Dominator-Path Scheduling — A Global Scheduling Method

Philip H. Sweany
Computer Science Department
Michigan Technological University
1400 Townsend Drive
Houghton MI 49931-1295
(906) 487-3392
sweany@cs.mtu.edu

Steven J. Beaty
Department of Mechanical Engineering
Colorado State University
Fort Collins, Colorado 80523
beaty@longs.lance.colostate.edu

Abstract

Dominator-path scheduling performs global instruction scheduling of paths in the dominator tree. Unlike other global scheduling methods, dominator-path scheduling does not require copies of operations to preserve program semantics. In a limited test suite for a typical superscalar architecture, dominator-path scheduling produces schedules requiring 8.3% fewer cycles than local scheduling alone.

1 Introduction

Architectures exhibiting instruction-level parallelism (ILP), such as superscalar and superpipelined machines, are currently popular. To best exploit instruction-level parallelism in these machines, an instruction scheduling phase is required during compilation. Instruction scheduling is typically classified as *local* if it considers code only within a basic block and *global* if it schedules multiple basic blocks at once. Local scheduling methods are well known (see [Bea91] for one summary.) Local instruction scheduling's largest impediment 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.

Trace scheduling [Ell86] and *percolation scheduling* [AN88] are two methods of global instruction scheduling. While trace scheduling and percolation scheduling have several differences, they both allow massive inter-block movement of operations during scheduling, often requiring that the moved operations be copied into multiple blocks to ensure proper program semantics. This motion of operations can exploit a considerable amount of program parallelism, at the expense of executing copied operations multiple times. This appears an effective trade-off for architectures with a large degree of instruction-level parallelism such as VLIW machines, but it is not clear that such motion with copies is as useful for ILP architectures with more limited hardware, such as current superscalar architectures.

To address the issue of producing schedules without operation copies, Bernstein [BR91] defines a technique he calls *global instruction scheduling* which allows movement of instructions beyond block boundaries based upon the program dependence graph (PDG) [FOW87]. Bernstein's method differentiates between inter-block movements which require duplicates of moved operations and those which do not. This paper defines another technique which we call *dominator-path scheduling* (DPS) which allows inter-block motion **only** when no duplicates are required. DPS is based upon the control-flow graph rather than the PDG structure. We shall discuss differences between Bernstein's method and DPS later, after discussion of DPS itself.

2 Dominator Analysis

In their 1981 paper, [RT81] Reif and Tarjan provide a fast algorithm for determining the *approximate birthpoint*s of expressions in a program's flow graph. An expression's birthpoint is the first block in the control flow graph where the expression can be computed while guaranteeing the value computed will 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, $\mathcal{B}$ is that set of variables which are defined on some path between $\mathcal{B}$'s dominator and $\mathcal{B}$.

Reif and Tarjan's expression birthpoints are not sufficient to allow us to safely move entire operations from a block to one of its dominators because they 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. For an operation $A \leftarrow E$, in addition to computing the birthpoint of the right-hand-side expression, we must concern ourselves with the variable being assigned to as well. To ensure a "safe" motion for a rhs ex-

pression, we need only ensure that no expression operand move above any $possible$ definition of that operand, thus changing the program semantics. We need to make a similar requirement for the variable being assigned to, but we must do more. As well as not moving A above any previous definition of A, we must ensure that A does not 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 as well as the *idef* set. Using the *idef* and *iuse* sets, dominator analysis computes an approximate birthpoint for each machine operation.

To measure the motion possible in C programs (at the intermediate statement level of program abstraction), Sweany [Swe92] moved each statement to its birthpoint as defined by dominator analysis and counted the number of dominator blocks each statement jumped during such movement. In a test suite of twelve C programs, more than 25% of all statements moved at least one dominator block upwards towards the root of the dominator tree. One function allowed more than 50% of the statements to be hoisted an average of nearly eight dominator blocks. This considerable amount of motion (without copies) available at the intermediate statement level of program abstraction provides motivation for using similar analysis techniques to facilitate global instruction scheduling.

3 Dominator-Path Scheduling

Dominator-path scheduling is a global instruction scheduling method that does not require copies of operations that move from one block to another. 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.

DPS performs global instruction scheduling by treating a group of basic blocks found on a dominator tree path as a single block, scheduling them as a whole. This allows instruction scheduling to choose the most advantageous position for an operation which we might "legally" place in any one of a number of blocks. Because machine operations are represented by nodes of a data dependence DAG (DDD) used in scheduling and, like intermediate statements, DDD nodes are represented by def and use sets, the same analysis that allows us to move intermediate statements applies to nodes of a basic block's DDD as well.

