

Experiences with Cooperating Register Allocation and Instruction Scheduling *

Cindy Norris

Lori L. Pollock

Mathematical Sciences
Appalachian State University
Boone, NC 28608
(704)262-2359
can@cs.appstate.edu

Computer and Information Sciences
University of Delaware
Newark, DE 19716
(302) 831-1953
pollock@cis.udel.edu

June 29, 1998

Abstract

Compile-time reordering of low level instructions is successful in achieving large increases in performance of programs on fine grain parallel machines. However, because of the interdependences between instruction scheduling and register allocation, a lack of cooperation between the scheduler and register allocator can result in generating code that contains excess register spills and/or a lower degree of parallelism than actually achievable. This paper describes a strategy for providing cooperation between register allocation and both global and local instruction scheduling. We experimentally compare this strategy with other cooperative and uncooperative scenarios.

Keywords: *register allocation, instruction scheduling, fine grain parallelism*

1 Introduction

A major focus of optimizing compilers for architectures supporting instruction level parallelism has been the rearrangement of the low level code with the ultimate goal of increasing the amount of parallelism that a program can exploit by hiding instruction latencies and thus reducing possible run-time delays [26, 21, 43].

A scheduler that rearranges code within a basic block in isolation of the rest of the program is called a local scheduler [1, 7, 21, 26, 6]; a scheduler that moves instructions across basic blocks by considering the effects

*This work was partially supported by NSF under grant CCR-9625219.

of code movement on a global level is called a global scheduler [23, 30, 8, 19, 40]. The goal of an ambitious register allocator is to allocate the machine's physical registers to program values to minimize the number of run-time memory accesses. Register allocation techniques are either local [27], global [15, 14, 13, 38, 12, 25, 33, 29, 24], or interprocedural [41, 39] depending on whether the allocator attempts an assignment of registers to values within basic blocks in isolation of other basic blocks, across basic blocks of a procedure, or across procedure boundaries, respectively.

The objectives of instruction scheduling and register allocation cause conflicts in code generation in several ways: (1) The actual assignment of the same physical register to different virtual registers can create dependences that did not exist in the original code, but can also save spilling another register to hold one of the current values to be allocated. (2) While a particular ordering of instructions may increase the potential for instruction level parallelism, the reordering due to instruction scheduling may also extend the lifetime of certain values, which can increase register pressure. (3) The instruction scheduler wants an adequate number of local registers to avoid register reuse, since register reuse limits the opportunity for instruction level parallelism. However, the register allocator would prefer sufficient global registers in order to avoid spills at basic block boundaries. (4) An effective scheduler can lose its achieved degree of instruction level parallelism when spill code is inserted afterwards.

Without any communication of information and cooperation between the scheduling and allocation phases, the compiler writer faces the problem of determining which of these phases should run first to generate the most efficient final code. The lack of communication and cooperation between the instruction scheduler and the register allocator can result in generating code that contains excess register spills and/or a lower degree of instruction level parallelism than actually achievable.

Techniques have been developed to provide communication of requirements and cooperation between local instruction scheduling and local register allocation [22], cooperation between local instruction scheduling and global register allocation [11, 32, 37], and cooperation between global instruction scheduling and register allocation [20, 17, 9, 10, 34]. 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. However, the experimental studies have focused on providing cooperation between register allocation and either local instruction scheduling or global instruction scheduling and have not experimentally studied whether it is worthwhile to provide cooperation with both levels of instruction scheduling, or experimentally compared cooperative local scheduler approaches with cooperative global scheduler approaches.

This paper describes a strategy for providing cooperation between register allocation and both global and local instruction scheduling. Our approach has been to maintain three separate, but cooperative, phases of global instruction scheduling, register allocation, and local instruction scheduling. The global instruction scheduler has been made sensitive to the subsequent register allocation phase [34, 31]. The register allocator has been modified to take into consideration the actions of the subsequent local instruction scheduling phase [32]. This overall organization of the phases has these advantages: (1) By keeping separate, cooperative phases, the complexities of attempting to perform register allocation, global instruction scheduling and local instruction scheduling simultaneously are avoided. (2) By applying the local code scheduler after register allocation, spill code inserted during register allocation will be carefully scheduled along with the other instructions during normal execution of the scheduler.

We have experimentally compared the performance of this strategy against our previous strategies of cooperation with only local scheduling and cooperation with only global scheduling, in addition to normal postpass local scheduling, and uncooperative global and local scheduling in combination. We report on these results in this paper.

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 describes RASER, our global instruction scheduling technique which is sensitive to register allocation. Section 4 describes SSG, our global register allocation technique which cooperates with local instruction scheduling. Section 5 describes the experimental study of the various scenarios of cooperative register allocation and instruction scheduling. Section 6 summarizes and discusses future directions.

2 Previous Work on Cooperative Strategies

This section discusses previous strategies to provide cooperation between register allocation and instruction scheduling. Section 2.1 describes techniques which provide cooperation between register allocation and per-basic block, ie. local, instruction scheduling. Section 2.2 describes techniques which provide cooperation between register allocation and schedulers which move instructions across basic block boundaries.

2.1 Register Allocation and Local Instruction Scheduling

Goodman and Hsu [22] have developed two separate cooperative strategies. In their cooperative postpass scheduling approach, the local register allocator runs first and uses information typically used by the scheduler, namely the data dependence dag, to guide the register allocator in making its allocation, while the scheduler is constrained by dependences added by the register allocator. The test results showed that dag-driven local register allocation significantly improved the performance of postpass code schedulings.

Alternatively, in their second approach, called integrated prepass scheduling, the instruction scheduler is executed first, but is constrained by a limit on the number of registers that it has available and thus oscillates in its heuristic for scheduling based on whether the current number of live variables has reached the register limit. If there are insufficient available registers, then the scheduler tries to schedule an instruction which will decrease the number of live values, otherwise the scheduler schedules to increase fine grain parallelism. On highly pipelined machine models, the integrated prepass scheduling approach slightly outperformed the dag-driven register allocation approach.

Bradlee, Eggers, and Henry [11] developed a slightly modified version of integrated prepass scheduling (which they call IPS) in which they calculate the register limit in a different way, replace the local allocator with a global allocator, and invoke the scheduler again after allocation in order to better schedule spill code. Their version of integrated prepass scheduling showed greater speedups than those found by Goodman and Hsu.

Bradlee, Eggers, and Henry also developed a more integrated approach called RASE. A prescheduling phase is run in which the scheduler is invoked twice in order to calculate cost estimates for guiding the register

allocator. A global register allocator then uses the cost estimates and spill costs to obtain an allocation and to determine a limit on the number of local registers for each block. A final scheduler is run using the register limit from allocation and inserting spill code as it schedules. Because the scheduler is invoked 3 to 4 times, RASE runs six times slower than postpass scheduling on average without significant improvement in the generated code quality over IPS. In addition, the final scheduler is complicated by simultaneously performing local register allocation with instruction scheduling.

Pinter [37] developed a cooperative approach based on building a parallel interference graph which represents both register allocation conflicts and scheduling constraints. If the parallel interference graph is colorable, it provides a register allocation which does not generate false dependences. Heuristics are provided for trading off between scheduling and register allocation. No experimental work is presented to compare this technique to previous methods.

2.2 Register Allocation and Global Instruction Scheduling

Freudenberger [20] developed a technique for integrating register assignment with trace scheduling in the MultiFlow compiler. 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 that is needed. A downfall of the trace-driven approaches to scheduling is that they require the user to spend considerable time obtaining program profiles.

Ebcioğlu and Nicolau [17] presented a resource-constrained code scheduling technique for VLIW and super-scalar 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 path of movement. 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 [9] developed a technique called URSA for allocating both functional units and registers in VLIW machines. In attempting 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. They are investigating methods to enable URSA to handle superscalar architectures.

In a later paper [10], 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 instruction scheduling is given. Thus far, no experimental results have been obtained to judge the effectiveness of the URSA approach.

Novack and Nicolau [35] designed a technique called Mutation Scheduling which combines code selection, register allocation and instruction scheduling into a unified framework. Associated with each value in the program is a set of functionally equivalent expressions called the mutations of the value. When scheduling an expression to compute the value, the mutation selected is the one which best fits the available resources. Like [17], Mutation Scheduling starts with an initial register allocation and modifies the allocation dynamically

during scheduling in response to changing resource availability, attempting to eliminate false dependences. When a value is computed and stored in a register, that register becomes one of the mutations of the value. If a value is spilled, then one of the mutations of the value is a load of that value.

3 Register Allocation Sensitive Region Scheduling

We provide cooperation between global scheduling and register allocation via our technique known as **Register Allocation Sensitive Region scheduling** (RASER) [34], which is a modification of a global instruction scheduling technique known as *region scheduling* [23] to take into consideration the requirements of a subsequent register allocation phase. 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 code which will be introduced in the subsequent register allocation phase.

Region scheduling was chosen to study the potential for cooperation between a global instruction scheduler and a register allocator for several reasons. Foremost, the region scheduling algorithm uses the program dependence graph. 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 since the PDG has been used successfully as the basis for various scalar optimizations [18, 36, 5] as well as for detecting and improving parallelization for vector machines [42, 5], multiple processor machines [44, 4], and architectures that exhibit instruction level parallelism [23, 8, 3]. Although the global scheduling algorithm of Bernstein and Rodeh [8] also operates on the PDG, their approach only achieved modest improvements over local instruction scheduling. Other global scheduling techniques such as trace scheduling [19] and percolation scheduling [2] utilize the control flow graph rather than a program dependence graph to rearrange code.

