peeling, without also eliminating dependences, will never increase the number of inter-



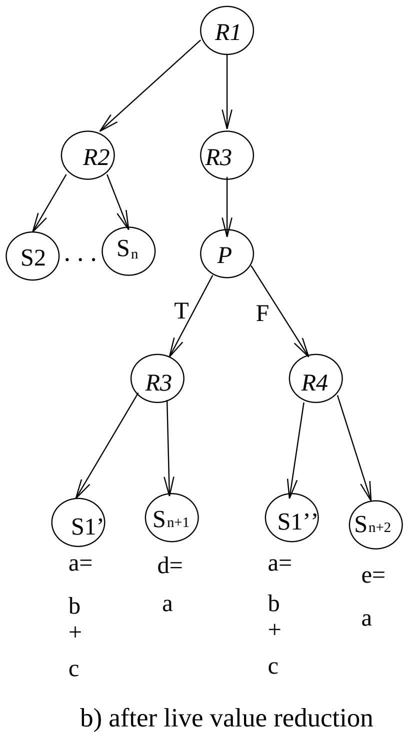


Figure 5: Live Value Reduction Example

```
for i = 1 to 10        k = 3
{                      for i = 1 to 10
   .                   {
   .                       .
   k = 3                   .
   a[i] = k                a[i] = k
   .                       .
   .                       .
}                      }

a) before invariant    b) after invariant
   code motion            code motion
```

Figure 6: Increase in Live Values

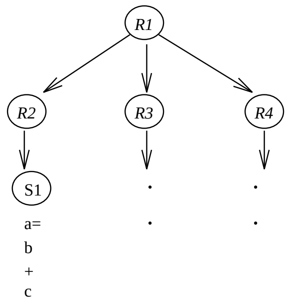


Figure 7: Sensitive Forward Motion

ferences found by the subsequent register allocation phase.

6 Evaluation of RASER

6.1 Experimental Framework

The performance of RASER has been compared to the performance of standard region scheduling shown in figure 2. The front end of these two routines is the *pdgcc* compiler, developed at the University of Pittsburgh [15], which accepts as input C source code and outputs the corresponding PDG. Our region scheduler accepts as input the PDG representation of the C program, performs region scheduling transformations and outputs the transformed PDG. Similarly, RASER accepts as input the PDG representation of the C program, performs register allocation sensitive region scheduling transformations and outputs the transformed PDG. The transformed PDGs are then used as input to *pdg2i* which generates the intermediate code, *iloc*. Iloc is a low-level intermediate code designed at Rice University for the development of optimizing compilers [9].

Local scheduling is performed in RASER by executing IPS on each region of the PDG. In order to design a fair comparison, we executed a local scheduler on the iloc code generated after performing region scheduling. The local scheduler is based on the Gibbons and Muchnick [18] basic block scheduling algorithm. Thus, our experimental study compared a compilation scheme consisting of region scheduling, basic block scheduling, and register allocation against RASER with IPS incorporated into it, followed by register allocation.

Our implementation of the *optimistic allocator* [10] is used to perform register allocation on the iloc code. The iloc code produced is targeted for a hypothetical medium pipelined machine with pipelined functional units. A simulator/interpreter is used to determine the number of cycles required to execute the iloc code and the number of loads and stores executed due to spilling. Performance measurements of the region scheduler and RASER have been taken for 13 of the Livermore Loops, the Clinpack routines, implementations of heapsort, hanoi and 8-queens routines. We varied the number of physical registers from 3 to 16. We also performed experiments in order to isolate the effects of live value reduction and the allocation sensitive transformations.

6.2 Results

Table 1 contains the results of the experiments for register set sizes 3, 4, and 7. The *tot* columns show the percentage decrease in the total cycles executed, calculated as $(cycles(RS) - cycles(RASER))/cycles(RS)$. $Cycles(RS)$ is the number of cycles executed by the iloc interpreter/simulator when region scheduling/local scheduling have been applied. $Cycles(RASER)$ is the number of cycles executed by the iloc interpreter/simulator when RASER was applied. The *ld* column contains the percentage decrease in the number of loads executed, calculated by $(loads(RS) - loads(RASER))/loads(RS)$ where $loads(RS)$ and $loads(RASER)$ are the number of loads executed due to spilling by the region scheduling and by RASER, respectively. The *st* column contains the percentage decrease in the number of stores executed, calculated by $(stores(RS) - stores(RASER))/stores(RS)$ where $stores(RS)$ and $stores(RASER)$ are the number of stores executed due to spilling. Blank entries represent cases where no spilling occurred and $-\infty$ indicates that there were no loads (or stores) in the region scheduling/local scheduling generated code for that experiment.

