A Code Motion Technique for Scheduling Bottleneck Resources

Extended Abstract

Efraim Fogel, Immersia Ltd.[1]
James C. Dehnert, Silicon Graphics, Inc.[2]

Abstract: Most microprocessors have resources that may become bottlenecks for scheduling. If such a resource is explicitly referenced in the instruction set architecture, anti-dependences between producer-consumer instruction sequences in the data dependence graph (*DDG*) derived from the original program order may result in unnecessary and costly constraints on scheduling.

We describe a transformation of the data dependence graph which reorders these producer-consumer instruction sequences in the DDG, shortening the critical path, and therefore improving the potential schedule length, by replacing the original anti-dependence arcs by less constraining ones.

This technique has been implemented as part of a compiler for a C-like programming language for the GE11, and used to generate production microcode for a special-purpose VLIW SIMD graphics processor used in the geometry processing units of the Silicon Graphics IMPACT™ and RealityEngine™ graphics subsystems[3]. It produced improvement in most cases, ranging up to 2x speedups.

---

[1] Most of this work was done while the first author was with Silicon Graphics, Inc. His current address is Immersia, Tel-Aviv, Israel; telephone +972.3639.3483; FAX +972.3537.1264; email efi@immersia.com.

[2] Address 1600 Amphitheatre Pkwy., m/s 178, Mountain View, CA; telephone (650)933-4272; email dehnert@sgi.com.

[3] IMPACT™ and RealityEngine™ are trademarks of Silicon Graphics, Inc.

1 Introduction

1.1 Motivation

It is common for a microprocessor architecture to include unique resources, even when the architecture supports extensive instruction-level parallelism (*ILP*). When such resources are referenced explicitly in the instruction set, written by one instruction and read by a subsequent instruction, they can become a serious scheduling constraint. For example, consider the C code fragment:

```
int i,j,k,l,m,n; i = (i+j)*k; l = m*n;
```

On the MIPS architecture, which writes multiply/divide results to dedicated registers named hi/lo, straightforward compilation would produce the following code:

```
1)   add  rt = ri+rj
2)   mul  hi,lo = rt*rk  # implicit hi/lo results
3)   mflo ri = lo        # implicit lo operand
4)   mul  hi,lo = rm*rn  # implicit hi/lo results
5)   mflo rl = lo        # implicit lo operand
```

Normal data dependence analysis will identify true input dependences in the instruction sequences (1)->(2)->(3) and (4)->(5). In addition, since (3) reads `lo` and (4) writes it again, it will identify an anti-dependence (3)->(4), fully sequentializing the code. This is not, however, an optimal dependence graph. Since the second multiply doesn't depend on an earlier add, a better DDG on a machine with ILP has arcs (4)->(5)->(2)->(3) and (1)->(2). This is similar to the artificial scheduling constraints introduced by allocating registers before scheduling, but the use of a unique dedicated resource prevents the use of methods like later allocation or register renaming typically applied to that problem. We will call such unique dedicated resources *bottleneck resources.*

The Silicon Graphics GE11 graphics processor has several such bottleneck resources, making their effective scheduling both more difficult and more important. For instance,

memory loads and stores (to each of several banks of memory) all require staging the data through specific staging registers. That is, storing a datum requires first writing it to a staging register and then moving it to memory with a second instruction which provides the address.

1.2 The GE11 Compiler

The work described in this paper is implemented in a compiler produced for the Silicon Graphics GE11 graphics processor. As a custom-designed embedded processor, its architecture is significantly different from any other processor designed before or since at Silicon Graphics. Nevertheless, the compiler is intended to be very retargetable, based on tables describing the target architecture. It was derived from a compiler for an earlier Silicon Graphics microprocessor, and has now been retargeted to the GE11's successor graphics engine. The approach used is to provide a high-level substitute for assembly language, which allows close control of machine features, while relieving the user of responsibility for the complexities of code scheduling and register allocation. To this end, it implements a C-like language with built-in names for the special architectural registers, optional explicit allocation of variables to registers, and special control constructs for handling the SIMD execute-under-mask capabilities. It does not perform sophisticated global optimization. Effort is focused on a high-quality instruction scheduler similar to that in the Cydrome compiler [DeTo93], and a global register allocator.

The instruction scheduler works on an intermediate representation of the program which consists of a flowgraph of basic blocks (BBs), each containing a sequence of *operations*, i.e. an assembly-level operator with some number of operands and results. The operands and results are *Temporary Names* (*TNs*), i.e. pseudo-registers. In cases where a particular physical register must be used, a *dedicated TN* (*DTN*) associated with that register is used. DTN usage may result from explicit references in the source code, or from translating higher-level constructs to operation sequences which must use specific DTNs. These DTNs are the bottleneck resources with which we are concerned here.

