Integrating Program Optimizations and Transformations with the Scheduling of Instruction Level Parallelism*

David A. Berson[1] Pohua Chang[1] Rajiv Gupta[2] Mary Lou Soffa[2]

[1] Intel Corporation, Santa Clara, CA 95052
[2] University of Pittsburgh, Pittsburgh, PA 15260

Abstract. Code optimizations and restructuring transformations are typically applied before scheduling to improve the quality of generated code. However, in some cases, the optimizations and transformations do not lead to a better schedule or may even adversely affect the schedule. In particular, optimizations for redundancy elimination and restructuring transformations for increasing parallelism are often accompanied with an increase in register pressure. Therefore their application in situations where register pressure is already too high may result in the generation of additional spill code. In this paper we present an integrated approach to scheduling that enables the selective application of optimizations and restructuring transformations by the scheduler when it determines their application to be beneficial. The integration is necessary because information that is used to determine the effects of optimizations and transformations on the schedule is only available during instruction scheduling. Our integrated scheduling approach is applicable to various types of global scheduling techniques; in this paper we present an integrated algorithm for scheduling superblocks.

1 Introduction

Compilers for multiple-issue architectures, such as superscalar and very long instruction word (VLIW) architectures, are typically divided into phases, with code optimizations, scheduling and register allocation being the latter phases. The importance of integrating these latter phases is growing with the recognition that the quality of code produced for parallel systems can be greatly improved through the sharing of information. Integration of register allocation and instruction scheduling has been recently studied by various researchers [1, 14, 13]. However, the integration of optimizations and transformations with schedulers has not been fully addressed.

In the optimization phase, optimizations are applied to improve the quality of code while restructuring transformations are applied to increase the instruction level parallelism (ILP). Since optimizations and transformations may increase the requirements for registers and functional units beyond the numbers supported by the architecture, their application may adversely affect the quality of code generated.

* Partially supported by National Science Foundation PYI Award CCR-9157371 and a grant from Intel Corporation to the University of Pittsburgh.

In the scheduling phase, both local and global techniques are used to exploit ILP. Global scheduling techniques uncover and exploit ILP across basic blocks [5, 8, 6, 1]. The program is divided into scheduling units (such as traces, superblocks, and control dependence regions) composed of a collection of basic blocks. The scheduling units are processed one at a time by a scheduler and are represented using directed acyclic graphs (*DAG*) to assist in the scheduling process. ILP is uncovered by restructuring the code within a scheduling unit. If enough parallelism is not found within the unit, code is propagated from adjacent scheduling units. Code is moved across units using code duplication, speculative execution [9], and predicated execution [7].

The separation of the optimization and scheduling phases is problematic in that the applications of optimizations and transformations are performed without knowing if the scheduler will be able to utilize their effects. For example, DAG restructuring transformations can be applied to increase parallelism within a superblock as well as reduce critical path length within a superblock. However, the optimizer cannot apply these transformations unless it is aware of the superblocks that will be constructed during instruction scheduling. Thus, the usefulness of these transformations cannot be known until scheduling actually begins. Also, when a choice of restructuring opportunities is available, the scheduler has the best information to determine which choice would produce the best schedule. Code that is moved by the scheduler can create opportunities for further optimizations. For example, hoisting of partially redundant code can result in fully redundant code. Such opportunities can only be exploited by integrating the redundancy elimination optimization with instruction scheduling.

In this paper, we present an approach to scheduling with the goal of integrating selected code transformations and optimizations into the scheduler. We present simple guidelines that enable the identification of optimizations that should be integrated with instruction scheduling and those that may be performed prior to instruction scheduling. The decision to apply an optimization or transformation duing scheduling is based upon information that is only available during instruction scheduling. Such information includes the effect of optimizations and transformations upon (1) the availability of registers, (2) the availability of functional units, and (3) the schedulability of instructions in an idle slot of a schedule in progress. The integrated approach utilizes measures of the resources required, including both functional units and registers, that are maintained and used by the scheduling algorithm. By reorganizing code within a block in a manner that keeps register pressure in check, we attempt to minimize the generation of spill code. Furthermore, resource utilization levels also guide the selection of code statements for code motion between blocks.

The remainder of the paper is organized as follows. In section 2 we identify the optimizations that should be integrated with instruction scheduling. In section 3 we briefly outline the relationship between resource requirements and global code motion. In section 4 we present an integrated schedule driven algorithm using superblocks [8].

2 Optimizations and Instruction Scheduling

While the integration of optimizations with instruction scheduling may improve their effectiveness, it also increases the complexity of the instruction scheduling algorithm. Therefore it is important to examine the interactions between various optimizations and the instruction scheduler to identify the optimizations that should be integrated. The optimizations that do not significantly interact with the instruction scheduling may be applied in a separate optimization phase that precedes scheduling.