Our experimental study indicates that for smaller register set sizes, in general, RASER generates code that executes faster than code generated via standard region scheduling/local scheduling. In 17 out of the 24 experiments assuming three registers and 19 out of the 24 experiments assuming four registers, the study showed a positive percentage decrease in the number of cycles executed. In those cases, the reduction in the number of cycles executed varied from 0.2% to 49.2%. As the number of registers increased, the percentage decrease in the number of cycles executed became smaller. For example, in a register set size of 7 for these experiments, where there was little spilling, the percentage decrease was positive in only 7 out of the 24 experiments. This agrees with what is expected to happen. With larger test programs with much more demand for registers, we would expect that the number of registers for which this starts to happen would be larger.

In nearly all experiments, RASER code contains less spill code than the code generated via region scheduling/local scheduling. Thus, RASER is successful in cooperating with register allocation in that it reorders code to reduce spilling. RASER is not as successful at reordering code to exploit fine-grain parallelism. As the number of registers increases, reordering to increase fine-grain parallelism becomes more important than reordering to reduce spilling. The region scheduling/local scheduling scenario is more successful at reordering to increase fine-grain parallelism given an ample amount of registers. Thus, we see that the performance of RASER degrades as the number of registers increases. This is partly attributable to performing both the global scheduling and local scheduling in RASER on high level intermediate code and not on the low-level code, iloc. *Pdg2i* may generate several iloc statements with dependences between them for each high level intermediate code in the PDG. Another pass of IPS on the basic blocks of the iloc code prior to register allocation or executing *scheduler-sensitive global register allocation* [25] would likely improve the performance of RASER.

We found by isolating the effects of live value reduction and the sensitive transformations, that the live value reduction sometimes produces a decrease in the amount of spill code inserted by the subsequent register allocation phase and sometimes increases the amount. Live value reduction is able to reduce the amount of spill code introduced when the code movement is able to reduce the number of live variables so that spilling is not necessary. Returning to the live value reduction example, in figure 5, if moving the statement $a = b + c$ before the uses of this definition of a prevents a from being spilled, then live value reduction has successfully eliminated one store and two loads. Essentially, we are “trading” the execution of the two loads for execution of the definition. Instead of executing a load of the value a just prior to its use, the statement that defines a is executed just prior to its use. In general, it is cheaper to execute the defining statement than to execute a memory operation.

Live value reduction is not able to reduce the amount of spill code introduced when the operands of the moved instruction are spilled. For example, in figure 5, if either of the operands are spilled then live value reduction actually increases the amount of spill code that would be introduced by the register allocator. Spilling variable

Benchmark	**Number of Registers**								
	3			**4**			**7**		
	tot	**ld**	**st**	**tot**	**ld**	**st**	**tot**	**ld**	**st**
Livermore									
loop1	22.0	33.3	47.1	25.5	43.7	77.7	-16.9	16.7	-98.8
loop2	45.8	62.7	77.1	45.5	65.2	85.7	22.9	64.1	58.8
loop3	5.5	14.3	25.0	14.3	40.0	66.6	-25.6		
loop4	4.1	14.3	57.1	-17.7	0.4	0.4	-40.0	50.9	80.0
loop6	12.7	35.2	71.5	17.3	51.2	70.1	-19.8	100.0	100.0
loop7	39.6	55.9	77.5	34.5	59.3	71.4	14.2	47.8	64.6
loop8	-23.8	-6.7	13.1	-24.3	-3.5	0.9	-28.7	-6.5	16.3
loop9	43.1	61.0	94.8	49.2	71.8	96.4	27.5	67.5	86.0
loop10	-8.2	22.2	32.3	-7.0	30.4	27.3	-22.1	60.2	61.7
loop11	22.2	46.2	66.7	22.6	66.7	60.0	-41.5	100.0	100.0
loop12	26.4	50.0	70.0	13.6	42.9	99.9	-65.4	-100.0	-100.0
loop13	-50.6	-32.7	-18.2	-70.8	-39.7	-120.9	-66.9	11.1	21.5
loop14	22.3	35.9	54.7	19.0	37.8	47.0	2.9	52.2	49.9
Clinpack									
idamax	-0.2	0.0	0.0	0.2	0.0	0.0	-1.2	0.0	0.0
dscal	-9.8	0.0	0.0	3.9	0.0	33.3	-38.5	0.0	$-\infty$
daxpy	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
matgen	24.4	35.8	66.2	16.9	29.8	86.2	-18.6	0.0	-99.6
dgefa	5.7	21.3	5.2	3.2	21.6	20.9	3.7	16.4	31.5
Heapsort									
hsort	-1.9	8.3	18.0	-1.1	13.9	24.7	-18.7	-4.1	-8.1
Hanoi									
main	10.1	43.5	71.2	5.1	63.8	58.3	-17.7	85.7	78.6
mov	-7.2	3.8	10.0	17.9	28.6	25.0	-17.5		
8 Queens									
queens	0.0			0.0			0.0		
try	12.4	34.4	2.3	10.4	27.3	-18.6	4.3	21.7	0.0
doit	16.1	42.7	91.0	4.1	46.3	16.7	-11.5	75.0	33.3

Table 1: Percentage decrease in cycles executed

b in figure 5 (a) causes one load to be inserted prior to the use of *b*; spilling variable *b* in figure 5 (b) causes two loads to be inserted. In this situation, reducing the live range of one variable by moving its definitions before each of its uses is not enough to eliminate the need to spill the operands of the definition. Since the operands of the definition are spilled, RASER inadvertently causes an increase in the amount of spill code.

