Compiler Optimization for Superscalar Systems: Global Instruction Scheduling without Copies

Philip H. Sweany
Steven M. Carr
Brett L. Huber

The performance of instruction-level parallel systems can be improved by compiler programs that order machine operations to increase system parallelism and reduce execution time. The optimization, called instruction scheduling, is typically classified as local scheduling if only basic-block context is considered, or as global scheduling if a larger context is used. Global scheduling is generally thought to give better results. One global method, dominator-path scheduling, schedules paths in a function's dominator tree. Unlike many other global scheduling methods, dominator-path scheduling does not require copying of operations to preserve program semantics, making this method attractive for superscalar architectures that provide a limited amount of instruction-level parallelism. In a small test suite for the Alpha 21164 superscalar architecture, dominator-path scheduling produced schedules requiring 7.3 percent less execution time than those produced by local scheduling alone.

Many of today's computer applications require computation power not easily achieved by computer architectures that provide little or no parallelism. A promising alternative is the parallel architecture, more specifically, the instruction-level parallel (ILP) architecture, which increases computation during each machine cycle. ILP computers allow parallel computation of the lowest level machine operations within a single instruction cycle, including such operations as memory loads and stores, integer additions, and floating-point multiplications. ILP architectures, like conventional architectures, contain multiple functional units and pipelined functional units; but, they have a single program counter and operate on a single instruction stream. Compaq Computer Corporation's AlphaServer system, based on the Alpha 21164 microprocessor, is an example of an ILP machine.

To effectively use parallel hardware and obtain performance advantages, compiler programs must identify the appropriate level of parallelism. For ILP architectures, the compiler must order the single instruction stream such that multiple, low-level operations execute simultaneously whenever possible. This ordering by the compiler of machine operations to effectively use an ILP architecture's increased parallelism is called *instruction scheduling.* It is an optimization not usually found in compilers for non-ILP architectures.

Instruction scheduling is classified as *local* if it considers code only within a basic block and *global* if it schedules code across multiple basic blocks. A disadvantage to local instruction scheduling is its inability to consider context from surrounding blocks. While local scheduling can find parallelism within a basic block, it can do nothing to exploit parallelism between basic blocks. Generally, global scheduling is preferred because it can take advantage of added program parallelism available when the compiler is allowed to move code across basic block boundaries. Tjaden and Flynn,[1] for example, found parallelism within a basic block quite limited. Using a test suite of scientific programs, they measured an average parallelism of 1.8 within basic blocks. In similar experiments on scientific pro-

grams in which the compiler moved code across basic block boundaries, Nicolau and Fisher[2] found parallelism that ranged from 4 to a virtually unlimited number, with an average of 90 for the entire test suite.

Trace scheduling[3,4] is a global scheduling technique that attempts to optimize frequently executed paths of a program, possibly at the expense of less frequently executed paths. Trace scheduling exploits parallelism within sequential code by allowing massive migration of operations across basic block boundaries during scheduling. By addressing this larger scheduling context (many basic blocks), trace scheduling can produce better schedules than techniques that address the smaller context of a single block. To ensure the program semantics are not changed by interblock motion, trace scheduling inserts copies of operations that move across block boundaries. Such copies, necessary to ensure program semantics, are called *compensation copies.*

The research described here is driven by a desire to develop a global instruction scheduling technique that, like trace scheduling, allows operations to cross block boundaries to find good schedules and that, unlike trace scheduling, does not require insertion of compensation copies. Like trace scheduling, DPS first defines a multiblock context for scheduling and then uses a local instruction scheduler to treat the larger context like a single basic block. Such a technique provides effective schedules and avoids the performance cost of executing compensation copies. The global scheduling technique described here is based on the dominator relation* among the basic blocks of a function and is called dominator-path scheduling (DPS).

Local Instruction Scheduling

Since DPS relies on a local instruction scheduler, we begin with a brief discussion of the local scheduling problem. As the name implies, local instruction scheduling attempts to maximize parallelism within each basic block of a function's control flow graph. In general, this optimization problem is NP-complete.[5] However, in practice, heuristics achieve good results. (Landskov et al.[6] give a good survey of early instruction scheduling algorithms. Allan et al.[7] describe how one might build a retargetable local instruction scheduler.)

List scheduling[8] is a general method often used for local instruction scheduling. Briefly, list scheduling typically requires two phases. The first phase builds a directed acyclic graph (DAG), called the data dependence DAG (DDD), for each basic block in the function. DDD nodes represent operations to be scheduled. The DDD's directed edges indicate that a node X preceding a node Y constrains X to occur no later than Y. These DDD edges are based on the formalism of data dependence analysis. There are three basic types of data dependence, as described by Padua et al.[9]

- Flow dependence, also called true dependence or data dependence. A DDD node M_2 is flow dependent on DDD node M_1 if M_1 executes before M_2 and M_1 writes to some memory location read by M_2.
- Antidependence, also called false dependence. A DDD node M_2 is antidependent on DDD node M_1 if M_1 executes before M_2 and M_2 writes to a memory location read by M_1, thereby destroying the value needed by M_1.
- Output dependence. A DDD node M_2 is output dependent on DDD node M_1 if M_1 executes before M_2 and M_2 and M_1 both write to the same location.

To facilitate determination and manipulation of data dependence, the compiler maintains, for each DDD node, a set of all memory locations *used* (read) and all memory locations *defined* (written) by that particular DDD node.

Once the DDD is constructed, the second phase begins when list scheduling orders the graph's nodes into the shortest sequence of instructions, subject to (1) the constraints in the graph, and (2) the resource limitations in the machine (i.e., a machine is typically limited to holding only a single value at any time). In general list scheduling, an ordered list of tasks, called a *priority list,* is constructed. The priority list takes its name from the fact that tasks are ranked such that those with the highest priority are chosen first. In the context of local instruction scheduling, the priority list contains DDD nodes, all of whose predecessors have already been included in the schedule being constructed.