Three main characteristics of optimizations that determine whether or not an optimization should be integrated with the instruction scheduler are:

Opportunity: If all of the opportunities for applying an optimization can be determined prior to scheduling, then the optimizations can be carried out independently of the scheduler. On the other hand, if scheduling actions create additional opportunities for optimization, these opportunities can only be exploited by applying optimizations during scheduling.

Usefulness: If the optimization is always considered to be useful, it need not be integrated. If the usefulness of an optimization requires information that is typically available during instruction scheduling, such as the availability of registers, then the optimization should be integrated with scheduling.

Reversability: An optimization may be generally useful, but harmful in some situations. Furthermore, the latter situations may only become apparent during instruction scheduling. This type of optimization may be applied prior to scheduling and selectively reversed during instruction scheduling if it is determined that the application of the optimization was indeed harmful and, in fact, the optimization can be reversed.

According to the above criteria, there are a number of traditional optimizations that can be applied prior to instruction scheduling. The optimizations of constant propagation, loop invariant code motion, induction variable elimination, strength reduction, and dead code elimination can be applied prior to scheduling. The opportunities for these optimizations are not created during scheduling, they are generally useful, and their reversal is not required. Copy propagation can also be applied prior to scheduling. However, in some situations creation of copies may facilitate code reordering. In these situations copy creation can be carried out during scheduling. Partial dead code is another optimization which is useful in most situations and therefore can be performed prior to scheduling. In some situations it may be beneficial to reverse this optimization. However, this can be easily achieved by the scheduler through global code motion.

The above criteria also suggest that some optimizations should be integrated with instruction scheduling. These include the redundancy elimination optimization and DAG restructuring transformations. Opportunities of redundancy elimination can be created by code motion, its usefulness cannot be fully estimated prior to scheduling due to its influence on register pressure, and it cannot always be reversed during scheduling. The usefulness of DAG restructuring cannot always be determined prior to scheduling. In the remainder of this section we discuss the integration of above optimizations with instruction scheduling.

2.1 Redundancy Elimination

The scope over which the redundancy elimination optimization is applied is important in determining its integration with instruction scheduling. The application of this optimization to individual basic blocks can be applied prior to instruction scheduling because the effect of local optimization on register pressure is expected to be minimal. In rare situations where the effect is significant, local optimization can be easily reversed during instruction scheduling using rematerialization. On the other hand, global redundancy elimination can significantly increase register pressure. Furthermore, in general, the reversal of this optimization during scheduling is not possible due to code that is already scheduled. Finally opportunities for redundancy elimination may be created due to code motion performed during instruction scheduling. Therefore global redundancy elimination should be integrated with instruction scheduling.

Let us consider instruction scheduling based upon superblocks. Elimination of global redundancy within a superblock can be performed immediately preceding its scheduling. This redundancy may arise due to multiple evaluations of an expression within the superblock or due to the availability of an expression at the entry of the superblock. The former opportunities can be exploited in much the same way as local redundancy elimination. The exploitation of latter opportunities increases the register pressure for other parts of the program and also may require the introduction of copy statements to save the value of the expression in a specific location. However, if the superblocks processed earlier are more important than those processed later, then the application of redundancy elimination improves the quality of code for the current superblock at the expense of the quality of code for superblocks processed later.

The application of redundancy elimination to a superblock requires the computation of available expressions at the entry point of the superblock. Since new available expressions may be generated during code motion performed by the instruction scheduler, the expressions available at the entry to the superblock currently being scheduled must be computed immediately preceding the optimization of the superblock. Only the expressions computed inside the superblock prior to the redefinition of variables used in the expressions can be potentially optimized. Given such an expression, the demand driven algorithm presented in Figure 1 determines whether or not the expression is available at the superblock's entry.

Starting from the entry of the superblock, the algorithm searches backwards through the flow graph of a predecessor superblock for the evaluations of the expression. If such evaluations are found, a temporary variable T is created into which the value of the expression is copied so that it is available at the entry of the superblock in T. It should be noted that during the search for an expression, parts of the program that have already been scheduled may also be encountered. Since scheduled parts correspond to more important superblocks, we should not introduce a copy instruction within the schedule since it would increase the schedule's length. Instead we ensure that an expression evaluated in a scheduled superblock is only considered available at one of its exits if the

value of the expression is available in a register at the exit.

Our algorithm maintains a worklist that contains points in the program at which availability of the expression is required to ensure that the expression is available at the superblock's entry. Each member of the worklist is a query of the form $(n, regs)$ indicating that we need to determine whether the expression is available at n's exit. The set $regs$ is used during traversal through scheduled portions of the code to track the identities of registers whose current values are available at a superblock exit. Thus, if the expression being searched for is computed into one of the registers in $regs$, then it will be available at the exit. On the other hand, if the scheduled code computes the expression into registers that are overwritten prior to exiting the superblock, then the expression is not available. If the algorithm finds available evaluations of the expressions along all paths, the worklist becomes empty and the search terminates. The introduction of copies into T makes this value available at the superblock entry.