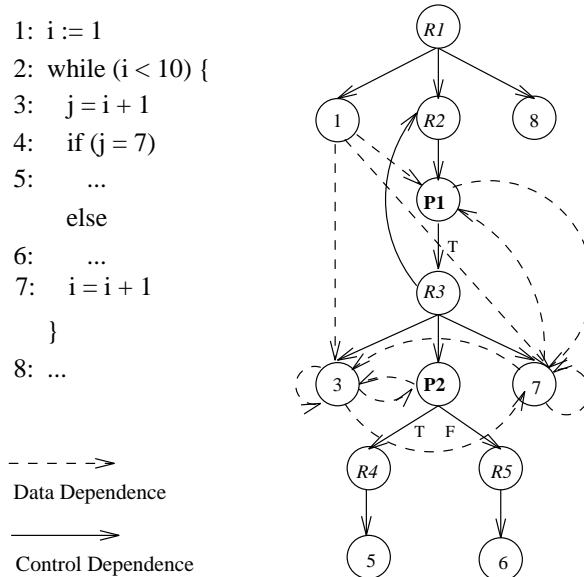


Figure 1: Program dependence graph

3.1 Program Dependence Graphs

The program dependence graph [18] 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 . 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

```

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(Regionsi)
    list = need_parallelism(Regions)
    i = 0
    transform = false
    while (i < |list| && transform = false) {
      transform = apply_peel(listi)
      if (transform = false)
        transform = apply_loop_invariant_motion (listi)
      if (transform = false)
        transform = apply_forward_motion(listi)
      if (transform = false)
        transform = apply_backward_motion(listi)
      i++
    }
    if (transform) cleanup_pdg()
  } while (transform)
}

```

Figure 2: Region scheduling

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. Thus, region node insertion creates a hierarchical organization of the PDG. Figure 1 shows a program segment and its PDG representation.

3.2 Region Scheduling

Region scheduling [23] is a global instruction scheduling technique which operates on the program dependence graph 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.

Figure 2 contains an overview of the implementation of region scheduling used in the experimental study. The *est_parallelism* procedure attaches an estimate of the number of instructions which can be executed in parallel in the region. This estimate is calculated as O_i/D_i where O_i is the number of operations in region R_i and D_i is the length of the longest data dependency chain [23, 28]. Note that although not indicated in this algorithm, the estimate need not be recalculated for every region of the PDG after each transformation. Only those regions which have been changed via a transformation need to have the estimate recomputed. 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. The parallelism in the architecture is measured as the maximum number of operations that the system can perform in parallel. Like [23], 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. If during the course of executing the inner loop, a transformation has been applied to $list_i$ then the loop is exited and the procedure *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.

3.3 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) [22, 11] on each region. IPS is described in detail in section 3.3.1. 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

```

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(Regionsi)
    for (i = 0; i < |Regions|; i++) ips_sched(Regionsi)
    for (i = 0; i < |Regions|; i++) calc_max_live(Regionsi)
    schList = need_parallelism(Regions)
    maxList = too_many_max_live(Regions)
    transform = false
    i = 0
    while (i < |maxList| && transform = false) {
      transform = live_value_reduction(maxListi)
      i++
    }
    i = 0
    while (i < |schList| && transform = false) {
      transform = apply_peel(schListi)
      if (transform = false)
        transform = apply_sen_loop_invariant_motion(schListi)
      if (transform = false)
        transform = apply_sen_forward_motion(schListi)
      if (transform = false)
        transform = apply_sen_backward_motion(schListi)
      i++
    }
    if (transform) cleanup_pdg()
  } while (transform)

```

Figure 3: RASER

computing *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 *maxList*. Live value reduction is described in detail in section 3.3.3.

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. Section 3.3.4 explains how these transformations have been modified. Finally, RASER calls *cleanup_pdg* to eliminate any completely unrolled loops or region nodes with no descendants.

3.3.1 Integrated Prepass Scheduling

The procedure *ips_sched*, based on IPS [22, 11], performs *local* scheduling on each region while keeping track of the number of live variables. IPS was chosen to perform on each region in the PDG during RASER for the following reasons: (1) By incorporating integrated prepass scheduling into RASER, one is able to obtain a better estimate of the maximum number of live variables than if the scheduling was performed after RASER. (2) 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 subtracting the number of variables live on entrance to the region from the number of physical registers.

3.3.2 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.