Expressions, Statements, and Operations

Within the context of this paper, we discuss algorithms for code motion. Before going further, we need to ensure common understanding among our readers for our use of terms such as *expressions, statements,* and *operations.* To start, we consider a computer program to be a list of operations, each of which (possibly) computes a right-hand side (rhs) value and assigns the rhs value to a memory location represented by a left-hand side (lhs) variable. This can be expressed as

$$A \leftarrow E$$

where A represents a single memory location and E represents an expression with one or more operators and an appropriate number of operands. During different phases of a compiler, operations might be represented as

- Source code, a high-level language such as C
- Intermediate statements, a linear form of three-address code such as quads or *n*-tuples[10]

*A basic block, D, dominates another block, B, if every path from the root of the control-flow graph for a function to B must pass through D.

- DDD nodes, nodes in a DDD, ready to be scheduled by the instruction scheduler

Important to note about operations, whether represented as intermediate statements, source code, or DDD nodes, is that operations include both a set of definitions and a set of uses.

Expressions, in contrast, represent the rhs of an operation and, as such, include uses but not definitions. Throughout this paper, we use the terms *statement, intermediate statement, operation,* and *DDD node* interchangeably, because they all represent an operation, with both uses and definitions, albeit generally at different stages of the compilation process. When we use the term *expression,* however, we mean an rhs with uses only and no definition.

Dominator Analysis Used in Code Motion

In order to determine which operations can move across basic block boundaries, we need to analyze the source program. Although there are some choices as to the exact analysis to perform, dominator-path scheduling is based upon a formalism first described by Reif and Tarjan.[11] We summarize Reif and Tarjan's work here and then discuss the enhancements needed to allow interblock movement of operations.

In their 1981 paper, Reif and Tarjan provide a fast algorithm for determining the approximate *birthpoints* of expressions in a program's flow graph. An expression's birthpoint is the first block in the control flow graph at which the expression can be computed, and the value computed is guaranteed to be the same as in the original program. Their technique is based upon fast computation of the *idef* set for each basic block of the control flow graph. The *idef* set for a block B is that set of variables defined on a path between B's immediate dominator and B. Given that the dominator relation for the basic blocks of a function can be represented as a *dominator tree,* the immediate dominator, IDOM, of a basic block B is B's parent in the dominator tree.

Expression birthpoints are not sufficient to allow us to safely move entire operations from a block to one of its dominators because birthpoints address only the movement of expressions, not definitions. Operations in general include not only a computation of some expression but the assignment of the value computed to a program variable. Ensuring a "safe" motion for an expression requires only that no expression operand move above any *possible definition* of that operand, thus changing the program semantics. A similar requirement is necessary, but not sufficient, for the variable to which the value is being assigned. In addition to not moving A above any previous definition of A, A cannot move above any possible use of A. Otherwise, we run the risk of changing A's value for that previous use. Thus, dominator analysis computes the *iuse* set for each basic block and for the *idef* set. The *iuse* set for a block, B, is that set of variables used on some path between B's immediate dominator and B. Using the *idef* and *iuse* sets, dominator analysis computes an approximate birthpoint for each operation.

In this paper, we use the term *dominator analysis* to mean the analysis necessary to allow code motion of operations while disallowing compensation copies. Additionally, we use the term *dominator motion* for the general optimization of code motion based upon dominator analysis.

Enhancing the Reif and Tarjan Algorithm

By enhancing Reif and Tarjan's algorithm to compute *birthpoints* of operations instead of expressions, we make several issues important that previously had no effect upon Reif and Tarjan's algorithm. This section motivates and describes the information needed to allow dominator motion, including the *use, def, iuse,* and *idef* sets for each basic block. An algorithmic description of this dominator analysis information is included in the section Overview of Dominator-Path Scheduling and the Algorithm for Interblock Motion.

When we allow code motion to move intermediate statements (or just expressions) from a block to one of its dominators, we run the risk that the statement (expression) will be executed a different number of times in the dominator block than it would have been in its original location. When we move only expressions, the risk is acceptable (although it may not be efficient to move a statement into a loop) since the value needed at the original point of computation is preserved. Relative to program semantics, the number of times the same value is computed has no effect as long as the correct value is computed the *last* time. This accuracy is guaranteed by expression birthpoints.

Consider also the consequences of moving an expression from a block that is *never* executed for some particular input data. Again, it may not be efficient to compute a value never used, but the computation does not alter program semantics. When dominator motion moves entire statements, however, the issue becomes more complex. If the statement moved assigns a new value to an induction variable, as in the following example,

$$n = n + 1$$

dominator motion would change n's final value if it moved the statement to a block where the execution frequency differed from that of its original block. We could alleviate this problem by prohibiting motion of any statement for which the *use* and *def* sets are not disjoint, but the possibility remains that a statement may define a variable based indirectly upon that variable's previous value. To remedy the more general problem, we disallow motion of any statement, S,

whose *def* set intersects with those variables that are *used-before-defined* in the basic block in which S resides.

Suppose the optimizer moves an intermediate statement that defines a global variable from a block that may never be executed for some set of input data into a dominator block that *is* executed at least once for the same input data. Then the optimized version has defined a variable that the unoptimized function did not, possibly changing program semantics. We can be sure that such motion does not change the semantics of that function being compiled; but there is no mechanism, short of compiling the entire program as a single unit, to ensure that defining a global variable in this function will not change the value used in another function. Thus, to be conservative and ensure that it does not change program semantics, dominator motion prohibits interblock movement of any statement that defines a global variable. At first glance, it may seem that this prohibition cripples dominator motion's ability to move any intermediate statements at all; but we shall see that such is not the case.