Available$(exp, \mathcal{SB}_{entry})$
 $Worklist = \{(n, \mathcal{REGS})_{exp} : n \in Pred(\mathcal{SB}_{entry})$ *and n is scheduled*$\}$
  $\cup\{(n, \phi)_{exp} : n \in Pred(\mathcal{SB}_{entry})$ *and n is unscheduled*$\}$;
 success = true;
 while $Worklist \neq$ *empty* and *success* **do**
  get a query $(m, regs)_{exp}$ from Worklist;
  if m computes exp **then**
   if (m is scheduled and it writes to a register $R \in regs$) or (m is unscheduled)
   then Add m to expression evaluation set *Eval* **endif**
  elseif (m defines variables used in exp) or (m is the dummy start node of cfg)
  then success = false
  else – propagate query
   $Worklist = Worklist$
    $\cup\{(p, \phi)_{exp} : p \in Pred(m)$ *and p is unscheduled*$\}$
    $\cup\{(p, regs - \{R\})_{exp} : p \in Pred(m)$, *p and m are scheduled and m writes to* $R\}$
    $\cup\{(p, \mathcal{REGS})_{exp} : p \in Pred(m)$, *p is scheduled and m is unscheduled*$\}$
  endif
 endwhile
 if *success* **then**
  create a new temporary T;
  for each *instruction/statement* $\in$ *Eval* **do**
   move value of expression from register/variable to T
  endfor
  return(T)
 else return() **endif**

Fig. 1. Demand Driven Determination of the Availability of expression exp at superblock $\mathcal{SB}_{entry}$.

From the above discussion it is clear that our algorithm exploits only those opportunities that benefit superblocks scheduled earlier at expense of superblocks scheduled later. That is, the register pressure is only increased for superblocks that are yet to be scheduled. Also, our approach allows the exploitation of new opportunities for redundancy elimination that arise due to code motion performed by the instruction scheduler.

In our discussion so far we have not considered the elimination of partial redundancy elimination [10]. This is because the application of tail duplication that is performed prior to superblock construction, converts partial redundancy to full redundancy. On the other hand, if trace scheduling is being used, then prior to scheduling a trace, we must also perform partial redundancy elimination using a similar integrated technique.

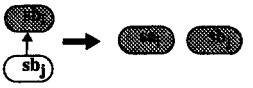
2.2 DAG Restructuring Transformations

As in the case of redundancy elimination, the scope over which DAG restructuring transformations are applied is relevant to their integration with instruction scheduling. The transformations of individual basic blocks can be carried out prior to instruction scheduling since their application is typically beneficial. However, the transformations over a larger scope, such as within a superblock and across superblocks, cannot be applied prior to instruction scheduling. This is due to two main reasons. First, the superblocks are identified during instruction scheduling. Only after one superblock is identified and scheduled is another superblock constructed. This is because the scheduling of a superblock is followed by introduction of compensation code which must be considered prior to constructing future superblocks. Second, the application of DAG restructuring transformations are typically accompanied with an increase in register and functional unit requirements. Therefore the application of these transformations over large segments of code, such as those that form a superblock, may significantly increase register pressure and hence the potential for spill code generation. Thus, this class of transformations, when performed over a scope larger than a basic block, is best left for the instruction scheduler to perform.

During instruction scheduling, DAG restructuring transformations are used in two types of situations. First, if a superblock contains insufficient parallelism, then restructuring of a DAG can be performed to increase the amount of parallelism within the block. Note that the transformation of a superblock may require global code motion. Second, if restructuring of the DAG is not enough to create sufficient parallelism in a superblock, then the DAGs of adjacent unscheduled basic blocks can be restructured to facilitate the propagation of operations from an adjacent basic block to the superblock with insufficient parallelism.

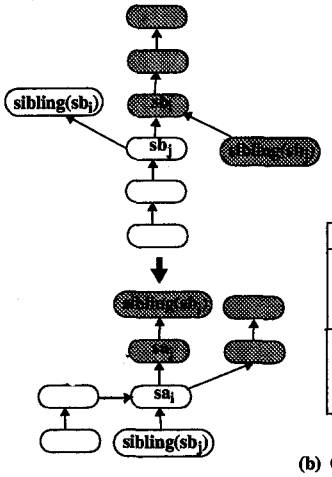
There are two types of transformations that we consider for DAG restructuring. The first type of transformation, shown in Figure 2, exploits the presence of *constant operands* to reduce the height of a DAG and increase ILP. The second type of transformation, shown in Figure 3, exploits *algebraic properties* of the operators to restructure the DAG. The situations under which the transformations are applicable and the conditions under which they are useful during the restructuring of a DAG are also given in the figures. In the discussion of these transformations we assume that the DAG corresponds to a superblock. Thus, it should be noted that the application of DAG restructuring transformations involves global code motion and thus may require the introduction of compensation code. To emphasize this fact, we assume that the nodes in the figures come from two different basic blocks which are indicated by shaded and

unshaded nodes. A change in the shading of a node indicates that the node has been moved using global code motion.



