A Code Motion Technique for Scheduling Bottleneck Resources

Efraim Fogel, Immersia Ltd., and James C. Dehnert, SGI, Inc.

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 identifies clusters of instructions that define and then use such **bottleneck resources***, and then reorders these producer-consumer instruction sequences in the DDG, replacing the original anti-dependence arcs by less constraining ones. This shortens critical paths, improving the achievable schedule length.*

This technique has been implemented as part of a compiler for a C-like programming language that was used to generate production microcode for the GE11, a special-purpose VLIW SIMD graphics processor used in the geometry processing units of the Silicon Graphics IMPACT™ and RealityEngine™ graphics subsystems[1]*. It produced improvement in most cases, ranging up to 2x speedups.*

Most of this work was done while the first author was at SGI, Inc. Current addresses are: Efraim Fogel, Immersia, Tel-Aviv, Israel; email efi@immersia.com; James C. Dehnert , m/s 178, SGI, 1600 Amphitheatre Pkwy., Mountain View, CA; dehnertj@acm.org.

[1] 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, such as special-purpose registers, even when the architecture supports extensive instruction-level parallelism (*ILP*). When such resources are referenced explicitly in the instruction set, defined by one instruction and used 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)** uses `lo` and **(4)** defines 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 **(1)->(2)** and **(4)->(5)->(2)->(3)**. This situation 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*.

1.2 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].

For instance, if a particular expression is calculated and used multiple times, rather than assigning it to a single pseudo-register everywhere it is referenced, each new calculation with its associated uses is assigned to a new pseudo-register. Because the pseudo-registers are different, the definitions are not connected by anti-dependence arcs in the DDG, allowing the distinct values to be scheduled in any order or even in parallel.

This produces an optimally unconstrained DDG for this typical case, but because of the uniqueness of bottleneck resources, it does not work for them. In particular, distinct values cannot coexist in a unique register, so all uses of one must be scheduled before the next definition. This means we must seek a way to allow optimal reordering while preventing the parallel scheduling that renaming would permit.

The remainder of this paper is organized as follows. In Section 2, we describe features of the GE11 graphics processor that motivated this work and relevant features of the compiler in which it was implemented, followed by a concrete example. In Section 3, we describe the transformation we implemented. Finally, in Section 4 we comment on our results and summarize.

2 Preliminaries

2.1 GE11 Graphics Processor

The Silicon Graphics GE11 graphics processor was designed as the geometry processing unit for a pair of high-performance graphics subsystems. The IMPACT™ mid-range graphics subsystem employs two GE11 processors, while the RealityEngine™ high-end graphics subsystem employs eight, working in parallel as a MIMD system.

Each GE11 is itself a 3-way SIMD processor, with three processing cores consisting of CPUs, register files, and local memories, executing a single instruction stream under an execution mask and sharing some memory and register resources. Its architecture is tailored to its graphics processing purpose, with special-purpose registers, buses, and memories, all explicitly controlled by a VLIW instruction set. As a result, the GE11 has several bottleneck resources, making their effective scheduling both more difficult and more important.

For instance, consider memory. The GE11 has three memory component types on chip, called **wramf**, **cramf**, and **eramf**. A value is read from memory in two steps, first sending an address to the memory, and then reading the result from a special staging register associated with the memory. For example, the code to read a value at the address in register **a1** from **wramf** and multiply it by another value in register **ry** might be as follows:

```
WRDF = wramf(a1, 0);
T1 = mul (ry, WRDF);
```

Here, the first statement sends the address to **wramf**, causing it to be loaded into load staging register **WRDF**, and the second statement references **WRDF** as an operand in a multiply instruction.

Similarly, stores to **eramf** and **cramf** use the staging register **ECDF**. Data is first copied to **ECDF**, and then a store instruction specifies the address to which it is stored. Thus, storing the contents of register **ry** to the **eramf** address in **a1** would be done as follows:

```
    ECDF = copy(ry);
    eramf(a1, 0) = ECDF;
```

Each of the memory staging registers is a bottleneck resource. The **cramf** and **eramf** memories are shared by all three core processing units; **wramf** and the staging registers are all replicated, with one copy per core. The full set is:

- **cramf**: loads staged through **CRDF**; stores through **ECDF**.
- **eramf**: loads staged through **CRDF**; stores through **ECDF**.
- **wramf**: loads staged through **CRDF** or **WRDF**; stores direct (not staged).

2.2 The GE11 Compiler

The work described in this paper is implemented in a compiler produced for the 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 been retargeted to the GE11's successor graphics engine.