One final addition to Reif and Tarjan information is required to take care of a subtle problem. As discussed above, dominator analysis uses the *idef* and *iuse* sets to prevent illegal code motion. The use of these sets was assumed to be sufficient to ensure the legality of code motion into a dominator block; unfortunately, this is not the case. The problem is that a definition might pass through the immediate dominator of B to *reach* a *use* in a sibling of B in the dominator tree. If there were a definition of this variable in B, but the variable was not defined on any path from the immediate dominator, there would be nothing in dominator analysis to prevent the definition from being moved into the dominator. But that would change the program's semantics. Figure 1 shows the control-flow graph for a function called findmax(), with only the statements referring to register r7. Register r7 is defined in blocks B3 and B7, and referenced in B9. This means that r7 is *live-out* of B5 and *live-in* to B8, but not *live-in* to B7; there is a definition of r7 in B3 that *reaches* B8. Because there is no definition or use between B7 and its immediate dominator B5, the *idef* and *iuse* sets of B7 are empty; thus, dominator analysis, as described above, would allow the assignment of r7 to move upward to block B5. This motion is illegal; it changes the definition in B3. Moving the operation from B7 to B5 changes the conditional assignment of r7 to an unconditional one.

To prevent this from happening, we can insert the variable into the *iuse* set of the block B, in which we wish the statement to remain. We do not, however, want to add to the *iuse* set unnecessarily. The solution is to add each variable, V, that is *live-in* to any of B's siblings in the dominator tree, but not into B, or to B's

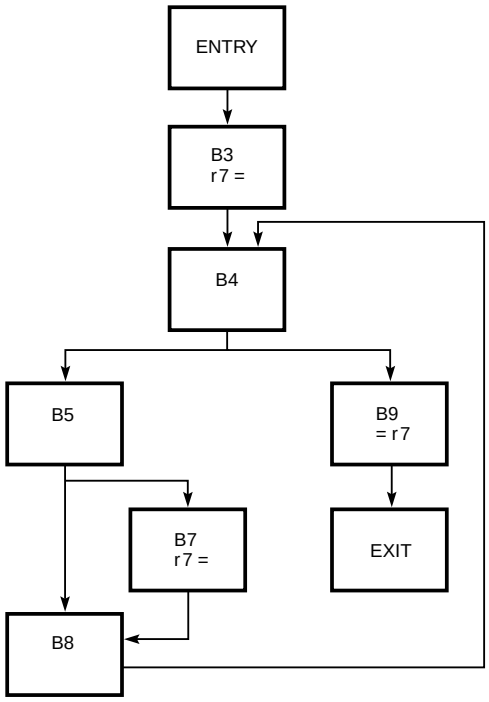


Figure 1
Control Flow Graph for the Function findmax()

iuse set. This will prevent any definition of V that might exist in B from moving up. If there is a definition of V in B, but V is *live-in* to B, there must be some use of V in B before the definition, so it could not move upward in any case.

Measurement of Dominator Motion

To measure the motion possible in C programs, Sweany[12] defined dominator motion as the movement of each intermediate statement to its birthpoint as defined by dominator analysis and by the number of dominator blocks each statement jumps during such movement. Sweany's choice of intermediate statements (as contrasted with source code, assembly language, or DDD nodes) is attributed to the lack of machine resource constraints at that level of program abstraction. He envisioned dominator motion as an upper bound on the motion available in C programs when compensation copies are included. In the test suite of 12 C programs compiled, more than 25 percent of all intermediate statements moved at least one dominator block upwards toward the root of the dominator tree. One function allowed more than 50 percent of the statements to be hoisted an average of nearly eight dominator blocks. The considerable amount of motion (without copies) available at the intermediate statement level of program abstraction

provided us with the motivation to use similar analysis techniques to facilitate global instruction scheduling.

Overview of Dominator-path Scheduling and the Algorithm for Interblock Motion

Since experiments show that dominator analysis allows considerable code motion without copies, we chose to use dominator analysis as the basis for the instruction scheduling algorithm described here, namely dominator-path scheduling. As noted above, DPS is a global instruction scheduling method that does not require copies of operations that move from one basic block to another. DPS performs global instruction scheduling by treating a group of basic blocks found on a dominator tree path as a single block, scheduling the group as a whole. In this regard, it resembles trace scheduling, which schedules adjacent basic blocks as a single block. DPS's foundation is scheduling instructions while moving operations among blocks according to both the opportunities provided by and the restrictions imposed by dominator analysis.

The question arises as to how to exploit dominator analysis information to permit code motion at the instruction level during scheduling. DPS is based on the observation that we can use *idef* and *iuse* sets to allow operations to move from a block to one of its dominators during instruction scheduling. Instruction scheduling can then choose the most advantageous position for an operation that is placed in any one of several blocks. Because machine operations are incorporated in nodes of the DDD used in scheduling and, like intermediate statements, DDD nodes are represented by *def* and *use* sets, the same analysis performed on intermediate statements can also be applied to a basic block's DDD nodes.

The same motivation that drives trace scheduling—namely that scheduling one large block allows better use of machine resources than scheduling the same code as several smaller blocks—also applies to DPS. In contrast to trace scheduling, DPS does not allow motion of DDD nodes when a copy of a node is required and does not incur the code explosion due to copying that trace scheduling can potentially produce. For architectures with moderate instruction-level parallelism, DPS may produce better results than trace scheduling, because the more limited motion may be sufficient to make good use of machine resources, and unlike trace scheduling, no machine resources are devoted to executing semantic-preserving operation copies.

Much like traces,* the dominator path's blocks can be chosen by any of several methods. One method is a heuristic choice of a path based on length, nesting depth, or some other program characteristic. Another is programmer specification of the most important paths. A third is actual profiling of the running program. We visit this issue again in the section Choosing Dominator Paths. First, however, we need to discuss the algorithmic details of DPS.

Once DPS selects a dominator path to schedule, it requires a method to combine the blocks' DDDs into a single DDD for the entire dominator path. In our compiler, this task is performed by a DDD coupler,[13] which is designed for the purpose. Given the DDD coupler, DPS proceeds by repeatedly

- Choosing a dominator path to schedule
- Using the DDD coupler to combine each block's DDD on the chosen dominator path
- Scheduling the combined DDD as a single block

The dominator-path scheduling algorithm, detailed in this section, is summarized in Figures 2 and 3.