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 — applies to DPS as well. Much like $traces$ (groups of blocks to be scheduled together in trace scheduling), the dominator path's blocks can be chosen by any of several methods. Heuristically choosing a path based on length, nesting depth, or some other program characteristic is one method. Allowing the programmer to specify the most important paths is another. Actual profiling of the running program is a third.

Once we select a dominator path to schedule, we need a method of combining the blocks' DDDs into a single DDD for the entire dominator path. In our compiler, this is easy, as the DDD coupler (described in [Swe92]) is designed for just such a purpose.

3.1 Algorithm

When combining two DDDs, the coupler will add arcs for any data dependencies which exist between the two DDDs. DPS uses this coupler feature by adding "dummy" nodes to the DDD for each basic block. These dummy nodes include uses and/or definitions of appropriate dataflow values to prevent potentially hazardous motion of DDD nodes across block boundaries during scheduling. When the basic blocks of a dominator path are combined, the coupler automatically inserts those arcs necessary to prevent nodes' "illegal" motion from one block to another.

Because the combined DDD for a dominator path includes control flow we need, when scheduling a group of basic blocks represented by a single DDD, some mechanism to map the scheduled instructions to the correct basic blocks. We can easily accomplish this by adding two special nodes to each block's DDD. These nodes are called BlockStart and BlockEnd. They represent the basic block boundaries. Since BlockStart and BlockEnd are nodes in the eventual combined DDD, they are scheduled just like all the other nodes of the combined DDD. After scheduling, all scheduled instructions between the instruction containing the BlockStart node for a block and the instruction containing the BlockEnd node for that block will be considered instructions for that block. The only remaining chore is to 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, we add use and def information to the nodes to represent a pseudo-resource, BlockBoundary. By initializing each BlockStart node to define BlockBoundary and each BlockEnd node to use BlockBoundary, we ensure that no BlockEnd node can be scheduled ahead of its associated BlockStart node (due to *flow dependence*.) We also ensure that no BlockStart node can be scheduled before its dominator block's BlockEnd node (because of an *anti-dependence*). By establishing these imaginary dependencies, DPS ensures that the DDD coupler will add DDD arcs between all BlockStart and BlockEnd nodes.

In dominator analysis, interblock motion is prohibited if 1) the operation being moved defines something which is included in either the *idef* or *iuse* set or 2) uses something included in the *idef* set for the block where the opera-

tion currently resides. To obtain the same effect in the combined DDD we again turn to using the use and def sets for the BlockStart nodes. By adding the $idef$ set for a basic block $\mathcal{B}$ to the def set of $\mathcal{B}$'s BlockStart node, and similarly adding the $iuse$ set for $\mathcal{B}$ to the use set of $\mathcal{B}$'s BlockStart node, we enforce the same restriction on movement that dominator analysis imposed upon intermediate statements, thus ensuring that any interblock motion preserves program semantics.

DPS is complicated by some factors which are not relevant when moving intermediate statements. Foremost is the added complexity imposed by the bidirectional motion of operations that instruction scheduling allows. In the cited experiments, intermediate statements moved in only one direction — towards the top of the function's control flow graph. There was no concept of a statement moving from a dominator block to a dominated one. To gain the full benefit from DPS, we would like to allow operations to move past block boundaries in either direction. To allow bi-directional motion, we use the "post-dominator" relation which says that a basic block $\mathcal{PD}$ is a post-dominator of a basic block $\mathcal{B}$ if all paths from $\mathcal{B}$ to the function's exit must pass through $\mathcal{PD}$. Using this strategy, we similarly define *post-idef* and *post-iuse* sets. In fact, it is not difficult to compute all these quantities for a function. The problem lies in the fact, that no matter whether we schedule blocks in a dominator path or a post-dominator path, we have no guarantee that the path will be the inverse of some path of the dominator tree computed in the other direction. Thus to allow operations to move "freely" in both directions, the successors of a block in the dominator path must themselves be dominators of that block in the post-dominator tree. Since this cannot always be so, we need some mechanism to limit bi-directional motion when we must. Again, we rely on the technique of adding dependencies to the combined DDD. In this case (assuming that we are scheduling paths in the forward dominator tree), for any basic block, $\mathcal{B}$, whose successor in the forward dominator path is not itself an immediate dominator of $\mathcal{B}$ in the post-dominator tree, we add $\mathcal{B}$'s def set to the use set of the BlockEnd node associated with $\mathcal{B}$. In similar fashion, we add $\mathcal{B}$'s use set to that BlockEnd node's def set. This will prevent any DDD node originally in $\mathcal{B}$ from moving downward in the dominator path. Since this addition of dependencies will hamper instruction scheduling's ability to find a good schedule for the combined DDD, we may wish to choose dominator paths to schedule such that the paths chosen are inverses of some path in the post-dominator tree.