We found from our experiments that live value reduction is much more aggressive than the sensitive transformations. Large reductions in the amount of spill code are due to the application of live value reduction. However, the sensitive transformations are simple to apply and sometimes can result in a modest decrease in the amount of spill code generated. In addition, unlike live value reduction, sensitive transformations rarely produce an increase in the amount of spill code.

7 Conclusions

In this work, we set out to develop a register allocation sensitive global scheduler based on region scheduling, with the dual goals of increasing the amount of fine-grain parallelism and decreasing the amount of simultaneously live variables. We describe how we achieved these goals by simple transformations to a region scheduler and by introducing a new transformation known as *live value reduction*. We have incorporated a local instruction scheduler into our sensitive global scheduler in order to improve the estimates of live values used in the sensitive global scheduler. Our experiments show that this approach does indeed reduce the execution time and number of spills in programs that have a demand for registers greater than the number available. Our future work includes integrating RASER with our other cooperative register allocation and instruction scheduling schemes, and performing various experimental studies to determine the most beneficial combination of cooperation.

References

[1] A. Aiken and A. Nicolau. A development environment for horizontal microcode. *IEEE Transactions on Software Engineering*, 14(5):584–594, May 1988.

[2] V. H. Allan, J. Janardhan, R. M. Lee, and M. Srinivas. Enhanced region scheduling on a program dependence graph. In *Proceedings of the twenty-fifth International Symposium on Microarchitecture*, pages 72–80, Portland, OR, 1992.

[3] F. E. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5:617–640, 1988.

[4] W. Baxter and H. R. Bauer, III. The program dependence graph and vectorization. In *Proceedings of the Sixteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, Austin, TX, 1989.

[5] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, CANADA, June 1991.

[6] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. URSA: A unified resource allocator for registers and functional units in VLIW architectures. In *IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, Florida, January 1993.

[7] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. Resource spackling: A framework for integrating register allocation in local and global schedulers. In *PACT '94: International Conference on Parallel Architectures and Compilation Techniques*, Montreal, Canada, August 1994.

[8] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, Santa Clara, CA, April 1991.

[9] Preston Briggs, Keith D. Cooper, and Linda Torczon. R^n Programming Environment Newsletter #44. Department of Computer Science, Rice University, September 1987.

[10] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992.

[11] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 192–203, Toronto, CANADA, June 1991.

[12] Gregory Chaitin, Marc Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[13] Frederick Chow and John Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4), October 1990.

[14] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[15] Claude-Nicolas Fiechter. PDG C Compiler. University of Pittsburgh, 1992.

[16] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[17] S. M. Freudenberger and J. C. Ruttenberg. Phase ordering of register allocation and instruction scheduling. In *Code Generation - Concepts, Tools, Techniques: Proceedings of the International Workshop on Code Generation*, May 1992.

[18] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, June 1986.

[19] James R. Goodman and Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. In *Supercomputing '88 Proceedings*, pages 442–452, Orlando, Florida, November 1988.

[20] Rajiv Gupta and Mary Lou Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, 1990.

[21] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In *International Workshop on Compiler Construction*, Paderdorn, GERMANY, October 1992.

[22] J. L. Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[23] Jayashree Janardhan. Enhanced region scheduling for instruction level parallelism. Utah State University Master's Thesis, 1992.

[24] S-M Moon and K. Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *Proceedings of the twenty-fifth International Symposium on Microarchitecture*, Portland, OR, 1992.

[25] Cindy Norris and Lori L. Pollock. A scheduler-sensitive global register allocator. In *Supercomputing '93 Proceedings*, Portland, OR, November 1993.

[26] Cindy Norris and Lori L. Pollock. Register allocation over the program dependence graph. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994.

[27] K. J. Ottenstein. An intermediate program form based on a cyclic data-dependence graph. Technical Report 81-1, Department of Computer Science, Michigan Tech. University, 1981.

[28] S. S. Pinter. Register allocation with instruction scheduling: a new approach. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, June 1993.

[29] Todd A. Proebsting and Charles N. Fischer. Probabilistic register allocation. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 300–310, San Francisco, CA, June 1992.

[30] J. Warren. A hierarchical basis for reordering transformations. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 272–282, 1984.

[31] S. Weiss and J. E. Smith. A study of scalar compilation techniques for pipelined supercomputers. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[32] M. J. Wolfe. *Research Monographs in Parallel and Distributed Computing.* The MIT Press, 1989.

[33] H. Young. Evaluation of a decoupled computer architecture and the design of a vector extension. *Computer Sciences Technical Report 603*, 21(4), July 1985.