A significant aspect of the DPS process is to ensure "appropriate" interblock motion of DDD nodes and to prohibit "illegal" motion. As noted earlier, the combined DDD for a dominator path includes control flow. Therefore, when DPS schedules a group of blocks represented by a single DDD, it needs a mechanism to map correctly the scheduled instructions to the basic blocks. The mechanism is easily accomplished by the addition of two special nodes to each block's DDD. Called BlockStart and BlockEnd, these special nodes represent the basic block boundaries. Since dominator-path scheduling does not allow branches to move across block boundaries, each BlockStart and BlockEnd node is initially "tied" (with DDD arcs) to the branch statement of the block, if any. Because BlockStart and BlockEnd are nodes in the eventually combined DDD, they are scheduled like all other nodes of the combined DDD. After scheduling, all instructions between the instruction containing the BlockStart node for a block and the instruction containing the BlockEnd node for that block are considered instructions for that block. Next, DPS must ensure that the BlockStart and BlockEnd DDD nodes remain ordered (in the scheduled instructions) relative to one another and to the BlockStart and BlockEnd nodes for any other block. To do so, DPS adds *use* and *def* information to the nodes to represent a pseudoresource, BlockBoundary. Because each BlockStart node defines BlockBoundary and each BlockEnd node uses BlockBoundary, no BlockEnd node can be scheduled ahead of its associated BlockStart node (because of flow dependence.) Also, a BlockStart node cannot be scheduled before its dominator block's BlockEnd node (because of antidependence). By establishing these imaginary dependencies, DPS ensures that the DDD coupler adds arcs between all BlockStart and BlockEnd nodes.

*groups of blocks to be scheduled together in trace scheduling

```
Algorithm Dominator-Path Scheduling
Input:
        Function Control Flow Graph
        Dominator Tree
        Post-Dominator Tree

Output:
        Scheduled instructions for the function

Algorithm:
        While at least one Basic Block is unscheduled
                Heuristically choose a path B1, B2,…, Bn in the Dominator Tree that includes
                only unscheduled Basic Blocks.

                Perform dominator analysis to compute IDef and IUse sets

                        /* Build one DDD for the entire dominator path */
                CombinedDDD = B1
                For i = 2 to n
                        T = InitializeTransitionDDD (Bi-1, Bi)
                        CombinedDDD = Couple(CombinedDDD,T)
                        CombinedDDD = Couple (CombinedDDD, Bi )

                Perform list scheduling on CombinedDDD
                Mark each block of DP scheduled
                Copy scheduled instructions to the Blocks of the path (instructions between the
                BlockStart and BlockEnd nodes for a Block are "written" to that Block)
        End While
```

Figure 2
Dominator-path Scheduling Algorithm

Looking back to dominator analysis, we see that interblock motion is prohibited if the operation being moved

- Defines something that is included in either the *idef* or *iuse* set
- Uses something included in the *idef* set for the block in which the operation currently resides

To obtain the same prohibitions in the combined DDD, we add the *idef* set for a basic block, B, to the *def* set B's BlockStart node. Similarly, we add the *iuse* set for B to the *use* set of B's BlockStart node. Thus we enforce the same restriction on movement that dominator analysis imposed upon intermediate statements and ensure that any interblock motion preserves program semantics. In a similar manner, DPS includes the restrictions on movement of operations that define either global variables or induction variables. Figure 3 gives an algorithmic description of the process of "doping" the BlockStart and BlockEnd nodes to prevent disallowed code motion.

DPS is complicated by factors not relevant for dominator motion of intermediate statements. Foremost is the complexity imposed by the bidirectional motion of operations that instruction scheduling allows. In dominator motion, intermediate statements move in only one direction, i.e., toward the top of the function's control flow graph, not from a dominator block to a dominated one. This one-directional motion is reasonable when attempting to move intermediate statements because one statement's movement will likely open possibilities for more motion in the same direction by other statements. When statements move in different directions, one statement's motion might inhibit another's movement in the opposite direction. The goal of dominator motion is to move statements as far as possible in the control flow graph. In contrast, the goal of DPS is not to maximize code motion, but rather to find, for each operation, O, that location for O that will yield the shortest schedule. Thus our goal has changed from that of dominator motion. To gain the full benefit from DPS, we wish to allow operations to move past block boundaries in either direction. To permit bidirectional motion, we use the post-dominator relation, which says that a basic block, PD, is a post-dominator of a basic block B if all paths from B to the function's exit must pass through PD. Using this strategy, we similarly define *post-idef* and *post-iuse* sets. In

```
Algorithm InitializeTransitionDDD(B1, B2)
Input:
        A Transition DDD templates, with a Dummy DDDNode
        for B1's block end and one for B2's block start
        Two basic blocks, B1 and B2 that we wish to couple
        Dominator Tree
        Post-Dominator Tree
        The following dataflow information
                Def, Use, IDef, and IUse sets for B1 and B2
                Used-Before-Defined set for B2
                Post-IDef, and Post-IUse sets for B1 and B2
                B2's "sibling" set, defined to include any variable
                        live-in to a dominator-tree sibling of B2, but not
                        live-in to B2
                A basic block DDD for each of B1 and B2
Output:
        An initialized Transition DDD, T
Algorithm:
        T = TransitionDDD
        /* "Fix" set for global and induction variables. */
        Add set of global variables to B2's IUse
        Add B2's Used-Before-Defined to B2's IUse
        Add B2's sibling set to B2's IUse

        If B2 does not post-dominate B1
                Add B1's Use set to T's BlockEnd Def set
                Add B1's Def set to T's BlockEnd Use set
        Else
                Add B1's Post-IDef set to T's BlockEnd Def set
                Add B1's Post-IUse set to T's BlockEnd Use set
        Add B2's IDef set to T's BlockStart Def set
        Add B2's IUse set to T's BlockStart Use set
        Return T
```