Before Transformation	After Transformation
sb_j: y = z [+|-] $const_2$ sb_i: x = y [+|-] $const_1$	sb_j: y = z [+|-] $const_2$ sa_i: x = z [+|-] $const_2$[+|-] $const_1$
sb_j: y = z * $const_2$ sb_i: x = y * $const_1$	sb_j: y = z * $const_2$ sa_i: x = z * $const_2$ * $const_1$

(a) Forward Substitution.



Conditions for height reduction and increasing parallelism:
Upheight(sb_i) > Upheight(sibling (sb_i)) + 1
Downheight(sb_j) > Downheight(sibling (sb_j)) + 1

Before Transformation	After Transformation
sb_j: y = z [+|-] $const_2$ sb_i: x = y [+|-] $const_1$	sa_i: x = z [+|-] $const_2$ [+|-] $const_1$ sa_j: y = x [-|+] $const_1$
sb_j: y = z * $const_2$ sb_i: x = y * $const_1$	sa_i: x = z * $const_2$ * $const_1$ sa_j: y = x div $const_1$

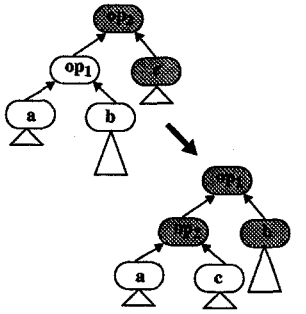
(b) Operator Replacement.

Fig. 2. Exploiting Constant Operands.

Forward substitution in the presence of constant operands eliminates a data dependency between operations (see Figure 2(a)). As can be seen in the Figure 2(a), whenever a dependency exists between two nodes that have constant operands and have no other dependencies, height reduction and increased parallelism always occurs when we eliminate the dependency through forward substitution. Therefore, there are no conditions needed for applying this transformation. Before the application of the transformations the two nodes are in different basic blocks. However, after the transformation, they belong to the same basic block.

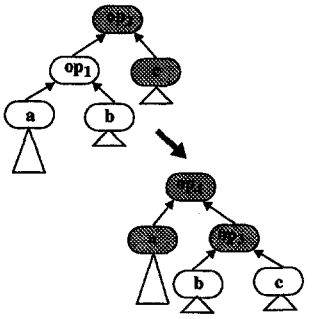
We can sometimes apply a transformation to reduce the height and increase parallelism even when there are other dependencies involved in expressions with constant operands. *Operator replacement* is the transformation used to reorder a pair of data dependent operations, enabling height reduction and parallelism under certain conditions. In Figure 2(b) a node $sibling(sb_i)$ is dependent on sb_j, and sb_i depends on $sibling(sb_j)$ (e.g., output dependency). In order to create more parallelism, we ideally would like to have the bottom node and sibling of sb_j, which have no dependencies, execute in parallel. Since one of sb_i's dependencies would have then been computed, by using the operator replacement transformation, we can compute sb_i before sb_j. We then compute sb_j and its de-

pendent node. This transformation enables the bottom part of the DAG and the top part of the DAG to execute in parallel with the computation of the dependent section. However, in order for this transformation to be beneficial, we must ensure that the transformed path that computes sb_i and sb_j does not increase such that it takes longer to execute than the bottom part of the DAG or the top part of the DAG. Thus conditions are given to ensure that the height of the DAG from sb_i to the top ($Upheight(sb_i)$) is greater by more than one than the height of the sibling of sb_i to the top of the DAG. It is this portion of the graph that would execute in parallel with the top part of the DAG. A similar check is needed for the bottom of the DAG, using the function $Downheight(sb_j)$ since the sb_j and the sibling sb_j would execute in parallel with the bottom part of the DAG. The shading of nodes indicates that the application of this transformation requires global code motion.



Condition for height reduction:
Downheight(b) > Downheight(c) + 1

Before Transformation (a op_1 b) op_2 c	After Transformation (a op_2 c) op_1 b
(a [+|-] b) [+|-] c	(a [+|-] c) [+|-] b
(a * b) * c	(a * c) * b
(a div b) div c	(a div c) div b



Condition for height reduction:
Downheight(a) > Downheight(c) + 1

Before Transformation (a op_1 b) op_2 c	After Transformation a op_4 (b op_3 c)
(a + b) + c	a + (b + c)
(a - b) - c	a - (b + c)
(a - b) + c	a - (b - c)
(a + b) - c	a + (b - c)
(a * b) * c	a * (b * c)
(a div b) div c	a div (b * c)

Fig. 3. Exploiting Algebraic Properties of Operators.