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.

```

Algorithm live_value_reduction ( $R, num\_regs$ ) {
/**
Input: Region  $R$ 
            $num\_regs$ , 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  $R$ 
            $S_{r_{op1}}$  = first operand of statement  $S_r$ 
            $S_{r_{op2}}$  = second operand of statement  $S_r$ 
***/
  for ( $i = 0; i < n; i++$ ) {
    if ( $|live| > num\_regs$ ) {
      for ( $j = 0; j < |live|; j++$ ) {
        if ( $!violated(v_j) \ \&\& \ live\_ops(v_j)$ 
           $\ \&\& \ (multiple\_uses(v_j) \ || \ diff\_regns(v_j))$ ) {
           $S_r$  = reaching def of  $v_j$ 
          move a copy of  $S_r$  before each of the uses  $v_j$ 
           $live = live - v_j$ 
          if ( $S_{r_{op1}} \notin live$ )
             $live = live \cup S_{r_{op1}}$ 
          if ( $S_{r_{op2}} \notin live$ )
             $live = live \cup S_{r_{op2}}$ 
        }
      }
    }
    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

3.3.3 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.

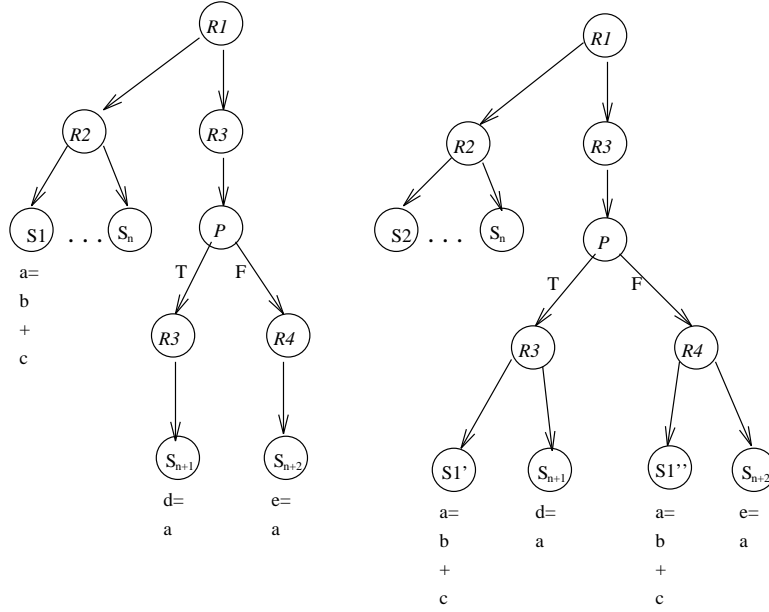
The second call, *live_ops*, insures that at least one of the operands of the reaching definition of v_j is live at p . Otherwise, the code motion may actually increase the number of live variables along the path from the definition to the use. The function *live_ops* returns true if one of the operands is currently live. The last two calls insure 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.

If the live value reduction is performed, the live set is updated with respect to the statement just moved. If one of the operands of the moved statement was not live at point p , the movement would cause it to now be live and the live set is updated to include that operand. Note that at most one of the operands will now be live, if it was not previously live at p . Also, the variable defined by the statement moved will be removed from the live set.

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 $S2 \dots S_{n+2}$. 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 $S2 \dots S_n$.

Live value reduction is similar to scheduling to reduce the number of live variables in IPS. IPS reorders only within a basic block oscillating in its scheduling technique based on whether the current number of live variables has reached the register limit. 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



(a) before live value reduction (b) after live value reduction

Figure 5: Live value reduction example

local level, and live value reduction performs cooperative scheduling globally.

3.3.4 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, before applying loop invariant motion, the function *sen_invariant_motion* precisely calculates the change, δ , in the number of live variables in the loop that would result because of the motion. The function *apply_sen_loop_invariant_motion* will only perform the code motion if (1) δ is nonpositive or (2) $\delta + \text{maxlive}$ is less than the number of physical registers.

Similarly, the RASER functions *apply_sen_forward_motion* and *apply_sen_backward_motion* will only perform the forward code motion if doing so does not increase the number of live variables along the motion path so

that the number of live variables is greater than the number of physical registers in the target architecture. The *apply-peel* routine in RASER is identical to the *apply-peel* routine executed by the region scheduling. Loop peeling, without also eliminating dependences, will not increase the number of interferences found by the subsequent register allocation phase.

4 The Scheduler-Sensitive Global Register Allocator

To provide cooperation between local instruction scheduling and register allocation, we used a Scheduler Sensitive Global Register Allocator (SSG) [32]. The goal in designing SSG was to improve on Goodman and Hsu [22] by addressing the problem of integrating *global* register allocation and local scheduling. In addition, a simpler and less expensive scheme than RASE [11] was desired which has been shown to be unnecessarily complex for its resulting code improvement over Goodman and Hsu. Thus, the requirements were: (1) Exploit advanced global register allocation schemes. (2) Invoke the scheduler only one time to avoid the overhead of multiple invocations of the scheduler. (3) Avoid the complexity of simultaneous allocation and scheduling. (4) Avoid the problem of partitioning local and global registers by making all variables compete equally for registers.

SSG performs a graph coloring allocation that takes into consideration the objectives of the subsequent scheduling phase throughout each phase of the allocation. This approach allows the spill code to be scheduled along with the other instructions during the normal execution of the scheduler. In addition, the complexity of simultaneously performing local register allocation and instruction scheduling is avoided, and multiple invocations of the scheduler are not needed.

4.1 The Base Register Allocator

Based on its demonstrated success, the *optimistic allocator* (OA) developed by Briggs, Cooper, and Torczon [12] was selected as the base allocator. Like most global register allocators, OA is based on a graph coloring allocation method, namely, the method developed by Chaitin [14]. An interference graph is constructed in

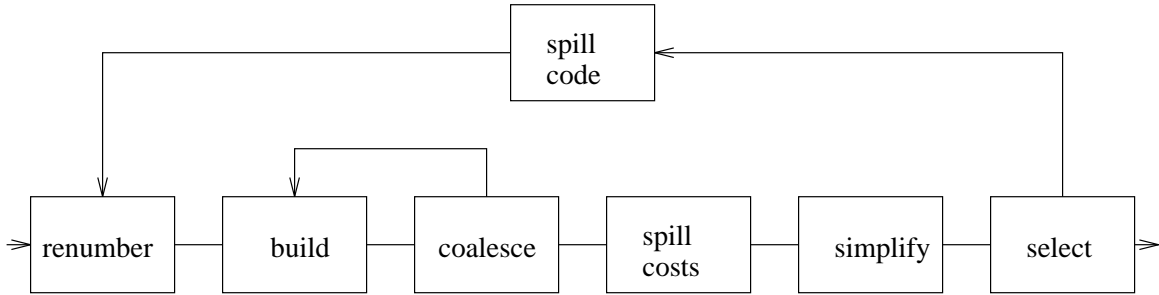


Figure 6: The Base Allocator (Optimistic Allocator)

which the nodes represent candidates for physical registers, and edges connect nodes that must be assigned different physical registers because their values will coexist during program execution. Using no more than k colors where k is the number of available physical registers, the allocator attempts to find a k -coloring of the interference graph. Figure 6 depicts the high level view of the base allocator, OA.

The input to the first phase, the RENUMBER phase, consists of intermediate code generated assuming an unlimited number of virtual registers. In the first step of the RENUMBER stage, the Static Single Assignment (SSA) graph [16] is computed. This representation transforms the code so that each variable use corresponds to a single definition. To achieve this, virtual registers are renumbered and special functions called phi functions are inserted into the code. The RENUMBER stage also determines which values in the program can be *rematerialized*. A value can be rematerialized if it can be recomputed immediately before its use because its operands are always available.¹ A rematerializable value is a good candidate for spilling since the value can be rematerialized, and thus does not need to be stored and reloaded.

The final step of RENUMBER eliminates the phi functions added during static single assignment by examining each operand of a phi statement and either unioning the operand with the result of the phi statement or by inserting a copy statement, known as a *split* instruction, in one of the predecessor blocks of the block containing the phi function. This step unions live ranges which are rematerializable by the same instruction, or live ranges which not rematerializable into a single live range.

The BUILD phase constructs the interference graph by creating a node for each live range in the procedure.

¹Chaitin et al. [14] called these *never-killed* values.

Edges are added by considering each definition as it is encountered during a backward scan of each basic block. Live variable analysis is necessary in order to know which virtual registers are live on exit from each basic block. When a definition, $v_i = \dots$ is encountered during the backward scan, an edge is added between the node which corresponds to the live range of v_i and the nodes of any live ranges that are currently live.

The COALESCE phase eliminates copy statements. The operands of a copy statement are coalesced, and the copy statement is eliminated if (1) the live ranges do not interfere, and (2) for copies introduced during renumbering, combining them will not cause the resulting live range to be spilled. The second constraint makes the coalescing *conservative* and prevents the coalescing stage from eliminating the copy statements that were deliberately added to enable rematerialization.

The SPILL COSTS stage determines the costs that will be used in selecting which nodes should be spilled when a coloring can not be found. The SIMPLIFY stage removes each node from the interference graph and pushes it onto a stack for coloring. Initially, nodes with degree less than the number of registers are removed from the interference graph since these nodes certainly will be colorable. Then, the node with the least spill cost is removed and pushed onto the stack. This process continues until there are no nodes left in the graph. During the SELECT stage, each node is popped from the coloring stack and inserted into the interference graph along with its edges. The node is then assigned a color that is different from all of its neighbors. If the node is not colorable, it is pushed onto a stack of nodes for which spill code is to be inserted.

The SPILL CODE stage inserts spill and/or rematerialization code into the intermediate code for each node that was not colored by the SELECT stage. The live range for a spilled variable is replaced by a set of smaller live ranges where each live range begins at the point of the load and ends with the use, or begins at the point of the definition and ends with the store. The interferences and the set of edges that can be added must be recalculated, and thus the process begins again with the RENUMBER stage. If all nodes are colored during the execution of the SELECT stage, the process terminates.