Figure 3
Initialize Transition DDD Algorithm

fact, it is not difficult to compute all these quantities for a function. The simplest way is to logically reverse the direction of all the control flow graph arcs and perform dominator analysis on the resulting graph. Having computed the post-dominator tree, DPS chooses dominator paths such that the dominated node is a post-dominator of its immediate predecessor in a dominator path. This choice allows operations to move "freely" in both directions. Of course, this may be too limiting on the choice of dominator paths. To allow for the possibility that nodes in a dominator path will not form a post-dominator relation, DPS needs a mechanism to limit bidirectional motion when needed. Again, we rely on the technique of adding dependencies to the combined DDD. In this case (assuming that DPS is scheduling paths in the forward dominator tree), for any basic block, B, whose successor, S, in the forward dominator path does not post-dominate B, DPS adds B's *def* set to the *use* set of the BlockEnd node associated with B. In similar fashion, we add B's *use* set to B's BlockEnd node's *def* set. This technique prevents any DDD node originally in B from moving downward in the dominator path.

Choosing Dominator Paths

DPS allows code movement along any dominator path, but there are many ways to select these paths. An investigation of the effects of dominator-path choice on the efficiency of generated schedules tells us that the choice of path is too important to be left to arbitrary selection; twice the average percent speedup* for several functions can often be achieved with a simple,

*(unoptimized_speed – optimized_speed)/unoptimized_speed

well-chosen heuristic. Some functions have a potential percent speedup almost four times the average. Thus, it is important to find a good, generally applicable heuristic to select the dominator paths.

Unfortunately, it is not practical to schedule all of the possible partitionings for large functions. If we allow a basic block to be included in only one dominator path, the formula for the number of distinct partitionings of the dominator tree is

$$\prod_{n \in N} [\text{outdeg}(n) + 1]$$

where N is the set of nodes of the dominator tree.[14] Although the number of possible paths is not prohibitive for small dominator trees, larger trees have a prohibitively large number. For example, whetstone's main(), with 49 basic blocks, has almost two trillion distinct partitionings.

To evaluate differences in dominator-path choices, we scheduled a group of small functions with DPS using every possible choice of dominator path. The target architecture for this study was a hypothetical 6-wide long-instruction-word (LIW) machine, which was simulated and in which it was assumed that all cache accesses were hits.

The results of exhaustive dominator-path testing show, as expected, that varying the choice of dominator paths significantly affects the performance of scheduling. For all functions of at least two basic blocks, DPS showed improvement over local scheduling for at least one of the possible choices of dominator paths. Table 1 shows the best, average, and worst percent speedup over local scheduling found for all functions that had a "best" speedup of over 2 percent; it also shows the speedup of the original implementation of DPS and the number of distinct dominator tree partitionings. The original implementation of DPS included a single, simple heuristic to choose dominator paths. More specifically, to choose dominator paths within a group, G, of contiguous blocks at the same nesting level, the compiler continues to choose a block, B, to "expand." Expansion of B initializes a new dominator path to include B and adds B's dominators until no more can be added. The algorithm then starts another dominator path by expanding another (as yet unexpanded) block of G. The first block of G chosen to expand is the tail block, T, in an attempt to obtain as long a dominator path as possible.

Unfortunately, not all functions are small enough to be tested by performing DPS for each possible partitioning of the dominator tree. Therefore, we defined 37 different heuristic methods of choosing dominator trees, based upon groupings of six key heuristic factors.

The maximum path lengths of the basic guidelines were adjusted to produce actual heuristics. We used the heuristic factors from which the individual heuristics were constructed; each seemed likely either to mimic the observed characteristics of the best path selection or to allow more freedom of code motion and, therefore, more flexibility in filling "gaps."

- One nesting level—Group blocks from the same nesting level of a loop. Each block is in the same strongly connected component, so the blocks tend to have similar restrictions to code motion. For a group of blocks to be a strongly connected component, there must be some path in the control flow graph from each node in the component to all the other nodes in the component. Since the function will probably repeat the loop, it seems likely that the scheduler will be able to overlap blocks in it.

Table 1
Percent of Function Speedup Improvement Using DPS Path Choices over Local Scheduling

	Percent Speedup				
Function Name	**Best**	**Average**	**Worst**	**Original**	**No. Dominator Tree Partitions**
bubble	39.2	10.6	– 0.1	11.7	72
readm	32.5	9.3	– 0.2	32.5	48
solve	27.8	9.9	– 0.2	27.8	96
queens	25.4	8.3	– 0.4	– 0.4	96
swaprow	23.1	5.8	– 3.7	19.5	24
print(g)	22.0	9.1	– 0.2	22.0	8
findmax	21.3	6.2	– 0.3	8.7	18
copycol	18.5	5.6	– 5.0	19.9	8
elim	14.3	2.3	– 3.8	10.2	576
mult	13.7	2.1	– 3.8	10.3	96
subst	12.9	2.4	– 4.9	4.9	96
print(8)	12.5	6.2	0.0	12.5	8

- Longest path—Schedule the longest available path. This heuristic class allows the maximum distance for code motion.
- Postdominator—Follow the postdominator relation in the dominator tree. When a dominator block, P, is succeeded by a non-postdominator block, S, our compiler adds P's *def* set to the *use* set of P's BlockEnd node and the *use* set to the *def* set to prevent any code motion from P to S. If P is instead succeeded by its postdominator block, no such modification is necessary, and code would be allowed to move in both directions. Intuitively, the postdominator relation is the exact inverse of the dominator relation, so code can move down, into a postdominator, as it moves up into a dominator. Further, the simple act of adding nodes to the DDD will complicate list scheduling, making it harder for the scheduler to generate the most efficient schedule.
- Non-postdominator—Follow a non-postdominator in the dominator tree. This heuristic class generally means adding loop body blocks to the path. Notice that this seems at odds with the previous heuristic class. The previous class was suggested by intuition about the scheduler, and this one by observation of path behavior.
- *idef* size—Group by *idef* set size. The larger the *idef* size, the more interference there is to code motion. A small *idef* size will probably allow more code motion, so we try to add blocks with small *idef* sizes.
- Density—Group by operation density. We define the density of each basic block as the number of nodes in the DDD divided by the number of instructions required for local scheduling. A dense block already has close to its maximum number of operations; adding or removing operations will probably not improve the schedule. For this reason, we want to avoid scheduling dense blocks together. Two methods are tried: scheduling dense blocks with sparse blocks and putting sparse blocks together.