Prior to scheduling, the compiler performs renaming of non-dedicated TNs and a variety of local optimizations like copy propagation and dead code removal. It then builds a data dependence graph (*DDG*, see [Towle76] and [KKPLW81]) for each BB. This is the structure manipulated by the algorithm described in this paper, and then processed by the instruction scheduler. Although there are cross-BB interactions, we will assume in this paper that a single basic block is being scheduled.

Bottleneck resources are a problem because they introduce anti-dependences into the DDG which force suboptimal ordering of references to them during scheduling. Those anti-dependence arcs in the DDG also make it easy to recognize potential problems, so our solution is implemented after construction of the DDG. Critical path analysis of the DDG also provides a means of evaluating the benefit of a transformation without full scheduling: if a transformation decreases the length of the critical path through the block, it is likely beneficial. We chose not to implement a solution in the scheduler proper, preferring to keep it independent of program transformations and therefore simpler.

1.3 Previous Work

We are unaware of any published approach to this specific problem. The related problem for non-dedicated registers has been widely approached by renaming live ranges prior to scheduling to remove the problematic anti-dependences, e.g. in [DeTo93].

2 The Inversion Transformations

2.1 Preparation

The data dependence graph in the GE11 compiler consists of a node for each operation in the BB, plus dummy START and STOP nodes. Each dependence between operations is represented by an arc between the corresponding nodes, decorated with the latency required to satisfy the dependence, and the kind of dependence. We will sometimes write an arc as op_1->op_2 where the *tail* of the arc, op_1, must precede the head of the arc, op_2, by

at least the latency decorating the arc. In addition to normal data dependences, other constraints, such as the requirement that an operation be executed before or after a call or other branch, are also represented by arcs in the DDG. Finally, the START node is made a predecessor of all nodes with no other predecessors, and the STOP node a successor of all nodes with no other successors.

References to bottleneck resource DTNs are identified, and divided into live ranges of the DTN, i.e. all operations which either define the value of the DTN or use the value later. Although this set normally contains one definition and one or more uses, the SIMD features of the GE11 processor sometimes result in multiple definitions for a single live range. We call this set of operations for a particular live range a *cluster*. An operation *op* and a DTN *tn* determine the associated cluster, which we denote *cluster(op,tn)*. Further, we denote the subsets of operations defining and using the DTN in a cluster *C* by *defSet(C,tn)* and *useSet(C,tn)* respectively, or, if *op* is one of the operations in *C*, by *defSet(op,tn)* and *useSet(op,tn)* respectively.

A cluster with definitions in predecessor BBs (i.e. the resource is live into the BB) must be scheduled first, and a cluster with references in successor BBs (i.e. the resource is live out of the BB) must be scheduled last. Clusters associated with volatile DTNs (e.g. output ports to the next processing stage in the GE11) are also excluded from reordering. The other clusters may be reordered by our algorithm.

Critical path analysis is performed on the DDG by a straightforward algorithm. The earliest start cycle (*estart*) for all nodes is initialized to zero. We repeatedly choose a node **N** for which all predecessors in the DDG have been processed (starting with the START node), and set the estart of each successor **M** to the maximum of estart(**N**)+latency(**N**,**M**) and the previous value of estart(**M**). Once we have calculated estart(STOP), we initialize the latest start cycle (*lstart*) of all nodes to estart(STOP). Finally, we repeatedly choose a node **N** for which all successors in the DDG have been processed (starting with the STOP node), and set the lstart of each predecessor **M** to the minimum of lstart(**N**)-latency(**M, N**) and the previous value of lstart(**M**). This calculation gives us a best-case estimate of the length of the BB's schedule (i.e.

estart(STOP)), and identifies critical path operations as those for which estart(**N**) = lstart(**N**).

In addition to the normal critical path analysis, we do a modified analysis to identify candidates for transformation. All output and anti-dependence arcs associated with bottleneck DTNs are identified, and those between clusters that might be reordered (i.e. not live-in or live-out clusters) are ignored. Based on this modified DDG, new estart and lstart functions are calculated, which we call *ecycle* and *lcycle*.

2.2 The *potential* Function

This modified DDG reflects scheduling constraints ignoring the artificial ones imposed by the DTN anti-dependences. Intuitively, we expect the best schedule to result from an ordering of the clusters which matches their order in this DDG. That is, if the *defSet* operations of $cluster_2$ have later ecycles than those of $cluster_1$, and the *useSet* operations of $cluster_2$ have later lcycles than those of $cluster_1$, then we would expect the best schedules to result from an anti-dependence arc from $cluster_1$ to $cluster_2$ rather than the opposite. Therefore, we define a *potential* function which reflects this objective, and our algorithm attempts to maximize its value on the DDG by reordering clusters.

Specifically, given an anti-dependence arc $op_1 \text{->} op_2$ associated with DTN *tn*, we define $potential(op_1 \text{->} op_2)$ as follows. Let *earlyDiff* be the maximum over all dop_1 in $defSet(op_1,tn)$ of $ecycle(op_2)\text{-}ecycle(dop_1)$. Similarly, let *lateDiff* be the maximum over all uop_2 in $useSet(op_2,tn)$ of $lcycle(uop_2)\text{-}lcycle(op_1)$. Then we define the potential of the arc by $potential(op_1 \text{->} op_2) = earlyDiff + lateDiff$.