4.2 Adding Scheduler-Sensitivity to the Allocator

The most significant changes made to OA to incorporate sensitivity to local instruction scheduling were made to the BUILD and SIMPLIFY phases. The goal in doing the scheduler-sensitive allocation is for the allocator to add only those dependences which would not interfere with the code scheduling. Thus, the BUILD phase builds the interference graph while considering whether adding a dependence between two instructions would hinder code scheduling. The SIMPLIFY phase makes tradeoffs between adding data dependences to eliminate interferences between variables and spilling. This section discusses these modifications in detail.

4.2.1 A Dag-Driven BUILD

In the BUILD of OA, the interferences are determined assuming that the given code order represents the final order of the instructions to be executed. In the context of local code scheduling, the interference graph should reflect whether two values interfere given any legitimate code reordering. For this reason, SSG builds the interference graph based on the data dependence dag which reflects all legitimate code reorderings for each basic block.

In Figure 7(a), assume that b , c , f , and g are live on entrance to the basic block and that d , h , and g are live on exit from the basic block. Figure 7(c) contains the interference graph for the code built by the BUILD stage of OA which assumes a fixed ordering of the code. Figure 7(d) contains the interference graph which would be built by SSG which reflects interferences for all legal code reorderings of the basic block. The interference graph in Figure 7(d) contains all of the interferences of Figure 7(c) plus 6 other interferences from other legal code reorderings.

Consider, for example, the interferences with virtual register, a . Assuming the statements are to be executed in the original code order in Figure 7(a), virtual register a interferes only with virtual registers g , f , and c since each of these is live at the point that a is defined. Thus, Figure 7(c) contains an interference edge between a and each of g , f , and c . The BUILD of SSG also contains these interferences since executing the statements in the order $S1$, $S2$, $S3$ and $S4$ is one legal order in which to execute these statements;

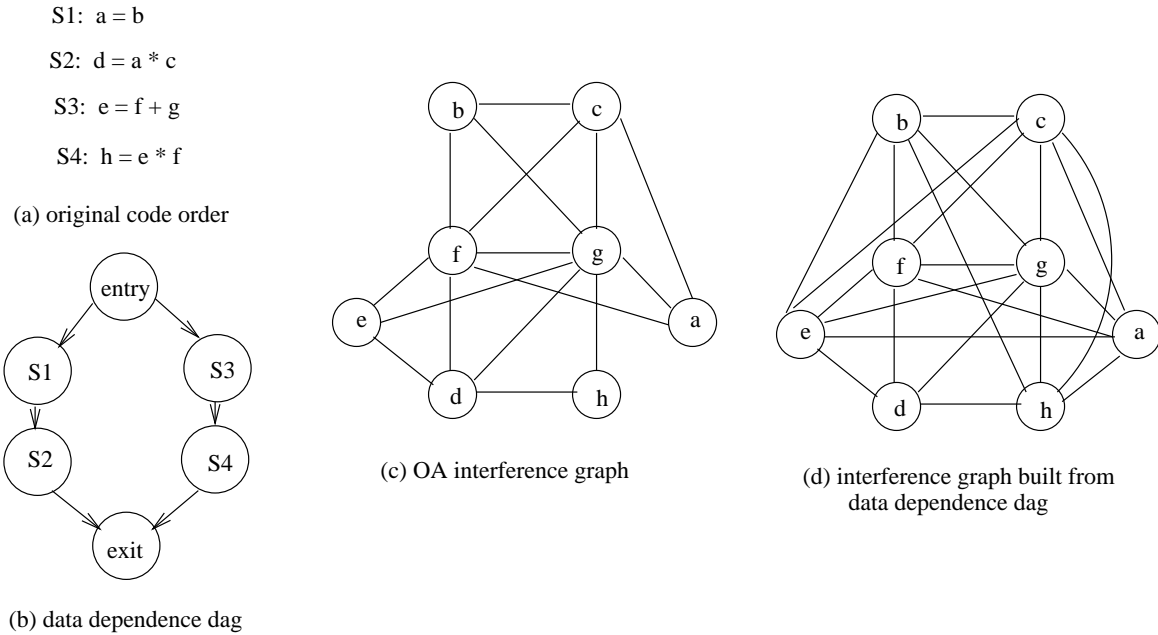


Figure 7: Data dependence dag and interference graphs

however, SSG also adds interferences for all other legitimate reorderings. For example, the statements could be executed in the order: $S1, S3, S2, S4$ which would result in variables a and e being simultaneously live. Thus, the interference graph in Figure 7(d) reflects that a and e interfere because they interfere in some legitimate ordering of this code. If this larger interference graph can be colored as it is, then the allocation can be performed without introducing any false dependences, and the instruction scheduler will be free to execute the code in any legal reordering.

Although the interference graph built for the dependence dag would reflect the maximum freedom of code reordering by the code scheduler, the increased number of interferences will make the register allocator's task more difficult as it will be less able to color the larger graph. The ideal situation would be to accurately predict the ordering that the scheduler would select and only include the interference edges that reflect that ordering. However, it is desirable to avoid multiple invocations of the scheduler. An alternative is to remove interference edges from the graph built from the dependence dag if the interference edges correspond to code orderings that the code scheduler would least likely select. In particular, an interference between two variables, v_i and v_j , can be removed by adding dependence dag edges to force all uses of v_i to be ancestors

of the definition of v_j , or vice-versa. If all uses of v_i are ancestors of the definition of v_j , then the statement with uses of v_i must be executed before the statement with the definition of v_j , and thus, v_i is dead before v_j is defined. Thus, in any code ordering that abides by the modified dependence dag, these variables are not simultaneously live and do not interfere. However, the added dag edges represent a restriction being put on the code reordering that did not hold in the original code. Also, some interferences are impossible to eliminate by the addition of dag edges either because one or both of the values are global or because adding edges would add a cycle to the dag.

When adding dag edges to restrict the legal code orderings to reduce the number of interferences, three concerns must be addressed. First, it would be best to avoid adding edges which would force the scheduler to schedule instructions in an order which it would not choose if those edges were not added. If edges are added from the uses of v_i to the definition of v_k , then these edges are said to be *scheduler-sensitive* if the scheduler would choose to schedule the instructions containing the uses before the instruction containing the definition anyway. Second, it is desirable to avoid adding edges to eliminate interferences when there are already sufficient physical registers available to meet the register demand. If the register demand of a basic block does not exceed the number of physical registers, then it is unnecessary to add dag edges for that basic block. Third, it is desirable to add dag edges in a way that adds the least amount of overhead at compile time. Our scheduler-sensitive BUILD builds the interference graph using the dependence dag but adds dependence dag edges to reduce the number of interferences by addressing each of these concerns.

The first concern, predicting the scheduler's decisions, is addressed by attempting to anticipate the instruction order that will be selected by the scheduler. A significant factor used by the scheduler in determining the execution order of the basic block is the cumulative cost of nodes in the dependence dag for that basic block. The cumulative cost of a node n in the dag is the minimum number of machine cycles required to execute the subgraph of the dag rooted at n . If the cumulative cost of all uses of a particular variable v_i is much greater than the cumulative cost of the definition of some other variable v_k , then SSG assumes that the scheduler will indeed choose to schedule the instructions containing the uses of v_i prior to the instruction containing the definition of v_k .

The second concern, avoiding unnecessary addition of dag edges, and thus unnecessary restrictions on the scheduler, is addressed by estimating the register demand of a basic block. The demand is estimated as the sum of the number of variables live on entrance and the number of definitions within the basic block. Since this information is available before register allocation begins, dag edges can be added to a basic block in the early phases before the interference graph is even built.

The third concern, minimizing the overhead added to the compiler, is addressed by adding dag edges during the point in the allocation where doing so is cheapest in compile time. Dag edges can be added at three different points (in order of increasing compile time expense): prior to BUILD, during BUILD and during the SIMPLIFY stage. In order to decrease the cost of identifying the interferences that can be eliminated by dag edges during the SIMPLIFY phase, we determine during the process of building the interference graph whether an interference can be eliminated by adding dag edges.

The *ddBUILD* Algorithm. Figure 8 contains the algorithm, *ddBUILD*, for the dag-driven BUILD phase. The *ddBUILD* algorithm is called for each basic block in the program. Each of the statements in the basic block is examined in sequential order; $S_{i_{def}}$ is the virtual register defined by the current statement, S_i . First, each virtual register, v , that is live on entrance to the basic block is examined to check for an interference between v and $S_{i_{def}}$. If a virtual register, v , is live on entrance and on exit, then it is not possible to make the current statement S_i be a descendant of all of the uses of v , and thus not possible to eliminate an interference between $S_{i_{def}}$ and v . If v is not live on exit, but is live on entrance, the procedure *check_dag_edges* is called to determine whether $S_{i_{def}}$ and v interfere, and if so, whether the interference can be eliminated by adding edges to the dag for the current basic block. Next, the algorithm checks for an interference between $S_{i_{def}}$ and each variable defined by previously examined statements in the same basic block.

If procedure *check_dag_edges* has determined that two virtual registers do interfere and that the interference can be eliminated by adding dag edges, the function *update_structures* is executed to add the necessary dag edges if desired at this stage in register allocation. If the desired strategy is to eliminate the interference now rather than adding it to the interference graph (these strategies are discussed in section 4.2.2), then the *DuringBuild* routine will be called to add dag edges so that the interference need not be added to the