The heuristic factors were used to make individual heuristics by changing the limit on the possible number of blocks in a path. It was reasonable to set limits for four factors: postdominator, non-postdominator, *idef* size, and density. We tried path length limits in blocks of 2, 3, 4, 5, and unlimited, making a total of five heuristics from each heuristic factor.

Running DPS using each of the heuristic methods and comparing the efficiency of the resulting code leads to several conclusions about effective heuristics for choosing DPS's dominator paths. For some heuristics, we can achieve the best schedules for DPS by using paths that rarely exceed three blocks. For any particular class of heuristics, we can achieve the best schedule with paths limited to five blocks or fewer. Consequently, path lengths can be limited without lowering the efficiency of generated code, and longer paths, which increase scheduling time, can be avoided.

Since no one heuristic performed well for all functions, we advise using a combination of heuristics, i.e., schedule by using each of three heuristics and taking the best schedule. The "combined" heuristic includes the following:

- Instruction density, limit to five blocks
- One nesting level on path, limit to five blocks
- Non-postdominator, unlimited length

Frequency-based List Scheduling

Like some other global schedulers, DPS uses a local scheduling algorithm (list scheduling) on a global context, namely the meta-blocks built by DPS. This algorithm raises the possibility of moving code from less frequently executed blocks to more frequently executed blocks. At first glance, this practice seems to be a bad idea.

In theory, to best schedule any meta-block, an instruction scheduler must account for the differing cost of the instructions within the meta-block. If a single meta-block includes multiple nesting levels, the scheduler must recognize that instructions added to blocks with higher nesting levels are more costly than those added to blocks with lower nesting levels. Even within a loop, there exists the potential for considerable variation in the execution frequencies of different blocks in the meta-block due to control flow. Of course variable execution frequency is not an issue in traditional local scheduling because, within the context of a single basic block, each DDD node is executed the same number of times, namely, once each time execution enters the block.

To address the issue of differing execution frequencies within meta-blocks scheduled as a single block by DPS, we investigated frequency-based list scheduling (FBLS),[15] an extension of list scheduling that provides an answer to this difficulty by considering that execution frequencies differ within sections of the meta-blocks. FBLS uses a greedy method to place DDD nodes in the lowest-cost instruction possible. FBLS amends the basic list-scheduling algorithm by revising only the DDD node placement policy in an attempt to reduce the run-time cycles required to execute a meta-block.

Unfortunately, although FBLS makes intuitive sense, we found that DPS produced worse schedules with FBLS than it produced with a naive local scheduling algorithm that ignored frequency differences within DPS's meta-blocks. Therefore, the current implementation of DPS ignores the execution frequency differences between basic blocks, both in choosing dominator paths to schedule and in scheduling those dominator-path meta-blocks.

Evaluation of Dominator-path Scheduling

To measure the potential of DPS to generate more efficient schedules than local scheduling for commercial superscalar architectures, we ran a small test suite of C programs on an Alpha 21164 server. The Alpha server is a superscalar architecture capable of issuing two integer and two floating-point instructions each cycle. Our compiler estimates the effectiveness of a schedule by modeling the 21164 as an LIW architecture with all operation latencies known at compile time. Of course this model was used only within the compiler itself. Our results measured changes in 21164 execution time (measured with the UNIX "time" command) required for each program.

Our test suite of 14 C programs includes 8 programs that use integer computation only and 6 programs that include floating-point computation. We separated those groups because we see dramatic differences in DPS's performance when viewing integer and floating-point programs. To choose dominator paths, we used the combined heuristic recommended by Huber.[14]

Table 2 summarizes the results of tests we conducted to compare the execution times of programs using DPS scheduling with those using local scheduling only. The table lists the programs used in the test suite and the percent improvement in execution times for DPS-scheduled programs. The execution time measurements were made on an Alpha 21164 server running at 250 megahertz with data cache sizes of 8 kilobytes, 96 kilobytes, and 4 megabytes.

Table 2
Percent DPS Scheduling Improvements over Local Scheduling of Programs

Program	Percent Execution Time Improvement
8- Queens	7.3
SymbolTable	7.3
BubbleSort	5.0
Nsieve	6.1
Heapsort	6.0
Killcache	2.6
TSP	2.4
Dhrystone	0.7
C integer average	**4.7**
Dice	3.7
Whetstone	5.4
Matrix Multiply	16.2
Gauss	12.3
Finite Difference	17.6
Livermore	9.3
C floating-point average	**10.8**
Overall average	**7.3**

Looking at Table 2, we see that, in general, DPS improved the integer programs less than it improved the floating-point programs. The range of improvements for integer programs was from 0.7 percent for Dhrystone to 7.3 percent each for 8-Queens and for SymbolTable. Summing all the improvements and dividing by eight (the number of integer programs) gives an "average" of 4.7 percent improvement for the integer programs. DPS improved some of the floating-point programs even more significantly than the integer programs. The range of improvements for the six floating-point programs was from 3.7 percent for Dice (a simulation of rolling a pair of dice 10,000,000 times using a uniform random number generator) to 17.6 percent improvement for the finite difference program. The average for the six floating-point programs was 10.8 percent. This suggests, not surprisingly, that the Alpha 21164 provides more opportunities for global scheduling improvement when floating-point programs are being compiled.