In expressions that do not necessarily involve constant operands, we may be able to exploit the algebraic properties of *associativity* and *commutativity* of operators to perform restructing of a DAG to reduce its height, enabling more parallelism. The general idea is to reduce the height of a DAG by shortening a path in the DAG that is longer than other paths. By using appropriate conditions on the length of paths involved, transformations will reduce the height of a DAG by reducing the length of a longer path in the DAG and lengthening a shorter path in the DAG. Consider the DAGs and two transformations shown in Figure 3. The first transformation handles the case where the length of the path from the bottom of the DAG to b is longer that the length of the path from

the bottom to c. The transformation is to swap the subDAGs of b and c, using commutativity and associativity properties of operators. This transformation is applied when the height of the subDAG rooted at b is at least one greater than the height of the subDAG rooted at c. The transformation shortens the overall height of the DAG by one. As in the case of earlier DAG transformations, this transformation may also require global code motion.

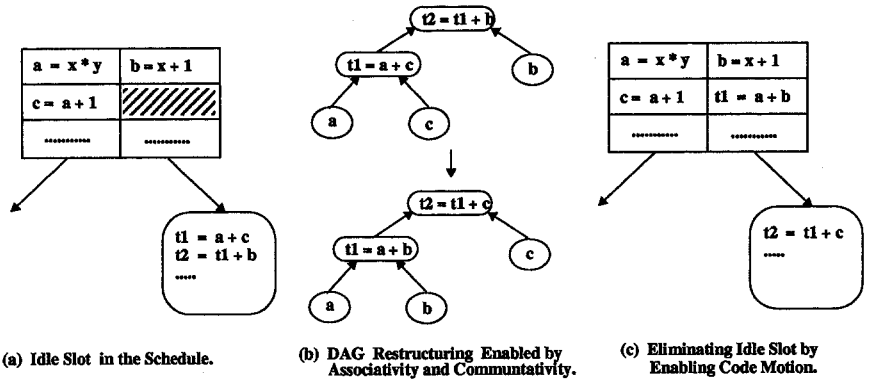


Fig. 4. Enabling Global Code Motion by DAG Restructuring.

Consider the other case where the longest path is rooted at a. The second transformation shown in the figure handles this case. If the height of the subDAG rooted at a is longer (by more than one) than that of the subDAG rooted at c, we can decrease the overall height of the DAG by interchanging the subDAG at a with the subDAG at c. This transformation is accomplished using associativity of the operators, as shown in the table. As was true in the previous case, the height of the DAG is reduced by one. Thus, we can repeatedly apply these transformations, reducing the height of the DAG by one as long as the conditions continue to exist.

Let us now consider the situation in which the scheduler is unable to find sufficient parallelism to completely fill a long instruction with operations even after DAG restructuring transformations have been applied to the superblock. Thus, we must examine unscheduled adjacent blocks for operations that can be moved to the current block being scheduled. Even if we are successful in finding operations that can be propagated, the propagation of operations may not always be useful. It is only beneficial to propagate operations that are immediately schedulable. However, if such operations are not immediately apparent in an unscheduled adjacent block, then we may be able to expose such operations by restructuring the DAG of the adjacent block. If the unscheduled adjacent block is a successor (predecessor) of the block with insufficient parallelism, then the leaves (roots) of the adjacent block must be considered for transformation.

The scheduled block in Figure 4(a) contains an idle slot and Figure 4(b) represents the portion of the DAG for an unscheduled adjacent block. We would like to propagate operations from that DAG to the scheduled block to eliminate the idle slot. The values of a, b and c are computed in the scheduled block and used by the adjacent block. Since the operation that computes the value of c

has not been scheduled prior to the instruction with the idle slot, the statement $t1 = a + c$ cannot be propagated and scheduled in the position of the idle slot. On the other hand since the values of a and b have already been computed, we can restructure the DAG as shown in Figure 4(b) and propagate the operation $t1 = a + b$ to the position of the idle slot. Thus, depending upon the availability of operand values we may choose between the equivalent DAGs shown in Figure 4(b). As we can see from this situation, the appropriate DAG can only be selected by the scheduler.

Another example in Figure 5(a) shows a schedule with two idle slots. These idle slots may be created due to a long latency of the multiplication operator. As shown in Figure 5(b) one of the idle slots can be eliminated by propagating a statement from the adjacent block. However, if the DAG for the adjacent block is restructured using forward substitution, then both idle slots can be eliminated (see Figure 5(c)).

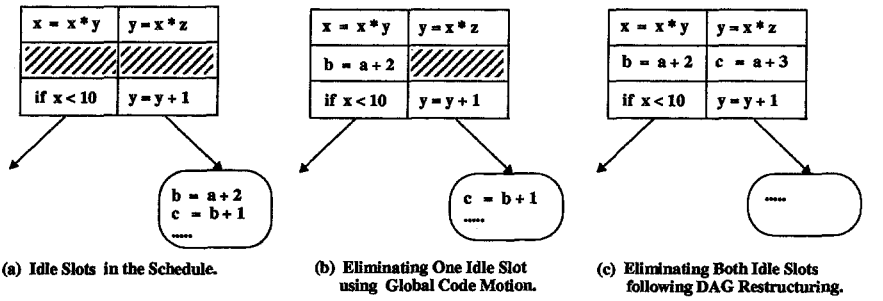


Fig. 5. Increasing Global Code Motion by Exposing Parallelism.