```

Procedure ddBUILD(bb, dag, G, edge_block, DB_strategy) {
  /** Input: basic block bb
    data dependence graph, dag, of basic block bb
    interference graph G initially containing nodes and interferences of previously examined basic blocks
    edge_block matrix to indicate what interferences can be eliminated
    DB_strategy is true if dependence dag edges should be added while building the interference graph
  Output: modified interference graph G
    possibly modified dag to eliminate interferences
    possibly modified edge_block matrix
    livein = set of virtual registers live on entrance to bb
    liveout = set of virtual registers live on exit from bb
    S = set of n statements in bb, ith statement in S is Si
    Si_def = virtual register defined by ith statement
    def(v) = statement which defines v in bb /**/
    deflist =  $\emptyset$ 
    for i = 1 to n { /* scan statements in bb */
      for each v  $\in$  livein {
        if (v  $\in$  liveout) add interference to G between Si_def and v
      } else {
        check_dag_edges(dag, v, Si, interfere, can_eliminate)
        if (interfere && can_eliminate)
          interfere = update_structures(dag, v, Si, bb, DB_strategy, edge_block)
        if (interfere) add interference between Si_def and v to G
      } }
    } }
    for each v  $\in$  deflist {
      interfere = true
      if (v  $\notin$  liveout) { /* check if v's uses can be made ancestors of Si */
        check_dag_edges(dag, v, Si, interfere, can_eliminate)
        if (interfere && can_eliminate)
          interfere = update_structures(dag, v, Si, bb, DB_strategy, edge_block)
        }
      if (interfere && Si_def  $\notin$  liveout) { /* check if uses of Si_def can be made ancestors of def(v) */
        check_dag_edges(dag, Si_def, def(v), interfere, can_eliminate)
        if (interfere && can_eliminate)
          interfere = update_structures(dag, Si_def, def(v), bb, DB_strategy, edge_block)
        }
      if (interfere) add interference between v and Si_def to G
    }
    deflist = deflist  $\cup$  Si_def
  } }
} }

```

Figure 8: Dag-driven BUILD

interference graph. If the interference is not eliminated at this time by adding the dag edges, but has been shown that it could be eliminated, then $edge_block[v][S_{d_{ef}}]$ is set to the number of the basic block in order to indicate that the interference between v and $S_{d_{ef}}$, the virtual register defined at S_d , can be eliminated by adding dag edges from the uses of v to S_d although it is not desirable to do so at this time. The function $update_structures$ returns true if an interference edge should be added to the interference graph because dag edges were not added to eliminate the interference.

An Example. This section details an example of how $ddBUILD$ handles the addition of dag edges to eliminate the interference between two virtual registers. Consider Figure 9 and assume that the variables b, c, f , and g are in the *livein* set of this basic block, and d, h , and g are in *liveout*. Suppose the statement currently being examined by $ddBUILD$ is $S2$. The algorithm first examines the variables that are in *livein*. Clearly, d does not interfere with either b or c since in any legal execution ordering of the statements, statement $S1$ must precede statement $S2$ forcing all uses of b and c to occur before d is defined. However, d and f do interfere since the statements could be executed in the order $S1, S2, S3, S4$ which would make f live when d is defined. But if the statements are executed in the order $S1, S3, S4, S2$, then the interference between d and f is eliminated since f would then be dead before d is defined. In Figure 9 (b), the edges added from nodes $S3$ and $S4$ to node $S2$ eliminate the interference between d and f since this forces the statements which use f ($S3$ and $S4$) to be executed before the statement which defines d . These edges may be added at one of three different times during the allocation as discussed later. The interference between d and g can not be eliminated by the addition of dag edges since g is both live on entrance and live on exit and thus, is live in any legal execution order when d is defined.

After each of the variables in the *livein* set is considered by $ddBUILD$, the variables defined by previous statements in the basic block are examined. Again, consider Figure 9 and now assume that the statement currently being examined by procedure $ddBUILD$ is statement $S3$. The *deflist* contains the virtual registers, a and d , defined by statements $S1$ and $S2$. By the dag in Figure 9(a), there is an interference between a and e since the statements could be executed legally in the order $S1, S3, S2, S4$ which would make a live at the point that e is defined. In Figure 9(b), this interference has been eliminated by adding an edge from $S4$ to

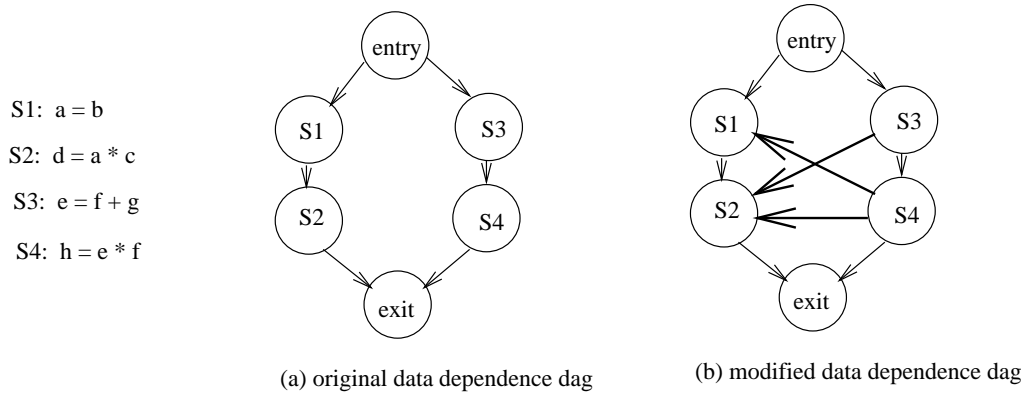


Figure 9: Eliminating interferences between d and f , and e and a

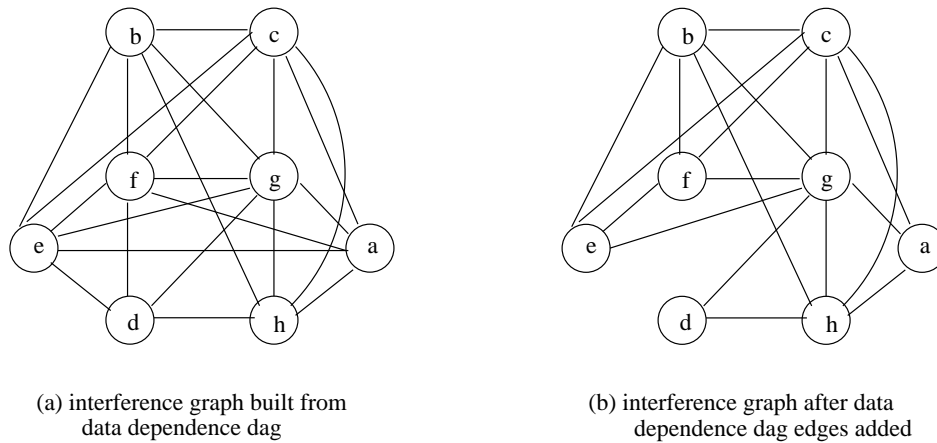


Figure 10: Interference graph before and after adding data dependence edges

$S1$. This forces the statements to be executed in the order $S3, S4, S1, S2$ for which there is no interference between a and e .

Figure 10(a) contains the interference graph built from the data dependence dag in Figure 9(a). There is an edge between any two nodes in Figure 10(a) if the corresponding virtual registers could be simultaneously live in any legal code reordering. The interference graph in Figure 10(b) was built from the data dependence dag in Figure 9(b). The edges from nodes $S3$ and $S4$ to node $S2$ were specifically added to the dag in Figure 9(b) to eliminate the interference between f and d . The edge from node $S4$ to $S1$ was specifically added to eliminate the interference between e and a . Other interferences were eliminated as well by the

addition of these edges. The dag edge from $S4$ to $S2$ eliminates the interference between e and d since the statement containing the only use of e ($S4$) would then be executed before d is defined. Also, the dag edge from $S4$ to $S1$ eliminates the interference between a and f because the statements containing the uses of f ($S3$ and $S4$) would then be executed before a is defined. No other dag edges can be added to eliminate interferences and the remaining interferences can only be eliminated via spilling. Although a smaller interference graph results, the addition of these dag edges means that the code can be executed in only one order: $S3, S4, S1, S2$.

4.2.2 Strategies for When to Remove Interferences

Three strategies for when to add dag edges to eliminate interferences were identified and experimentally evaluated. The first strategy, called *No Edges*, is to add no dag edges in the BUILD phase. The advantage of this technique is that edges will only be added as necessary to avoid spilling a node in the SIMPLIFY phase. The disadvantage is that the interference graph will contain more interferences than if dag edges were added in the BUILD phase, and in addition, all added dag edges will be added during the SIMPLIFY phase where the compilation expense is higher.

The second strategy, called *Before BUILD*, is to add dag edges prior to actually constructing the interference graph in addition to adding dag edges during SIMPLIFY. The dag edges added at this point in the allocation are the cheapest edges in terms of compilation expense. Generally, after an edge is added, the transitive closure is computed in order to prevent the addition of edges forming a cycle in the dag. Also, the cumulative costs of nodes within the dag can change, requiring the recomputation of those costs. The edges to add to a dag prior to building the interference graph are selected such that their addition does not require recomputation of either the transitive closure or the cumulative costs.

In particular, the *Before BUILD* strategy is to identify the basic blocks where the estimated register demand exceeds the number of available registers and add edges from all m statement nodes, S_{ui} , $i = 1..m$, which use variable, v_k , to a statement node, S_d , which defines a variable, v_i , when the following conditions hold:

1. The cumulative cost of each S_{ui} , $i = 1..m$ is greater than the cumulative cost of S_d . This restriction

adds only scheduler-sensitive edges, and also prevents the need to recompute cumulative costs.

2. The statement number of S_{ui} , $i = 1..m$ is less than the statement number of S_d in the original sequential ordering of the basic block. This restriction prevents the need to recompute the transitive closure between the addition of edges because adding edges that meet this requirement can not create a cycle in the dependence graph.

The disadvantage of the *BeforeBuild* approach is that it may add unnecessary dag edges because the actual register demand within the basic block may be less than the number of physical registers. Estimating the register demand is an attempt to prevent the addition of unnecessary dag edges.

The third strategy, called *Before & During BUILD*, is to add dag edges while the interference graph is being built in addition to adding edges prior to BUILD by the *BeforeBuild* strategy. The cheapest scheduler-sensitive dag edges in terms of compile time expense are the ones added before BUILD. However, there could still be scheduler-sensitive edges left in the dag that do not cause a cycle in the dag, but are directed from a later statement to an earlier statement in the original code sequence, (i.e., Condition 2 does not hold.). If edges are added that do not satisfy condition 2, then the transitive closure must be computed between the addition of sets of these edges to insure that dag edges to be added in the future will not create a cycle in the dependence graph. The transitive closure operation is performed incrementally for increased efficiency. If an edge is added from a node a to node b , then a and the ancestors of a are made to be the ancestors of b and b 's descendants.

4.2.3 A Scheduler-Sensitive SIMPLIFY Phase

Figure 11 contains the algorithm for the scheduler-sensitive SIMPLIFY phase. The interference graph is first simplified by removing each node which has fewer than k neighbors where k is the number of physical registers. When no more nodes can be removed from the interference graph in this way, SSG first finds the node with the least spill cost and then has two choices: (1) it can remove a node from the graph with least spill cost or (2) it can add edges to one of the basic block dags in order to eliminate enough interferences to