Given the potential function on arcs, we define the potential function on the full DDG ***G*** as follows. Let *arcSet* be the set of all anti-dependence arcs associated with bottleneck DTNs with head and tail in different clusters. Then $potential(\boldsymbol{G})$ is the sum of $potential(arc)$ over all $arc \in arcSet$.

2.3 Local Inversion

The primitive transformation of our algorithm, which we call *local inversion*, inverts the order of two clusters in the DDG. More precisely, suppose we have a DTN *tn*, and an

anti-dependence arc op_1->op_2 associated with *tn* connecting operations op_1 in cluster C_1 and op_2 in C_2, where neither C_1 nor C_2 is required to be first or last in the BB. Then local inversion of the arc op_1->op_2 removes all anti-dependence arcs associated with *tn* between the clusters, and replaces them with arcs in the opposite direction, i.e. with anti-dependence arcs from ops in $useSet(C_2,tn)$ to ops in $defSet(C_1,tn)$. (As explained later, we ignore output dependence arcs between clusters at this point.)

In accordance with the intuition described above, we expect a local inversion to be beneficial if the new arcs have higher potential than the old arcs they replace, i.e. if $potential(\mathbf{G})$ is increased by the inversion.

2.4 Global Inversion

Unfortunately, the legality and benefit of a local inversion cannot always be evaluated in isolation. The main problem is that an individual local inversion may introduce a cycle in the DDG, because other dependences in the graph require the original ordering. Nor is it simply a matter of performing the local inversions in the right order. It is possible to have pairs of clusters for different DTNs which must be scheduled together (e.g. because they share an operation), where two such pairs may be inverted but inverting just the clusters for either of the DTNs yields a cyclic (and therefore unschedulable) DDG.

To solve this problem, we embed local inversion steps in a more comprehensive heuristic called *global inversion*. After inverting a candidate arc op_1->op_2, global inversion checks for introduced circularity by checking for a path from $defSet(C_1,tn)$ back to $useSet(C_2,tn)$. If such a path is found, it identifies the lowest-potential between-cluster anti-dependence arc on the path that is not already the result of the current transformation, and inverts it. This procedure is repeated until one of three things occurs:

- No circularity remains in the DDG, and the total change in potential of the local inversions is positive. In this case, the transformation is applied.
- No further arcs remain as candidates for inversion, but circularity has not been eliminated. In this case, the entire sequence of local inversions is abandoned.

- At some intermediate step, the total change in potential of the local inversions falls below a threshold. We currently set the threshold to 0, but it could be negative to allow trial sequences with unbeneficial intermediate steps. In this case too, the global inversion is abandoned.

2.5 Optimization

We can now describe the full optimization process in terms of the above components. It operates on an input DDG and a set of DTNs. A *preparation* phase removes output dependence arcs between the DTNs (because they are not useful at this stage and can be easily regenerated later), and adds new anti-dependence arcs between the DTNs, filling out the transitive closure of the initial set.

The *optimization* phase identifies the anti-dependence arcs which are candidates for inversion, performs the modified critical path analysis ignoring them which is described above, and calculates the potential of each of the candidate anti-dependence arcs. It then repeatedly attempts global inversion on the lowest-potential candidate arc using a priority queue, removing arcs attempted or removed by successful inversions, and adding new arcs created by successful inversions, until the queue is empty.

Finally, the *restoration* phase inserts output dependence arcs between the clusters (based on the new ordering), and removes the redundant anti-dependence arcs inserted during preparation.

3 Conclusions

The optimization technique described in this paper is applicable to most architectures, and is not target-specific except in the identification of the bottleneck resources to be treated. It will be most useful for architectures with frequent use of such resources, and for application areas where complex algebraic expressions provide extensive opportunity for instruction level parallelism through instruction scheduling. Media processors and their applications often match this profile quite well.

Our target application, geometry processing for a graphics subsystem, was an excellent match, particularly because bottleneck resources were used for all memory references. In

this environment, we achieved at least a twofold speedup in most cases, and sometimes much more, with a reasonable cost in compilation time.

References

[DeTo93] J.C. Dehnert and R.A. Towle, "*Compiling for the Cydra 5*," The Journal of Supercomputing 7(1/2), 1993, pp. 181-227.

[KKPLW81] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M.Wolfe, "*Dependence Graphs and Compiler Optimizations*," Proceedings of the 8th ACM Symposium on Principles of Programming Languages, 26-28 January 1981, pp. 207-218.

[Towle76] Ross A. Towle, "Control and Data Dependence for Program Transformations," Ph.D. Thesis, Dept. of Computer Science, Univ. of Illinois at Champaign-Urbana, March 1976.