3 Resource Requirements and Global Code Motion

The scheduling of operations is driven by the requirement levels of resources, which includes both functional units and registers. We have developed a resource conscious scheduling technique that integrates register allocation and scheduling. The technique consists of identifying sections of a program that underutilize registers and functional units and sections that require more of these resources than are available. The scheduler is guided by the resource requirements at each point in the block being scheduled and the critical paths through the block. The scheduler tries to move code from the overutilized sections to underutilized sections. The technique first measures the requirements of resources at each program point. This information is then used by the scheduler to determine where code motion should occur, taking into account the resource requirements of both resources simultaneously [1].

To measure the resource requirements, we first create a special type of DAG for each resource that reflects the resource requirements for the code in a block. For each resource, the precedence and usage information in a block is used to construct a partial ordering of the operations indicating which pairs of operations temporally share a single instance of the resource under any allowable schedule. Details of computing usage information for registers is described elsewhere [2]. For each resource R, a $Reuse_R$ DAG is constructed to represent the partial

order. Sets of operations in the $Reuse_R$ DAG that are fully ordered are called *allocation chains*, since by definition of the partial order, they can all safely be allocated a single instance of R. The maximum number of resources required to exploit all exposed parallelism is given by the minimum number of allocation chains that cover the $Reuse_R$ DAG.

The technique requires identification of the following sets:

Excessive sets: A block may have areas where the ILP exceeds the resources provided by the architecture. We call sets of operations that can execute in parallel and would require too many resources *excessive sets*. While the number of allocation chains covering a schedulable block indicates if there are any excessive sets, it is desirable to know exactly which operations are in excessive sets. Excessive sets are computed by finding the sections of the schedulable block where there are portions of an excessive number of allocation chains, which is performed in graph linear time [2].

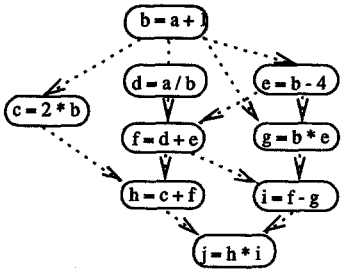
Critical sets: A critical set of length L is the minimal set of operations that must be propagated out of a schedulable block to reduce that block's critical path length by L cycles.

The goal of global scheduling is to move operations from a *source* block to a *destination* block to decrease the program's execution time. A decrease in execution time is achieved when the critical path length is reduced in the source block while the critical path length of the destination block is not increased. Thus it is desirable that operations should be moved in groups, where each group is a critical set. In addition, the operations should not be moved into or create excessive sets.

As an example, consider the block of code shown in Figure 6(a), which uses only integer functional units and registers. Since a single type of functional unit is used, the $Reuse_{FU}$ DAG is the same as the program DAG. The partial schedule for functional units is shown in Figure 6(b). Each column represents one allocation chain, so three functional units are required to exploit all exposed ILP. Similarly, a $Reuse_{Reg}$ DAG and partial schedule can be constructed for registers by considering their usage characteristics.

Assume that the architecture has two functional units available. Then the sets {C, D, E}, {C, D, G}, and {C, F, G} are all functional unit excessive sets. For scheduling, only a summary set of all nodes that are in at least one excessive set for a resource is needed, and can be computed in graph linear time. The summary excessive set for functional units is {C, D, E, F, G}. Now assume that global code motion is being performed on the DAG in Figure 6(a) to move operations from the top of the block to some other location. The first critical set is {B}, as that will reduce the height of the block by one operation. Now operations C, D, and E can be moved. However, moving C at this time will not affect the block's height, so the next critical set is {D, E}.

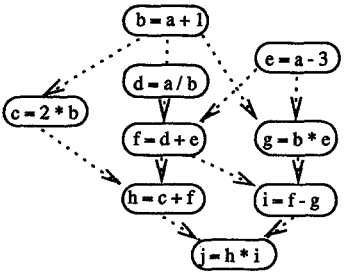
Consider the construction of a final schedule for the DAG in Figure 6(a). C is in an excessive set but there are no idle slots in the other two allocation chains. That is, the scheduling of C would require an increase in the execution time of the block. However E is a candidate for the forward substitution transformation



(a) *Reuse*$_{FU}$ DAG.

	b=a + 1	
c=2 * b	d=a / b	e=b - 4
	f=d + e	g=b * e
	h=c + f	i=f - g
	j=h * i	

(b) Functional Unit Partial Schedule.