```

stack Function Simplify( $G, DAGs, edge\_block$ ){
/**
Input: interference graph  $G$ 
        data dependence graphs,  $DAGs$ , for each block in procedure
         $edge\_block$  matrix to indicate what interferences can be eliminated by dag edges
Output: stack of nodes,  $node\_stack$ 

         $k$  = number of physical registers
         $degree(n)$  = degree of node  $n$  in  $G$ 
***/
while  $G$  not empty {
  if ( $\exists n \in G$  such that  $degree(n) < k$ ) {
    remove  $n$  from  $G$ 
    push( $n, node\_stack$ )
  } else {
     $n = n \in G$  such that  $\forall m \in G \neq n, cost(n)/degree(n) \leq cost(m)/degree(m)$ 
    remove  $n$  from  $G$ 
    if (SpillNode_strategy)
       $addedges = addDAGedges(n, DAGs, G, edge\_block)$ 
    if (MaxNode_strategy) {
       $maxnode = GetMaxNode(edge\_block)$ 
       $addedges = addDAGedges(maxnode, DAGs, G, edge\_block)$ 
    }
    if (CheapEdges_strategy)
       $addedges = CheapEdges(DAGs, G, edge\_block)$ 
    if (not  $addedges$ )
      push( $n, node\_stack$ )
  }
}
return  $node\_stack$ 
}

```

Figure 11: Scheduler-sensitive simplification of the interference graph

continue simplifying the interference graph. Three different strategies for selecting which dag edges should be added at this point during the SIMPLIFY stage were developed and experimentally evaluated. These strategies are called *SpillNode*, *MaxNode*, and *CheapEdges*. First, the measure that is used in comparing different candidate dag edge additions is described.

Any dag edges that increase the cumulative cost of the root of the dependence graph will increase the minimum number of cycles required to execute the basic block. For this reason, the cost associated with each candidate dag edge addition is defined to be one plus the difference between the cumulative cost of the sink and the cumulative cost of the source of the potentially added dag edge. (One is added to account for issuing the sink instruction along this new path.) If this cost is nonpositive, then the cumulative cost of the root of the dag will not be changed by adding the dag edge. Note that, in general, a set of dag edges, although generally a small set, is added to eliminate a single interference. The cost of the set of dag edges to be added is the maximum cost of adding any edge in the set.

When an interference has been eliminated by adding edges to a data dependence dag, the interference is permanently eliminated. Normally, during the SELECT stage, the interference graph is rebuilt while the

nodes are being colored, and each of the interference graph edges of the original graph are reinserted into the interference graph. But, if an interference is eliminated by adding dag edges, the interference is not reinserted into the interference graph in the SELECT stage. The addition of the dag edges represents an ordering of the instructions in which the virtual registers involved in the eliminated interference will not be simultaneously live.

The *SpillNode* strategy adds dag edges to eliminate interferences with the node in the interference graph which has been identified to have the least spill cost. The objective of this strategy is to avoid spilling the node by adding dag edges. If it is not possible to eliminate interferences with the spill node so that the resulting degree of the spill node is less than the number of physical registers, then no dag edges will be added. In that case, the spill node is simply removed from the interference graph and pushed onto the stack for coloring. Otherwise, sets of dag edges are added to eliminate interferences with the spill node in order of increasing cost until the degree of any node in the interference graph becomes less than the number of available registers. Note that it may happen that the degree of another node in the interference graph, in particular, a neighbor of the spill node, becomes less than the number of available registers before this happens to the spill node. By focusing on the spill node when adding dag edges to eliminate interferences, this strategy avoids any cost of finding a better candidate for eliminating interferences. However, it may not be as important to add edges to reduce the degree of the spill node since it is the cheapest to spill.

The *MaxNode* method attempts to find a better interference graph node candidate on which to focus while eliminating interferences. The node selected, the *MaxNode*, is the interference graph node that the *edge_block* matrix identifies as having the greatest number of interferences that can be eliminated by adding dag edges. This is not necessarily the node with the highest degree. A node may have a high degree, but it may be impossible to reduce the degree of the node by adding dag edges. Again, the sets of dag edges are added in order of increasing cost until there are no more edges to add or the degree of any node in the interference graph becomes less than the number of available registers. By focusing on *MaxNode*, there is more potential for causing a node to become colorable than the *SpillNode* approach. However, there is a small cost incurred for identifying this node.

In contrast to the other two strategies, the *CheapEdges* strategy does not focus on a particular interference graph node. Instead, it selects sets of dag edges that can eliminate interferences in order of cheapest to most expensive cost. Sets of dag edges continue to be added until there are no more edges to add or the degree of a node in the interference graph becomes less than the number of available registers. If no edges can be added, then function *Simplify* removes from the interference graph the node with the least spill cost. This method has the potential of adding edges with the least cost, but the possibility that spilling a node may be a better action is not considered in this method, so it may choose to add too many edges when spilling may have been more advantageous.

4.2.4 Other Modifications

A slight modification to RENUMBER was found to cause a significant impact on the overall outcome. The final step of the RENUMBER stage of OA is to eliminate phi functions by examining each operand of each phi function and either coalescing or inserting a split instruction. In order to prevent adding anti-dependences, the final phase of SSG's RENUMBER stage does not perform the coalescing; instead, a split instruction is always inserted. These copy statements may be removed later by the COALESCE stage if they do not add an anti-dependence.

In addition, the SPILL COSTS phase can be made to be scheduler-sensitive by incorporating the time to execute instructions into the weights of the spill cost estimates. A memory access is generally several cycles slower than the time to execute an instruction to rematerialize a value. That is, the load instruction prior to the use of a variable that has been spilled will cause more delay than the rematerialization instruction prior to the use if the value were rematerialized. To reflect this, the original allocator's cost estimate for rematerializing a node is used, but the spill cost estimate for spills into memory is slightly modified to reflect the increased cost of doing a memory access. We called this our *Weighted Chaitin* spill costs strategy and in the experimental study we compared this strategy to the *Chaitin* strategy [14] where the spill costs are not weighted.

5 Experimental study

5.1 Experimental setup

In this section, we report on our results of comparing the performance of the cooperative schemes to the performance of the postpass scheduling approach where a conventional register allocator is followed by a local code scheduler (BBSCH) based on the Gibbons and Muchnick [21] basic block scheduling algorithm. In order to do this comparison, the *optimistic allocator* (OA) of Briggs, Cooper, and Torczon [12] was implemented to perform the conventional register allocation. The registers are allocated using a *round robin* approach, because experimental results indicate that round robin provides a better allocation in the context of code scheduling and global register allocation than a *first fit* approach.

The front end of the compiler is *pdgcc*, developed at the University of Pittsburgh, which accepts as input C source code and outputs the corresponding PDG. The PDG can then be used as input to RASER, a standard region scheduler (REGN), or *pdg2i* which generates a low level intermediate code in the cases where no global scheduling is performed. REGN 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. REGN, RASER, and RASSG all perform loop peeling and attempt to unroll the code enough to match the parallelism in the architecture. In addition, code is being moved in order to increase the parallelism in regions with insufficient parallelism. The transformed PDGs are 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. Iloc code is generated assuming an infinite number of virtual registers.

The register allocators, SSG and OA, both accept iloc code and the number of physical registers as input and map the virtual registers onto the set of physical registers, rewriting the code in terms of these real registers with spill code added as necessary. SSG may also reorder the code to reflect dag edges added during register allocation. SSG also accepts as input an opcode file indicating the degree of pipelining of the

target architecture.

A simulator/interpreter is used to determine the number of cycles required to execute the iloc code. The iloc simulator takes as input the iloc intermediate code program and an opcode file which indicates the degree of pipelining of the target architecture. Our simulator does not simulate a complex memory system, but instead uses a fixed cycle time for all load's. We believe that this is sufficient for our purposes in comparing *relative* performance of the different allocation/scheduling schemes of interest. The local scheduler in our system is the same as a traditional local scheduler. The register allocator is modified to be scheduler-sensitive by performing register allocation in a way that seeks to add only those false dependences (created by separate register allocation and instruction scheduling) which would not interfere with the code scheduling goals. The register allocator is giving more freedom to the scheduler in its decision-making. The relative performance of this approach versus the traditional register allocation/scheduler should not see much impact by a complex memory system since the scheduler is given more freedom in possible reorderings.

As a global instruction scheduler, the performance of region scheduling itself can indeed be impacted by the memory system. The real question is whether the relative performance of RASER versus region scheduling can be greatly affected by the memory system. The modifications to region scheduling to create RASER involve live value reduction, and allocation-sensitive region scheduling transformations. Live value reduction seeks to reduce spilling by moving code from regions sure to have spilling to regions that have less live variables than the number of physical registers. The definitions of variables are moved closer to uses. This transformation should not be greatly affected by the memory system as we are increasing register pressure in any region above the number of physical registers. The allocation-sensitive scheduling transformations are the same as region scheduling transformations, but are sensitive to the number of live variables in respective regions of interest, and are not applied in some cases when this number would increase past the number of physical registers. Thus, we are not performing more transformations than region scheduling, but rather less. The memory system will have no greater effect on RASER than it does on region scheduling, so the relative performance results should not be greatly impacted by the memory system. If they are affected, it should actually be the case that the performance of our techniques would look even better than what we