The objective of the compiler was to replace hand-scheduled assembly code for the graphics microcode. The approach was to implement 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, we defined a language similar to C but much simpler, with special features allowing the programmer direct access to machine features. It has built-in names for the special architectural registers, e.g. **CRDF** for the load staging register; it allocates all variables to registers and allows the programmer to specify a particular register; and it has special control constructs for handling the SIMD execute-under-mask capabilities. Effort is focused on a high-quality instruction scheduler similar to that in the Cydrome compiler [DeTo93], and a global register allocator. It does not support direct memory allocation or a stack; nor does it perform traditional global optimizations. The result is a model requiring that program functionality be specified at a low level, but eliminating most of the tedium of instruction scheduling and register allocation, both of which are very complex on the GE11.

The instruction scheduler works on an intermediate representation of the program that consists of a flowgraph of extended basic blocks (BBs), single-entry, multiple-exit sequences of *operations*. An operation is an assembly-level operator with some number of operands and results, each of which is a *Temporary Name* (*TN*), i.e. a pseudo-register, or a literal value. 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 represent the bottleneck resources with which we are concerned.

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. The nodes of this graph are the operations, and the arcs represent ordering constraints between the operations connected. Data dependence arcs connect operations that both reference the same pseudo-register, or TN, and their significance differs depending on the kind of references. Input dependences constrain a use to follow the operation producing the value. Output dependences enforce the order of multiple definitions of the same TN. Anti-dependences require a later definition of a TN to follow a use of an earlier definition. 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 extended basic block is being scheduled.

Bottleneck resources are a problem because they introduce anti-dependences into the DDG that prevent an optimal ordering of definitions and uses of the bottleneck resource 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 the scheduler simpler by keeping it independent of program transformations.

2.3 Example

For illustration of the later discussion, we present an example here. Suppose we are compiling the statement pair:

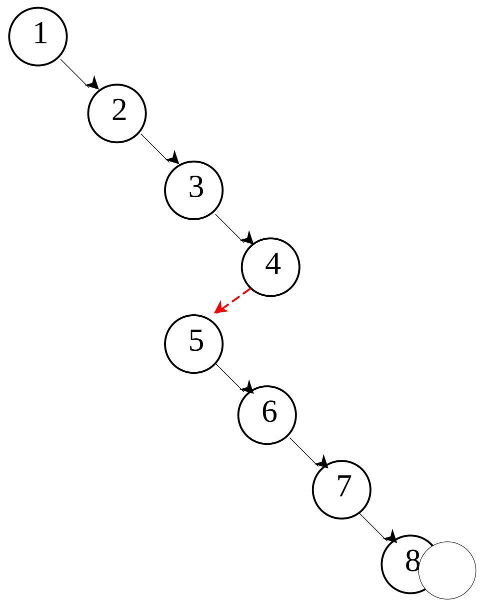
```
x4 = cramf((base+i1)*2) * 4;
y8 = ((wramf(ay) * 4) + i7) * 8;
```

Then the pseudo-assembly code might be as follows:

```
1) t1 = base + i1
2) a2 = t1 * 2
3) CRDF = cramf(a2,0)
4) x4 = CRDF * 4
5) CRDF = wramf(ay,0)
6) y4 = CRDF * 4
7) t7 = y4 + i7
8) y8 = t7 * 8
```

The DDG for this code fragment is given in Figure 1, with solid arrows for input dependence arcs, and dashed arrows for anti-dependence arcs. For reasons explained later, we ignore output dependences.

Figure 1: Initial Data Dependence Graph



If we assume for simplicity that all latencies are 1 cycle, then the critical path to node 8 is 7 cycles. As we shall see later, this can be cut to 3 cycles by reordering the uses of **CRDF**.

3 The Inversion Transformations

3.1 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 is represented by an arc between the nodes representing the operations involved, 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, arcs in the DDG also represent other constraints, such as the requirement that an operation be executed before or after a call or other branch. 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 that 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. Our algorithm may reorder the other clusters.

In our example, the clusters are pairs that define and then use the DTN for **CRDF**: {3,4} and {5,6}. All three clusters may be reordered.

Critical path analysis is performed on the DDG by a straightforward two-phase algorithm. The first phase calculates the earliest start cycle (*estart*), starting by initializing it to zero for all nodes. 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**), the second phase calculates the latest start cycle (*lstart*), starting by initializing it to estart(**STOP**) for all nodes. 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 anti-dependences between clusters that might be reordered (i.e. not volatile, 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*.

In our example, the critical path analysis yields the values given in the following table. Note that all of the nodes are on the critical path. In calculating ecycle/lcycle, we ignore the anti-dependence arcs given as dotted lines in Figure 1.