Even within the six floating-point programs, however, we see a distinct bi-modal behavior in terms of execution-time improvement. Three of the programs range from 12.3 percent to 17.6 percent improvement, whereas three are below 10 percent (and two of those significantly below 10 percent). A reason for this wide range is the use of global variables. Remember that DPS forbids the motion of global variable definitions across block boundaries. This is necessary to ensure correct program semantics. It is hardly a coincidence that both Dice and Whetstone include only global floating-point variables, whereas Livermore's floating-point variables are mixed about half local and half global, and the three better performers use almost no global variables. Thus we conclude that, for floating-point programs with few global variables, we can expect improvements of roughly 12 to 15 percent in execution time. Inclusion of global variables and exclusion of floating-point values will, however, decrease DPS's ability to improve execution time for the Alpha 21164.

Related Work

As we have discussed, local instruction scheduling can find parallelism within a basic block but cannot exploit parallelism between basic blocks. Several global scheduling techniques are available, however, that extract parallelism from a program by moving operations across block boundaries and subsequently inserting compensation copies to maintain program semantics. Trace scheduling[3] was the first of these techniques to be defined. As previously mentioned, trace scheduling

requires compensation copies. Other "early" global scheduling algorithms that require compenstation copies include Nicolau's *percolation scheduling*[16,17] and Gupta's *region scheduling*.[18] A recent and quite popular extension of trace scheduling is Hwu's SuperBlock scheduling.[19,20] In addition to these more general, global scheduling methods, significant results have been obtained by software pipelining, which is a technique that overlaps iterations of loops to exploit available ILP. Allan et al.[21] provide a good summary, and Rau[22] provides an excellent tutorial on how *modulo scheduling,* a popular software pipelining technique, should be implemented. Promising recent techniques have focused on defining a meta-environment, which includes both global scheduling and software pipelining. Moon and Ebcioglu[23] present an aggressive technique that combines software pipelining and global code motion (with copies) into a single framework. Novak and Nicolau[24] describe a sophisticated scheduling framework in which to place software pipelining, including alternatives to modulo scheduling. While providing a significant number of excellent global scheduling alternatives, none of these techniques provides global scheduling without the possibility of code expansion (copy code) as DPS does.

To address the issue of producing schedules without operation copies, Bernstein[25–27] defined a technique he calls *global instruction scheduling* (GPS) that allows movement of instructions beyond block boundaries based upon the program dependence graph (PDG).[28] In a test suite of four programs run on IBM's RS/6000, Bernstein's method showed improvement of roughly 7 percent over local scheduling for two of the programs, with no significant difference for the others.

Comparing DPS to Bernstein's method, we see that both allow for interblock motion without copies. Bernstein also allows for interblock movement requiring duplicates that DPS does not. Interestingly, Bernstein's later work[27] does not make use of this ability to allow motion that requires duplication of operations, suggesting that, to date, he has not found such motion advisable for the RS/6000 architecture to which his techniques have been applied. Bernstein allows operation movement in only one direction, whereas DPS allows operations to move from a dominator block to a postdominator. This added flexibility is an advantage to DPS. Of possibly greater significance, DPS uses the local instruction scheduler to place operations. Bernstein uses a separate set of heuristics to move operations in the PDG and then uses a subsequent local scheduling pass to order operations within each block. Fisher[3] argues that incorporating movement of operations with the scheduling phase itself provides better scheduling than dividing the interblock motion and scheduling phases. Based on that criterion alone, DPS has some advantages over Bernestein's method.

Conclusions

It is commonly accepted that to exploit the performance benefits of ILP, global instruction scheduling is required. Several varieties of global instruction scheduling exist, most requiring compensation copies to ensure proper program semantics when operations cross block boundaries during instruction scheduling. Although such global scheduling with compensation copies may be an effective strategy for architectures with large degrees of ILP, another approach seems reasonable for more limited architectures, such as currently available superscalar computers.

This paper outlines DPS, a global instruction scheduling technique that does not require compensation copies. Based on the fact that more than 25 percent of intermediate statements can be moved upward at least one dominator block in the control flow graph without changing program semantics, DPS schedules paths in a function's dominator tree as meta-blocks, making use of an extended local instruction scheduler to schedule dominator paths.

Experimental evidence shows that DPS does indeed produce more efficient schedules than local scheduling for Compaq's Alpha 21164 server system, particularly for floating-point programs that avoid the use of global variables. This work has demonstrated that considerable flexibility in placement of code is possible even when compensation copies are not allowed. Although more research is required to look into possible uses for this flexibility, the global instruction scheduling method described here (DPS) shows promise for ILP architectures.

Acknowledgments

This research was supported in part by an External Research Program grant from Digital Equipment Corporation and by the National Science Foundation under grant CCR-9308348.

References

1. G. Tjaden and M. Flynn, "Detection of Parallel Execution of Independent Instructions," *IEEE Transactions on Computers,* C-19(10) (October 1970): 889–895.

2. A. Nicolau and J. Fisher, "Measuring the Parallelism Available for Very Long Instruction Word Architectures," *IEEE Transactions on Computers,* 33(11) (November 1984): 968–976.

3. J. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers,* C-30(7) (July 1981): 478–490.

4. J. Ellis, *Bulldog: A Compiler for VLIW Architectures* (Cambridge, MA: MIT Press, 1985), Ph.D. thesis, Yale University (1984).

5. D. DeWitt, "A Machine-Independent Approach to the Production of Optimal Horizontal Microcode," Ph.D. thesis, University of Michigan, Ann Arbor, Mich. (1976).

6. D. Landskov, S. Davidson, B. Shriver, and P. Mallett, "Local Microcode Compaction Techniques," *ACM Computing Surveys,* 12(3) (September 1980): 261–294.