have reported from our studies.

Performance measurements have been taken for 13 of the Livermore Loops, the cLinpack routines, a heapsort implementation and some of the Stanford routines. The performance of each scenario was calculated by $totalcycles(Postpass)/totalcycles(scenario)$ where $totalcycles$ is the number of cycles required to execute the iloc code as determined by an iloc simulator.

Two separate sets of experiments were performed, one to investigate the different strategies developed for the design of SSG, and one to investigate the various forms of cooperative register allocation and local and global code scheduling. Sections 5.2.1 and 5.2.2 describe these experiments in more detail.

5.2 Results

5.2.1 Comparative Study of the SSG Strategies

For the first experiment, a total of 18 different scenarios of scheduler-sensitive register allocation were studied, comparing each scenario of SSG to Postpass scheduling. In Table 1, each of these strategies is denoted by a name xyz where x is either n for *No Edges*, b for *Before BUILD*, or bd for *Before & During BUILD*, y is either c for *Chaitin* or w for *Weighted Chaitin* spill costs, and z is either c for *CheapEdges*, s for *SpillNode*, or m for *MaxNode*. For example, ncm denotes the strategy of using *No Edges*, *Chaitin*, and *MaxNode* in combination.

Table 1 presents the speedups over Postpass for each of the 18 different strategies assuming 10, 15, 20, 25 and 30 available physical registers. In performing the experiments, we assumed a hypothetical medium-pipelined machine in which an add takes 2 cycles, a multiply takes 3 cycles and a load takes 6 cycles. We discuss the most prominent results of the experimental studies here.

First, adding edges before the construction of the interference graph increases the speedup of the generated code over not adding edges before BUILD when used in conjunction with *SpillNode* and *MaxNode*. When dag edges are added during the SIMPLIFY phase, these added dag edges may eliminate interferences other than the one that the technique is trying to eliminate because these edges may also cause the uses of another

Table 1: Average speedup of SSG over postpass

Strat	Number of Registers					Avg
	10	15	20	25	30	
ncc	1.050	1.228	1.253	1.200	1.169	1.180
ncs	0.960	1.186	1.246	1.195	1.162	1.150
ncm	1.039	1.218	1.183	1.123	1.132	1.139
nwc	1.115	1.232	1.262	1.200	1.169	1.196
nws	0.996	1.163	1.229	1.159	1.138	1.131
nwm	1.033	1.249	1.185	1.147	1.132	1.149
bcc	1.048	1.202	1.249	1.198	1.169	1.173
bcs	1.025	1.232	1.247	1.165	1.169	1.168
bcm	1.123	1.245	1.243	1.197	1.170	1.196
bwc	1.097	1.259	1.255	1.198	1.169	1.196
bws	1.135	1.261	1.252	1.186	1.168	1.200
bwm	1.152	1.263	1.247	1.197	1.170	1.206
bdcc	1.050	1.211	1.246	1.194	1.169	1.174
bdcs	0.948	1.205	1.247	1.192	1.169	1.152
bdcm	1.163	1.251	1.260	1.198	1.170	1.208
bdwc	1.124	1.254	1.255	1.194	1.167	1.199
bdws	1.101	1.248	1.248	1.196	1.169	1.192
bdwm	1.100	1.232	1.232	1.198	1.170	1.188

value to become ancestors of a different, or the same, definition. Because SIMPLIFY does not account for elimination of these interferences, it may add more dag edges than necessary to make a node colorable.

In contrast, the *CheapEdges* selection of dag edges to add in SIMPLIFY and the *Before BUILD* strategy of adding dag edges in BUILD try to add essentially the same edges, those with the cheapest cost, throughout the entire dag. Since *Before BUILD* adds these edges before the SIMPLIFY phase is reached, the cheap edges added during the SIMPLIFY stage generally will be of positive cost.

An important result of adding edges *Before BUILD* or *Before & During BUILD* is the subsequent reduction in the number of conflicts in the interference graph. The number of interferences in the interference graph was recorded during the first iteration of the SSG algorithm. The *Before BUILD* strategy reduces the number of interferences in the interference graph by 21% on average over *NoEdges*. The *Before & During BUILD* strategy reduces the number of interferences by 27% on average. Thus, a large reduction in the number of interferences can be achieved by these two strategies, which causes a significant savings in the memory required to maintain the interference graph.

As expected, using the *Weighted Chaitin* method for determining spill costs yields a greater speedup than

calculating spill costs by Chaitin’s method, because *Weighted Chaitin* eliminates more references to memory by favoring rematerialization over spilling.

The *CheapEdges* strategy used in conjunction with *No Edges* provides a greater speedup than using *No Edges* with either *SpillNode* or *MaxNode*. *CheapEdges* will add dag edges of positive cost only after all the negative cost edges have been added. Using *SpillNode* or *MaxNode* in conjunction with *No Edges* does not allow for eliminating interferences by adding these cheap dag edges.

The *MaxNode* parameter used in conjunction with *Before BUILD* provides a greater speedup than using *Before Build* with either *SpillNode* or *CheapEdges*. Adding dag edges via the *SpillNode* strategy severely limits the number of dag edges that can be added, because it will only add dag edges that will eliminate interferences with the node with least spill cost. The *CheapEdges* strategy represents the other extreme in that it could eliminate interferences with any node in the interference graph with degree greater than the number of registers. Also, the least cost edges that *CheapEdges* could add are a superset of the edges already added *Before BUILD*. The *MaxNode* strategy falls somewhere between these strategies.

The *SpillNode* strategy causes the least amount of spill code to be added. Instead of spilling a node, it attempts to add dag edges deliberately to avoid spilling that node. This results in less overall spill code being inserted, but this does not however result in the most efficient generated code since spill code can be executed in otherwise unused cycles. Most of the strategies cause some spilling in loops 1, 2, 7, 8, 9, and 10 given 10 registers. In the 20 register case, spill code is inserted in only loops 2 and 8. Thirty registers is sufficient for performing an allocation on any of the loops without spilling. Note that even when provided sufficient registers to perform an allocation without spilling, a cooperative strategy still results in the generation of more efficient code.

Figure 12 graphically depicts the average speedups over Postpass for the 18 SSG strategies. The strategies that performed the worst and the best overall are labeled. Except for a few strategies with 10 available registers, all of the strategies outperformed Postpass. Although the performance of the different strategies varied widely for 10 registers, they all hovered around a speedup of 1.17 for 30 registers. Most of the speedups disregarding the 10 register case were between 1.10 and 1.26. The speedups for most of the strategies were

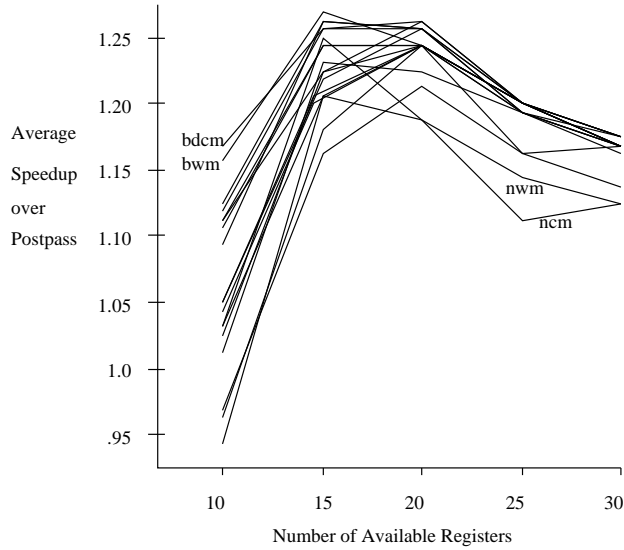


Figure 12: Average speedup of SSG over Postpass

the greatest for either 15 or 20 registers. The best average speedup achieved by any of the strategies was 1.26, which was attained by *bwm* for 15 registers.

The two strategies which provide the best average speedup over all numbers of registers were two *MaxNode* strategies, *bdcn* and *bwm*. In order to compare scheduler-sensitive register allocation to other cooperative approaches, the *bwm* strategy was selected which is to add dag edges *Before BUILD*, use *Weighted Chaitin* spill cost estimates, and select dag edges to add during *SIMPLIFY* based on *MaxNode*. The *bwm* was chosen for the following reasons: (1) Adding dag edges before the interference graph is constructed significantly reduces the size of the interference graph that is built. (2) The *Weighted Chaitin* spill cost estimate in general yielded an increase in speedup over Chaitin’s spill cost calculation. (3) Although the *bdcn* strategy achieved a slightly higher average speedup over *bwm*, *bdcn* is more costly in terms of compile time, because adding edges during the *BUILD* can not be done as efficiently as adding edges before *BUILD*. Table 2 contains the speedups of the *bwm* strategy over Postpass calculated for each benchmark.

loop	Number of Registers		
	10	20	30
1	1.514	1.304	1.156
2	1.232	1.436	1.205
3	0.956	1.068	1.061
4	1.230	1.333	1.311
5	1.021	1.120	1.120
6	1.320	1.063	1.000
7	1.026	1.653	1.533
8	1.085	1.284	1.453
9	0.857	1.347	1.090
10	1.060	0.993	1.008
11	1.247	1.107	1.107
12	1.277	1.259	1.000

Table 2: Speedup of bwm over Postpass

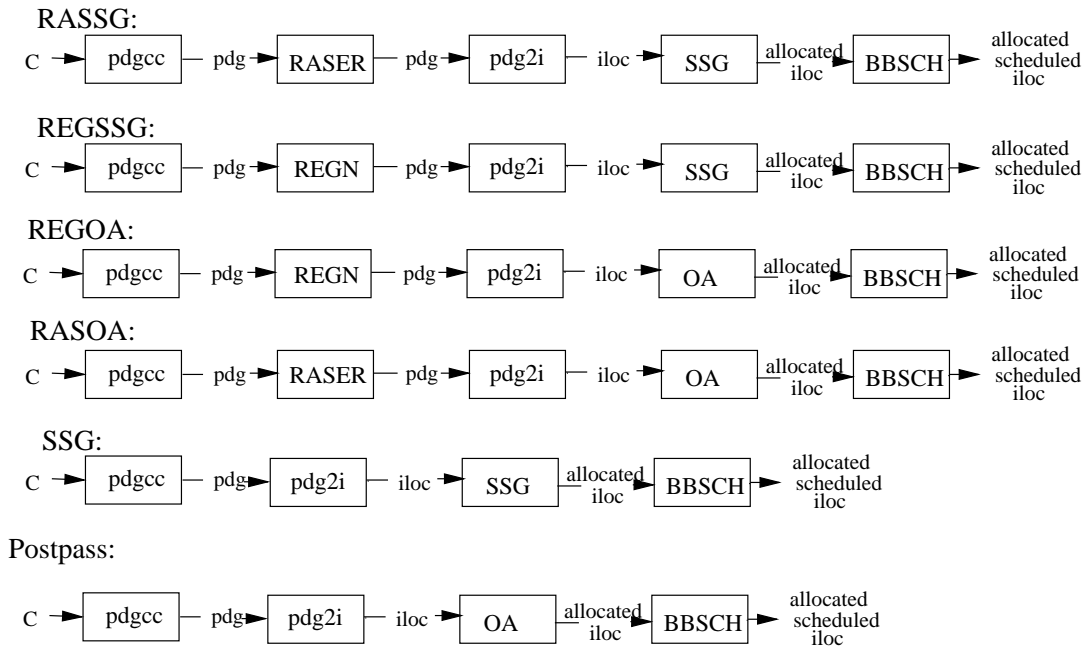


Figure 13: Overview of implementations

5.2.2 Comparing the Overall Cooperative Strategies

Figure 13 contains an overview of the second set of experiments performed. The RASSG experiment represents fully cooperative register allocation with local and global code scheduling, executing global scheduling via RASER, register allocation via SSG and the local code scheduler, BBSCH. The SSG scenario selected is the bwm scenario which was determined via the first experiment to perform among the best of the SSG scenarios. The REGSSG experiment executes global scheduling via standard region scheduling, register allocation via SSG and the local code scheduler, BBSCH. The REGOA experiment executes standard region scheduling, register allocation via OA, and the local code scheduler, BBSCH. The RASOA experiment executes global scheduling via RASER, register allocation via OA, and the local code scheduler, BBSCH. The SSG experiment executes SSG and BBSCH; global instruction scheduling is not executed. The Postpass experiment executes uncooperative register allocation and local instruction scheduling. The results of the Postpass experiment are used to calculate the speedups of the other scenarios.

Table 3 contains a table of the speedups calculated for the RASSG and the REGOA experiments for 8 and 16 registers assuming hypothetical medium and highly-pipelined machines. For the medium-pipelined machine, we assumed that an add takes 2 cycles, a multiply takes 3 cycles, and a load takes 6 cycles. For the highly-pipelined machine, we assumed an add takes 4 cycles, a multiply takes 6 cycles and a load takes 12 cycles. The RASSG scenario represents a fully cooperative scheme where the global instruction scheduler cooperates with register allocation and the register allocator cooperates with the local instruction scheduler. The REGOA scenario represents a fully uncooperative scheme where global instruction scheduling, register allocation and local instruction scheduling are all performed, but none are cooperative. Speedups for REGOA range from .68 to 2.66 for the highly-pipelined architecture and .57 to 2.38 for the medium-pipelined architecture. Speedups for RASSG range from .35 to 3.34 for the highly-pipelined architecture and .40 to 2.49 for the medium-pipelined architecture. Of the 124 RASSG entries in the table, 89 of the entries represent cases in which the RASSG scheduled and allocated code executed faster than Postpass code. Of the 124 REGOA entries in the table, 32 of the entries represent cases in which the REGOA code executed faster than Postpass.

Benchmark	Number of Registers							
	8				16			
	long		medium		long		medium	
	RASSG	REGOA	RASSG	REGOA	RASSG	REGOA	RASSG	REGOA
Livermore								
loop1	1.56	1.02	1.48	1.01	1.23	1.04	1.17	1.04
loop2	1.17	1.06	1.15	1.04	1.30	0.68	1.07	0.57
loop3	1.14	1.04	1.13	1.03	1.02	1.02	1.04	1.03
loop4	1.05	0.88	1.12	0.88	1.06	1.04	1.03	0.98
loop6	1.09	0.96	0.84	0.97	1.27	0.92	1.05	0.92
loop7	1.04	0.95	0.93	0.95	2.26	1.01	1.77	0.98