Comparing DPS to Bernstein's *global instruction scheduling* (GPS), we see that both allow for inter-block motion without copies. GPS also allows for inter-block movement requiring duplicates which DPS does not. GPS, however, is currently defined only within the context of a loop, while DPS paths can include blocks of different nesting levels. GPS allows operation movement in only one direction, while DPS allows operations to move from a dominator block to a post-dominator. Finally, global scheduling in DPS uses the local instruction scheduler to place operations. GPS 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.

3.2 Results

To measure the potential of dominator-path scheduling, we have compared it to local scheduling for a small test suite of C programs run on an IBM RS6000 computer. The RS6000 is a popular superscalar architecture which allows the simultaneous start of three operations (an *integer operation*, a *floating point* operation and a *branch* operation) during each execution cycle, should the next three instructions to execute be data-independent integer, floating point and branch operations. To model the RS6000's operation, our compiler inserts NOPs where it determines that the architecture will stall waiting for data to become ready before it can continue. The goal of instruction scheduling then is to minimize the number execution cycles needed by minimizing necessary stalls and by overlapping instructions to best take advantage of the multiple instruction issue capability of the hardware. In short, we model the RS6000 as though it were an LIW architecture, and in that way, we can measure the effectiveness of the scheduling by counting "instructions" (which are in effect execution cycles).

For the test suite of six programs used, we identified 879 dominator paths to be scheduled. In this experiment we picked dominator paths only within a loop, ensuring that all blocks in the path are at the same nesting level. While this conservative strategy is not required by dominator-path scheduling (and indeed, is probably not even advantageous), it was chosen for reasons which will be discussed shortly. For the chosen dominator paths, local scheduling required 9670 cycles, while dominator-path scheduling required 8868. This represents a savings of 8.3% of the instructions required by local scheduling. This is a significant improvement for an architecture with the parallelism of the RS6000. The programs used in the test suite are listed in Table 1 with the number of dominator paths scheduled for each program and the percentage improvement that dominator-path scheduling demonstrated for those paths.

Still it should be recognized that this is, to some extent, a lower bound on the potential for dominator-path scheduling, as conservative methods were employed to choose dominator paths. There is no inherent requirement that dominator paths should include blocks only within a single loop construct. This is, in fact, a powerful feature of dominator-path scheduling, since it allows both motion of loop invariant code out of loops and motion of opera-

Program	Paths	%Improvement
dhrystone	15	8.0
diff3	102	7.4
grep	36	11.3
linpack	47	6.4
livermore	103	9.4
whetstone	11	7.6

Table 1: On-Path Improvement

tions into loops when they can be overlapped with existing code without requiring additional instruction cycles. The reason that multi-loop paths are not included in this study is that the "local" instruction scheduler used to schedule paths does not currently include any feature to prioritize the instructions of a path. If a single path were to include multiple nesting levels, we would want the scheduler to recognize that instructions added to blocks with higher nesting levels are more "costly" than those at lower nesting levels. This is not an issue for local scheduling and is thus not currently included in the scheduler. We are, however, making plans to incorporate such a feature for use in future dominator-path scheduling experiments.

4 Conclusions

This paper has introduced dominator-path scheduling, a method of global instruction scheduling based on an extended version of Reif and Tarjan's symbolic cover analysis. As dominator-path scheduling does not require semantic-preserving copies like those of other global scheduling techniques, it appears promising for architectures that have a limited amount of instruction-level parallelism.

References

[AN88] Alexander Aiken and Alexandru Nicolau. "A development environment for horizontal microcode". *IEEE Transactions on Software Engineering*, 14(5):584–594, May 1988.

[Bea91] S.J. Beaty. *Instruction Scheduling Using Genetic Algorithms*. PhD thesis, Mechanical Engineering Department, Colorado State University, Fort Collins, Colorado, 1991.

[BR91] D. Bernstein and M. Rodeh. "Global instruction scheduling for superscalar machines". In *Conference on Programming Language Design and Implementation*, pages 241–255, Toronto, June 1991. SIGPLAN '91.

[Ell86] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, Cambridge, MA, 1986. PhD thesis, Yale, 1984.

[FOW87] J. Ferrante, K.J. Ottenstein, and J.D. Warren. "The program dependence graph and its use in optimization". *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[RT81] J.H. Reif and R.E. Tarjan. "Symbolic program analysis in almost-linear time". *SIAM Journal of Computing*, 11(1):81–93, February 1981.

[Swe92] P.H. Sweany. *Inter-Block Code Motion without Copies*. PhD thesis, Computer Science Department, Colorado State University, Fort Collins, Colorado, 1992.