Table 1: Critical Path Analysis

op	estart	lstart	ecycle	lcycle
START	0	0	0	0
1	0	0	0	0
2	1	1	1	1
3	2	2	2	2
4	3	3	3	3
5	4	4	0	0
6	5	5	1	1
7	6	6	2	2
8	7	7	3	3
STOP	7	7	3	3

3.2 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 ecycle/lcycle 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* $op_1 \rightarrow op_2$ associated with DTN *tn*, we define *potential(arc)* as follows.

$$earlyDiff(arc) = cycle(op_2) - \max_{o \text{ in } defSet(op_1,tn)} \{ ecycle(o) \}$$

$$lateDiff(arc) = \min_{o \text{ in } useSet(op_2,tn)} \{ lcycle(o) \} - lcycle(op_1)$$

$$potential(arc) = earlyDiff(arc) + lateDiff(arc)$$

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:

$$potential(\boldsymbol{G}) = \sum_{arc \in arcSet} potential(arc)$$

In applying the potential function to our example, we find one anti-dependence arcs associated with the DTN for **CRDF**: **(4)->(5).** The results of the analysis are given in the following table:

Table 2 : *potential* Analysis

arc	earlyDiff	lateDiff	potential
(4)->(5)	-2	-2	-4

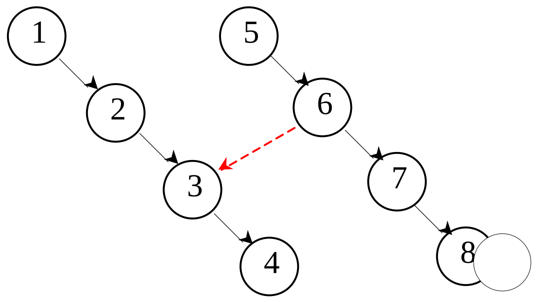
3.3 Local Inversion

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 $op_1 \rightarrow op_2$ 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 volatile or is required to be first or last in the BB. Then local inversion of the arc $op_1 \rightarrow op_2$ removes it, and replaces it 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 have no DTN 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(**G**)* is increased by the inversion.

In our example, suppose we choose to invert the anti-dependence arc, **(4)->(5)**. That involves removing it, and adding a new anti-dependence arc **(6)->(3)**. Since the anti-dependence arcs are not considered in the ecycle/lcycle calculations, they are not redone, and the potential calculation for our new arc yields *earlyDiff = 2, lateDiff =2,* and *potential = 4*. We expect the increase in potential to produce an improvement, and indeed, if we recalculate estart/lstart, we now find that the estart of the **STOP** node is now 4 instead of 8. Since there are critical paths of length 4 containing only input dependence arcs, we can do no better. The modified data dependence graph is shown in Figure 2.

Figure 2: Inverted Data Dependence Graph



3.4 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 some 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*(op_1,*tn*) back to *useSet*(op_2,*tn*). If such a path is found, it identifies the lowest-potential anti-dependence arc on the path from *arcSet*, that is not already the result of the current global inversion, 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 intermediate steps that are not beneficial. In this case too, the global inversion is abandoned.

3.5 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, completing the transitive closure of the initial set.

The *optimization* phase identifies the anti-dependence arcs which are candidates for inversion, calculates ecycle/lcycle values (i.e. the modified critical path analysis that ignores 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.

4 Conclusions

4.1 Summary

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. Special-purpose media processors and their applications often match this profile quite well.

Our target application, microcode implementing geometry processing for a graphics subsystem, was an excellent match, particularly because bottleneck resources were used for all memory references. We did not do a careful performance study, but the overall performance improvement was approximately 10-20%, and we saw doubling of speed in individual functions, with a reasonable cost in compilation time. Since this optimization was implemented after much of the microcode had been written, many of the critical instances had already been hand-optimized, so these numbers understate the likely benefit if the optimization were present from the beginning.

4.2 Ideas for Future Work

There are several possible improvements to our heuristics that might be tried.

In local inversion, we replace only the arc being inverted. If there are multiple arcs between the clusters involved, that will necessarily leave a cycle to be removed by global inversion. It is probably more efficient to identify and invert all of the arcs between the two clusters at once.

We currently identify the local inversions that make up a global inversion one at a time. This is very simple to implement, but it does lead to dead ends. It might be more effective to first identify the partial order on clusters imposed by input dependences, and then attempt only reordering consistent with that partial order.

Acknowledgments

Ross Towle produced the original compiler front end. The SGI IMPACT™ and RealityEngine™ microcode development teams were tremendously helpful in identifying opportunities and problems.

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.

[Tarjan81] Robert Endre Tarjan, “Fast Algorithms for Solving Path Problems,” JACM 28(3), July 1981, pp. 591-642.

[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.