loop8	0.35	0.99	0.40	0.99	0.51	1.00	0.61	1.00
loop9	1.35	0.98	1.23	0.97	2.37	0.98	1.96	0.97
loop10	1.43	1.05	1.14	1.03	2.18	0.81	1.38	0.80
loop11	1.23	0.95	1.16	0.95	1.00	0.98	1.04	0.97
loop12	1.17	0.87	1.13	0.90	1.02	1.00	1.00	1.00
loop13	0.52	0.99	0.62	0.99	1.48	0.96	1.04	0.96
loop14	0.90	1.00	0.89	1.00	1.37	1.00	1.30	1.00
Clinpack								
idamax	1.09	1.00	1.17	1.00	1.00	1.00	1.00	1.00
dscal	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
daxpy	1.17	1.00	1.15	1.00	1.00	1.00	1.00	1.00
matgen	0.92	1.00	0.85	1.00	1.13	1.00	1.06	1.00
dgefa	0.78	1.00	1.13	1.00	1.32	1.00	1.23	1.00
MatMul								
initmatrix	1.38	0.93	1.22	0.92	1.24	1.04	1.26	1.05
innerproduct	0.86	0.95	0.98	0.95	1.07	0.82	1.03	0.83
intmm	1.51	1.49	1.51	1.49	1.28	1.26	1.29	1.28
Heapsort								
hsort	0.90	1.00	1.16	1.00	1.06	1.00	1.07	1.00
Permute								
permute	1.09	1.00	1.08	1.00	1.09	1.00	1.08	1.00
initialize	2.80	2.36	2.49	2.38	3.34	2.66	2.21	2.00
perm	2.25	2.25	2.24	2.24	1.89	1.89	1.92	1.92
8 Queens								
try	0.94	1.00	0.98	1.00	1.01	1.00	1.01	1.00
doit	1.19	1.00	1.17	1.00	0.95	1.00	0.96	1.00
queens	1.32	1.81	1.37	1.77	1.32	1.81	1.37	1.77
Quick								
quicksort	1.05	1.00	1.06	1.00	1.03	1.00	1.00	1.00
initarr	1.13	1.00	1.11	1.00	1.03	1.00	1.01	1.00
quick	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 3: Speedup of fully cooperative and fully uncooperative techniques over postpass

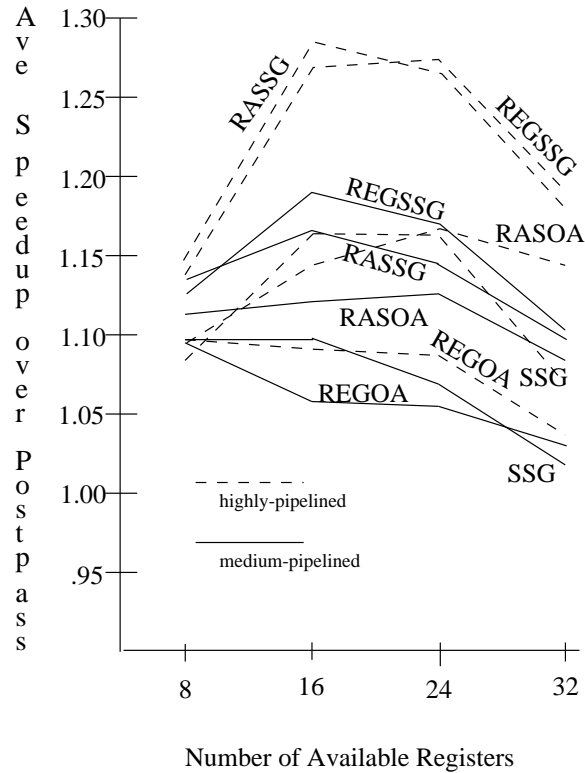


Figure 14: Average speedup of RASSG, SSG, RASOA, REGSSG and REGOA

REGOA performs better than Postpass in the cases that the region scheduling transformations have increased the amount of exploitable fine grain parallelism. REGOA can sometimes perform worse than Postpass for two reasons. First, the scheduling transformations can increase register pressure which can result in an increase in the amount of spill code inserted by the register allocator. And second, even in the presence of no spilling, the transformations change the size and shape of the interference graph which can cause the register allocator to add dependences which decrease the amount of exploitable fine grain parallelism.

RASSG performs better than Postpass for a couple of reasons. First, RASSG performs global instruction scheduling which increases the amount of exploitable fine grain parallelism. Also, the RASSG experiment performs scheduler-sensitive register allocation which adds only necessary dependences. In a couple of cases, Livermore loops 8 and 13, RASSG performed significantly worse than Postpass. This is explained in Section 5.2.3.

Figure 14 graphically depicts the average speedup over Postpass of each of the experiments. The dotted lines indicate the experiments in which the assumed architecture is a highly-pipelined machine. Each scenario, on average, performs better than Postpass. Except for the case of 32 registers in which SSG does not perform quite as well as REGOA, all of the techniques which include cooperation between either global scheduling and register allocation or local scheduling and register allocation or both, perform better than the totally uncooperative technique on the respective architecture.

The speedups are greatest on both architectures when either RASER or region scheduling is combined with SSG. This combination is particularly beneficial on the highly-pipelined machine. The RASSG scenario performs slightly better than REGSSG for 8 and 16 registers on the highly-pipelined machine and slightly worse for 24 and 32 registers. The RASSG scenario performs slightly better than REGSSG for 8 registers and worse than REGSSG for 16, 24, and 32 registers.

5.2.3 Problems with RASSG

The sometimes poor performance of RASSG can be attributed to 1) the inability of RASER to accurately measure register pressure and 2) problems with the live value reduction transformation. The register pressure of a region is estimated in *calc_max_live* by calculating the maximum number of live variables at any statement in the region. Since the register pressure is calculated on a higher level intermediate code and not on the lower level code upon which the allocation is actually performed, the calculation tends to be an overestimate of the number of registers actually needed. This results in REGSSG performing better than RASSG as can be seen in figure 14 with increasing number of registers. As the number of registers increases, reordering to increase fine grain parallelism becomes more important than reordering to reduce spilling since there are sufficient registers to perform an allocation without spilling. Region scheduling is more successful than RASER at reordering to increase fine-grain parallelism and thus, the performance of RASER degrades as the number of registers increases.

On the other hand, *calc_max_live* can sometimes underestimate the amount of registers needed to perform an allocation on a region without spilling. Figure 15 contains a section of code and lists the values live after

	live
x =	x
if (P)	
y = ...	x, y
= x	y
z = ...	z, y
= y	z
else	
z = ...	x, z
= x	z
endif	
= z	

Figure 15: Problem in Estimating Register Pressure

the execution of the corresponding statement. The maximum number of live values is 2; however, 3 registers are needed to perform an allocation without spilling. It is this type of underestimate which causes RASSG to perform worse than Postpass on loop 8. Our experiments with SSG indicated that some spilling occurs in loop 8 even in the 25 register case. Transformations made by RASER increase the register pressure even further. SSG does not perform well at higher register pressure levels and is therefore not able to attain an efficient allocation of the transformed loop 8 in the 8 and 16 register cases.

The live value reduction transformation attempts to reduce the amount of spill code inserted in the subsequent register allocation phase but can sometimes inadvertently cause an increase in the amount of spill code inserted. For example, returning to 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 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. This results in the poor performance of RASSG on loop 13 in the 8 register case.

6 Summary and Future Directions

In summary, the goal of this research was to investigate cooperation between register allocation and local and global instruction schedulers. We have presented a cooperative register allocator and a cooperative global instruction scheduler. An experimental comparison of five different scenarios of cooperation was made in order to determine the best speedup over uncooperative postpass scheduling. Our experiments indicate that the greatest speedups were obtained by performing either cooperative or uncooperative global instruction scheduling with cooperative register allocation and local instruction scheduling. This combination was particularly beneficial on a highly-pipelined machine. Our future work includes investigating the incorporation of scheduler-sensitivity into a hierarchical global register allocator [33] with the goal of reducing the compile-time expense of SSG, and investigation into integration with cooperative software pipelining strategies.

References

- [1] S. Abraham and K. Padmanabhan. Instruction reorganization for variable-length pipelined microprocessor. In *Proceedings of the International Conference on Computer Design*, New York, October 1988.
- [2] A. Aiken and A. Nicolau. A development environment for horizontal microcode. *IEEE Transactions on Software Engineering*, 14(5):584–594, May 1988.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] Steven. J. Beaty. Lookahead scheduling. In *Proceedings of the Twenty-fifth International Symposium on Microarchitecture*, pages 256–259, Portland, OR, 1992.
- [7] David Bernstein. An improved approximation algorithm for scheduling pipelined machines. In *International Conference on Parallel Processing*, St. Charles, Illinois, August 1988.
- [8] 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.
- [9] 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.
- [10] 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.

- [11] David G. Bradley, 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.
- [12] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992.
- [13] 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.
- [14] 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.
- [15] Frederick Chow and John Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4), October 1990.
- [16] Ron Cytron, Jeanne Ferrante, Barry Rosen, and Mark Wegman. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
- [17] K. Ebcioglu and A. Nicolau. A global resource-constrained parallelization technique. In *Proceedings of the ACM SIGARCH ICS-89: International Conference on Supercomputing*, Crete, Greece, 1989.
- [18] 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.
- [19] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.
- [20] 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.
- [21] 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.
- [22] James R. Goodman and Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. In *1988 International Conference on Supercomputing*, pages 442–452, Orlando, Florida, November 1988.
- [23] Rajiv Gupta and Mary Lou Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, April 1990.
- [24] Rajiv Gupta, Mary Lou Soffa, and Tim Steele. Register allocation via clique separators. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, Oregon, June 1989.
- [25] 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*, Paderborn, GERMANY, October 1992.
- [26] 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.
- [27] Wei Chung Hsu, Charles N. Fischer, and James R. Goodman. On the minimization of loads/stores in local register allocation. *IEEE Transactions on Software Engineering*, 15(10):1252–1260, 1989.
- [28] Jayashree Janardhan. Enhanced region scheduling for instruction level parallelism. Utah State University Master's Thesis, 1992.
- [29] P. Kolte and Mary Jean Harrold. Load/store range analysis for global register allocation. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, June 1993.
- [30] Monica Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988.
- [31] Cindy Norris. *Cooperative Register Allocation and Instruction Scheduling*. PhD thesis, University of Delaware, May 1995.
- [32] Cindy Norris and Lori L. Pollock. A scheduler-sensitive global register allocator. In *Supercomputing '93 Proceedings*, Portland, OR, November 1993.

- [33] 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.
- [34] Cindy Norris and Lori L. Pollock. Register allocation sensitive region scheduling. In *PACT '95: International Conference on Parallel Architectures and Compilation Techniques*, Limassol, Cyprus, June 1995.
- [35] S. Novack and A. Nicolau. Mutation scheduling: A unified approach to compiling for fine-grain parallelism. In *Languages and Compilers for Parallel Computing*, pages 16–30. Springer-Verlag, 1994.
- [36] 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.
- [37] 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.
- [38] 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.
- [39] Peter Steenkiste and John Hennessy. A simple interprocedural register allocation algorithm and its effectiveness for LISP. *ACM Transactions on Programming Languages and Systems*, January 1989.
- [40] Philip H. Sweany and Steven J. Beaty. Dominator-path scheduling – a global scheduling method. In *Proceedings of the Twenty-fifth International Symposium on Microarchitecture*, pages 260–263, Portland, OR, 1992.
- [41] David W. Wall. Register allocation at link time. *SIGPLAN Notices*, 21(7):264–275, July 1986.
- [42] 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.
- [43] 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.
- [44] M. J. Wolfe. *Research Monographs in Parallel and Distributed Computing*. The MIT Press, 1989.