7. V. Allan, S. Beaty, B. Su, and P. Sweany, "Building a Retargetable Local Instruction Scheduler," *Software—Practice & Experience,* 28(3) (March 1998): 249–284.

8. E. Coffman, *Computer and Job-Shop Scheduling Theory* (New York: John Wiley & Sons, 1976).

9. D. Padua, D. Kuck, and D. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Transactions on Computers,* C-29(9) (September 1980): 763–776.

10. A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools* (Reading, MA: Addison-Wesley, 1986).

11. H. Reif and R. Tarjan, "Symbolic Program Analysis in Almost-Linear Time," *Journal of Computing,* 11 (1) (February 1981): 81–93.

12. P. Sweany, "Interblock Code Motion without Copies," Ph.D. thesis, Computer Science Department, Colorado State University (1992).

13. R. Mueller, M. Duda, P. Sweany, and J. Walicki, "Horizon: A Retargetable Compiler for Horizontal Microarchitectures," *IEEE Transactions on Software Engineering: Special Issue on Microprogramming,* 14(5) (May 1998): 575–583.

14. B. Huber, "Path-Selection Heuristics for Dominator-Path Scheduling," Master's thesis, Department of Computer Science, Michigan Technological University (1995).

15. M. Bourke, P. Sweany, and S. Beaty, "Extending List Scheduling to Consider Execution Frequency," *Proceedings of the 28th Hawaii International Conference on System Sciences* (January 1996).

16. A. Nicolau, "Percolation Scheduling: A Parallel Compilation Technique," Technical Report TR85-678, Department of Computer Science, Cornell University (May 1985).

17. A. Aiken and A. Nicolau, "A Development Environment for Horizontal Microcode," *IEEE Transactions on Software Engineering,* 14(5) (May 1988): 584–594.

18. R. Gupta and M. Soffa, "Region Scheduling: An Approach for Detecting and Redistributing Parallelism," *IEEE Transactions on Software Engineering,* 16(4) (April 1990): 421–431.

19. S. Mahlke, W. Chen, W.-M. Hwu, B. Rao, and M. Schlansker, "Sentinel Scheduling for VLIW and Superscalar Processors," *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems,* Boston, Mass. (October 1992): 238–247.

20. C. Chekuri, R. Johnson, R. Motwani, B. Natarajan, B. Rau, and M. Schlansker, "Profile-Driven Instruction-Level-Parallel Scheduling with Application to Super Blocks," *Proceedings of the 29th International Symposium on Microarchitecture* (MICRO-29), Paris, France (December 1996): 58–67.

21. V. Allan, R. Jones, R. Lee, and S. Allan, "Software Pipelining," *ACM Computing Surveys,* 27(3) (September 1995).

22. B. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," *Proceedings of the 27th International Symposium on Microarchitecture* (MICRO-27), San Jose, Calif. (December 1994): 63–74.

23. S.-M. Moon and K. Ebcioglu, "Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining," *ACM Transactions on Programming Languages and Systems,* 18(6) (November 1997): 853–898.

24. S. Novak and A. Nicolau, "An Efficient Global Resource-Directed Approach to Exploiting Instruction-Level Parallelism," *Proceedings of the 1996 International Conference on Parallel Architectures and Compiler Techniques* (PACT 96), Boston, Mass. (October 1996): 87–96.

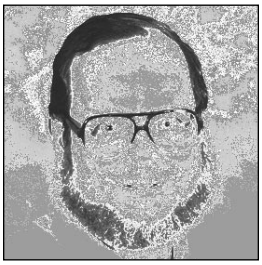
25. D. Bernstein and M. Rodeh, "Global Instruction Scheduling for Superscalar Machines," *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation,* Toronto, Canada (June 1991): 241–255.

26. D. Bernstein, D. Cohen, and H. Krawczyk, "Code Duplication: An Assist for Global Instruction Scheduling," *Proceedings of the 24th International Symposium on Microarchitecture* (MICRO-24), Albuquerque, N. Mex. (November 1991): 103–113.

27. D. Bernstein, D. Cohen, Y. Lavon, and V. Rainish, "Performance Evaluation of Instruction Scheduling on the IBM RS/6000," *Proceedings of the 25th International Symposium on Microarchitecture* (MICRO-25), Portland, Oreg. (December 1992): 226–235.

28. J. Ferrante, K. Ottenstein, and J. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems,* 9(3) (July 1987): 319–349.

Biographies



Philip H. Sweany
Associate Professor Phil Sweany has been a member of Michigan Technological University's Computer Science faculty since 1991. He has been investigating compiler techniques for instruction-level parallel (ILP) architectures, co-authoring several papers on instruction scheduling, register assignment, and the interaction between these two optimizations. Phil has been the primary designer and implementer of Rocket, a highly optimizing compiler that is easily retargetable for a wide range of ILP architectures. His research has been significantly assisted by grants from Digital Equipment Corporation and the National Science Foundation. Phil received a B.S. in computer science in 1983 from Washington State University, and M.S. and Ph.D. degrees in computer science from Colorado State University in 1986 and 1992, respectively.

Steven M. Carr
Steve Carr is an assistant professor in the Department of Computer Science at Michigan Technological University. The focus of his research at the university is memory-hierarchy management and optimization of instruction-level parallel architectures. Steve's research has been supported by both the National Science Foundation and Digital Equipment Corporation. He received a B.S. in computer science from Michigan Technological University in 1987 and M.S. and Ph.D. degrees from Rice University in 1990 and 1993, respectively. Steve is a member of ACM and an IEEE Computer Society Affiliate.

Brett L. Huber
Raised in Hope, Michigan, Brett earned B.S. and M.S. degrees in computer science at Michigan Technological University in Michigan's historic Keweenaw Peninsula. He is an engineer in the Software Development Systems group at Texas Instruments, Inc., and is currently developing an optimizing compiler for the TMS320C6x family of VLIW digital signal processors. Brett is a member of the ACM and an IEEE Computer Society Affiliate.