(c) Post *Reuse*$_{FU}$ DAG.

b=a + 1	e=a - 3
d=a / b	c=2 * b
f=d + e	g=b * e
h=c + f	i=f - g
j=h * i	

(d) Functional Unit Final Schedule.

Fig. 6. *Reuse* DAGs and Schedules.

discussed in Section 2.2. After performing the transformation, B and E can be scheduled in parallel, as shown in Figure 6(c). Application of the transformation allows C to be scheduled in the slot previously used by E, eliminating all excessive sets. The transformation also enables E to be moved out of the region by global code motion, decreasing the critical path length of the region by one instruction.

This scheduling technique is integrated with optimizations and transformations (including using a common DAG) to produce our final scheduler. The algorithm for this scheduler is given in the next section.

4 An Integrated Superblock Scheduling Algorithm

In this section we present a scheduling algorithm for superblocks that integrates the optimizations and restructuring transformations described in the previous sections with the resource conscious scheduling algorithm of the preceding section. A superblock is a loop free sequence of basic blocks which has a single entry point and possibly multiple exit points. Thus, a superblock can pass through several splits in the control flow graph but it cannot include a join in the flow graph. In order to create large superblocks our algorithm first carries out *tail duplication* which duplicates code following a join in the flow graph. Superblocks are constructed one at a time using profiling information and scheduled using the integrated scheduling and optimization algorithm.

Using the available expression information, a superblock is examined for redundancy elimination prior to scheduling. The DAG for the superblock is constructed, and using the analysis summarized in the previous section, the register

and functional unit requirements, the excessive sets, and critical sets in the superblock are identified. We then perform *DAG restructuring* transformations to increase parallelism, if so required, and reduce the critical path length.

During the scheduling of a given superblock, the set of operations ready for scheduling are repeatedly identified and packed into long instructions of a VLIW machine. At any given point during scheduling $AvailFUs$ and $AvailRegs$ denote the number of functional units and registers that are not in use and are therefore available for use by the operations that will be packed into the next long instruction. Let *Ready* denote the set of operations that are ready for scheduling. There are two possibilities at this point. Either the *Ready* set is large enough and therefore provides enough parallelism to keep all functional units busy during the next instruction or there is insufficient parallelism at this point. In the former case a subset of operations from the *Ready* set is selected for scheduling while in the latter case we must find additional parallelism by moving operations from as yet unscheduled superblocks that are adjacent to the current superblock. Thus, our algorithm not only performs code reordering within a superblock but it also performs code motion between superblocks.

Let us consider the situation in which the *Ready* set provides sufficient or excessive parallelism. The selection of operations for scheduling is based upon several criteria. First, we should give preference to operations which belong to a critical path in the superblock. Second, we should also keep register pressure in check so that scheduling of operations can continue unhindered through the rest of the superblock. By keeping a balance between operations requiring new registers and operations freeing up registers we attempt to keep register pressure below a preset threshold. An operation frees a register if it represents the last use of the register's value. If enough operations that free registers are not available we may also consider generating spill code. If a register contains a value that is not used for the remainder of the superblock, we can consider spilling the value into memory. Therefore, the operations scheduled in the current long instruction may contain operations from the *Ready* set and some additional spill code (*Spill*).

If the *Ready* set provides insufficient parallelism for filling the next long instruction we proceed as follows. We examine basic blocks that are as yet unscheduled and are adjacent to the superblock being currently scheduled. Operations from these adjacent blocks are moved into the current block to increase the number of operations available for scheduling. Our goal during the propagation of operations is not only to create sufficient parallelism in the current superblock but also to increase the potential of obtaining shorter schedules for the adjacent block being considered. If the adjacent block contains excessive functional unit sets, then we should propagate operations from these sets. On the other hand, if the adjacent block contains insufficient parallelism, then we must attempt to propagate operations from critical sets in order to reduce the critical path length of the adjacent block. Finally, if the operations that can be found for propagation cannot be immediately scheduled in the current block, then we apply DAG restructuring transformations to expose operations for propagation that can be immediately scheduled in the current block. Thus, in this situation operations

IntegratedScheduling() {
Perform *tail duplication* to create larger *superblocks.*
Apply all of the *scheduling-independent optimizations.*
Compute *available expressions* information.
while unscheduled code remains **do**
Construct the next superblock, $\mathcal{SB}$, using profile information.
Compute *available expression* information for $\mathcal{SB}$.
Eliminate *fully redundant* expressions within $\mathcal{SB}$ including
those that may be created by duplication of *partially redundant* expressions.
Compute resource usage, excessive sets, and critical sets for $\mathcal{SB}$.
Initialize $AvailFUs$ and $AvailRegs$ by examining usage of resources
at the end of each already scheduled superblock whose execution
immediately precedes the execution of $\mathcal{SB}$.
Apply *DAG restructuring transformations* to $\mathcal{SB}$ for
increasing parallelism, if required, and reducing critical path length.
while operations in $\mathcal{SB}$ remain unscheduled **do**
$Ready \leftarrow$ { I: operation I is ready to be scheduled }
if $|Ready \geq AvailFUs|$ **then**
Identify operations for scheduling $Select \subseteq Ready$ using following criteria:
- *Reduce critical path length*: give preference to operations that are a part of a critical path in the superblock.
- *Control register pressure and spilling*: balance selection of operations requiring additional registers with operations that free registers and *spill* registers if register pressure is too high.

Let $Sched = Select \cup Spill$ be the set of operations scheduled.
else $-|Ready < AvailFUs|$
Propagate set of operations *Prop* from unscheduled basic blocks
adjacent to $\mathcal{SB}$ using the following criteria:
- *Redistribute excessive ILP*: Move operations from an adjacent block that belong to functional unit *excessive sets.*
- *Reduce critical path length*: Move operations that are in a *critical set* of an adjacent block to reduce critical path length for the adjacent block.
- *Transform and propagate*: Apply *DAG restructuring transformations* to an adjacent block to enable propagation of operations.

Let $Sched = Ready \cup Prop$ be the set of operations scheduled.
endif
Mark operations in *Sched* as scheduled.
Update $AvailFUs$ and $AvailRegs$.
endwhile
endwhile
}

Fig. 7. An Integrated Scheduling Algorithm.

from the *Ready* set and also operations that are propagated from neighboring blocks (*Prop*) are scheduled in the current long instruction. The steps of the algorithm are summarized in Figure 7.

We presented an algorithm using the integrated approach for scheduling superblocks. However, this technique is also applicable to other global scheduling techniques such as those based upon control dependence regions.

The research most closely related to our work is the mutation scheduling technique proposed by Novack and Nicolau [12] which integrates a number of DAG transformations into the instruction scheduler. In most architectures certain functions can be performed through a number of alternative instruction sequences. Mutation scheduling selects the appropriate alternative based upon availability of resources. DAG transformations which exploit constant operands are also used. In other related work Ebcioglu et al. and Chang et al. consider the removal of partially dead code during instruction scheduling [4, 3].

References

1. D. Berson, R. Gupta and M.L. Soffa, "Resource Spackling: A framework for integrating register allocation in local and global schedulers," In *Proc. of Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 135-146, 1994.
2. D. Berson, R. Gupta and M.L. Soffa, "GURRR: A global unified resource requirements representation," In *ACM SIGPLAN Workshop on Intermediate Representations, Sigplan Notices*, vol. 30, pages 23-34, April 1995.
3. P.P. Chang, S.A. Mahlke, and W-M. Hwu, "Using profile information to assist classic code optimizations," *Software-Practice and Experience*, 21(12):1301–1321, Dec. 1991.
4. K. Ebcioglu, R.D. Groves, K-C. Kim, G. Silberman, and I. Ziv, "VLIW compilation techniques in a superscalar environment," In *Proc. of Sigplan Conf. on Prog. Language Design and Implementation*, pages 36-48, 1994.
5. J.A. Fisher, "Trace scheduling: a technique for global microcode compaction," *IEEE Trans. on Computers*, C-30(7):478-490, 1981.
6. R. Gupta and M.L. Soffa, "Region scheduling: an approach for detecting and redistributing parallelism," *IEEE Trans. on Software Engineering*, 16(4):421-431, 1990.
7. P. Hsu and E. Davidson, "Highly concurrent scalar processing," In *Proc. of 13th Annual International Symposium on Computer Architecture*, pages 386-395, 1986.
8. W-M. Hwu et al., "The superblock: an effective technique for VLIW and superscalar compilation," In *The Journal of Supercomputing* vol. A, pages 229-248, 1993.
9. W-M. Hwu and Y. Patt, "Checkpoint repair for out-of-order execution machines," In *Proc. of 14th Annual Intl. Symp. on Comp. Architecture*, pages 18-26, 1987.
10. J. Knoop, O. Ruthing and B. Steffen, "Optimal code motion: theory and practice," In *ACM Trans. on Programming Languages and Systems*, 16(4):1117-1155, 1994.
11. J. Knoop, O. Ruthing, B. Steffen, "Partial dead code elimination," In *Proc. of Sigplan Conf. on Prog. Language Design and Implementation*, pages 147-158, 1994.
12. S. Novack and A. Nicolau, "Mutation scheduling: a unified approach to compiling for fine-grain parallelism," *Proc. Languages and Compilers for Parallel Computing*, LNCS 892, 1994.
13. C. Norris and L. Pollock, "A scheduler-sensitive global register allocator," In *Proc. of Supercomputing*, pages 804-813, 1993.
14. S. Pinter, "Register allocation with instruction scheduling: a new approach," In *Proc. of Sigplan Conf. on Prog. Lang. Design and Impl.*, pages 248-257, 1993.