CRAIG: A Practical Framework for Combining Instruction Scheduling and Register Assignment

Thomas S. Brasier
Microware Systems Corporation
1900 NW 114th St.
Des Moines IA 50325
tomb@microware.com

Philip H. Sweany
Steve Carr
Computer Science Department
Michigan Technological University
Houghton MI 49931–1295
{*sweany,carr*}*@mtu.edu*

Steven J. Beaty
Cray Computer Corporation
1110 Bayfield Drive
Colorado Springs CO 80906
beaty@craycos.com

Abstract

In compilers for machines with instruction-level parallelism, the phases of register assignment and instruction scheduling can be antagonistic. Whichever phase is executed first can have negative effects on the other's performance. This paper describes a framework, called CRAIG (Combining Register Assignment Interference Graphs), that combines register assignment and instruction scheduling to alleviate the phase-ordering problem. CRAIG utilizes information gained from instruction scheduling before register assignment as an upper bound on the freedom needed by the instruction scheduler to attain its "best" schedule. CRAIG then allows heuristics to choose how close to the "best" schedule one can get before the cost of additional register pressure is too high. Within the context of this framework, the paper evaluates an instance of CRAIG called CRAIG_0.

1 Introduction

Recent work in architecture and compilation has identified and exploited significant amounts of Instruction-Level Parallelism (ILP)[1, 4]. To take best advantage of the ILP available in modern processors, compilers need to minimize delays due to memory latency using register assignment and instruction scheduling. Register assignment involves maximizing the scalar program values maintained in registers. Instruction scheduling involves hiding the latency to memory with useful instructions that can legally operate in parallel with a memory reference, as well as re-ordering operations to take advantage of available ILP.

Unfortunately, instruction scheduling and register assignment are antagonistic processes. When register assignment is done before instruction scheduling, unnecessary dependences are added. Spilling is minimized, but the potential for ILP is lowered and the execution time of the program increases. When instruction scheduling is carried out first, an efficient schedule is generated; however, the code motion that occurs generally increases register interference. This, in turn, can lead to a register interference graph that is not colorable with the number of available registers. If a graph is uncolorable, spill code will have to be added, and additional memory delays may be incurred. This paper describes a framework, called CRAIG (Combining Register Assignment Interference Graphs) that allows the benefits of instruction scheduling while taking into account the cost of additional register pressure. CRAIG combines the register interference graphs created before and after instruction scheduling to obtain both lower register pressure and increased flexibility in instruction scheduling.

2 Background

2.1 Graph Coloring Register Assignment

Chaitin [5, 6] is credited with applying the graph coloring paradigm to the register assignment problem. All scalar *live ranges* (sections of code over which a value has been defined and used but not redefined) are candidates for registers and are analyzed to create an *interference graph*. This graph is made up of nodes, each of which represents a program value, and arcs between them, representing interference between the values. Two values are said to interfere if both values are live at any execution point in the program.

In the graph coloring analogy, nodes represent scalars and colors represent registers. In general, the number of colors given to color the graph is the number of registers available on the machine. It is important to note that it is not always possible to color a graph with a limited number of colors. The problem of finding an optimal coloring for a graph has been proven to be NP-complete[9] and therefore heuristics must be applied to the process of coloring.

2.2 Instruction Scheduling

Instruction scheduling is the process of taking the code generated by the code selection process of the compiler and re-ordering operations such that a more time-efficient schedule is produced. Some operations may be limited in their ability to be re-ordered. This is usually due to data dependences. As the name implies, instruction scheduling attempts to exploit parallelism within a function's control flow graph by re-ordering instructions. In general, this optimization problem is NP-complete [7]. However, in practice, heuristics achieve good results. Landskov, et al. [11] give a good survey of instruction scheduling algorithms. Instruction scheduling typically uses a data structure called a data dependence DAG (DDD). 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**. Since these DDD edges represent constraints to instruction re-ordering, we wish to minimize such edges while ensuring the original program semantics.

2.3 Register Assignment Before Instruction Scheduling

When register assignment is done before instruction scheduling (**early** register assignment), graph coloring register assignment generally assigns the program's values to a near minimal number of registers; however this process can cause *anti-dependences* [2] to

be added. These anti-dependences arise due to the re-definition of registers that occurs when multiple values are mapped to the same physical register.

Consider the code in Figure 1. For this example, we shall as-

```
1)   t1   <-  A[i]
2)   sum  <-  sum + t1
3)   t2   <-  B[i]
4)   prod <-  prod * t2
```

Figure 1: Sample code

sume a machine that can initiate two operations per cycle. We further assume that the values of `sum, prod` and `i` are already assigned to registers `r0` through `r2` respectively. Figure 2 represents a typical early register assignment and the resulting instruction schedule.

r3 <- A[r2]	
r0 <- r0 + r3	r3 <- B[r2]
	r1 <- r1 * r3

Figure 2: Figure 1's code when register assignment is done **before** instruction scheduling

Note that two instructions side by side mean they can be executed simultaneously, thus the schedule in Figure 2 takes 3 steps to execute and uses 4 registers. Also, observe that when register assignment is done, an anti-dependence is generated between statements 2 and 3 due to the fact that physical register `r3` is reused. This anti-dependence prevents statement 3 from executing before statement 2.

2.4 Register Assignment After Instruction Scheduling

Conversely, when register assignment is done **after** instruction scheduling (**late** register assignment), the scheduler, uninhibited by anti-dependences, can generate an efficient schedule, but often increases the amount of interference between the programs values. If instruction scheduling were to occur first on the previous code, the code in Figure 3 could be generated.

r3 <- A[r2]	r4 <- B[r2]
r0 <- r0 + r3	r1 <- r1 * r4

Figure 3: Figure 1's code if register assignment is done **after** instruction scheduling

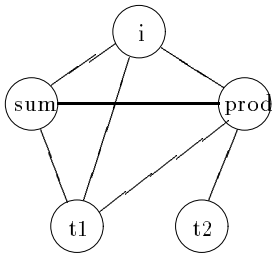
Note that the schedule in 3 requires only 2 steps, but uses 5 registers. Obviously this is an improved schedule. If there are only 4 registers available, however, spill code would be needed and the resulting schedule would be longer, due to delays caused by additional memory accesses.

2.5 Effect of Phase-Ordering on Interference Graphs

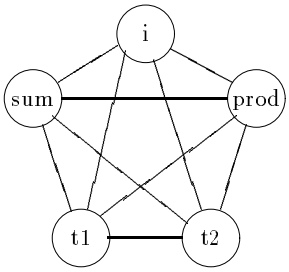
Using **late** register assignment, a schedule is generated with operations that have values rather than registers as their operands. After the schedule is generated, live variable analysis is done on this new schedule. Since several operations take place at the same time, each of these operation's operands must exist in separate registers. This generates more register interference, which means the interference graph will generally have more edges. A dense interference graph gives a register assignment with fewer anti-dependences and an increase in parallelism available to the instruction scheduler. Conversely, the dense graph makes it less likely that the graph will be colorable.

The interference graph from **early** register assignment generally has fewer edges than the graph from **late** register assignment. This is because the live-range analysis is done on the original sequential code where live ranges are likely to be shorter. An interference graph with fewer edges increases the probability of a colorable graph because more values can be assigned to the same register. However, the more sparse graph restricts the parallelism available to the instruction scheduler by introducing anti-dependences that arise from multiple values being mapped to the same register. Note that the anti-dependences can be eliminated by adding an edge between two values that are mapped to the same register. It is this observation that is the basis the framework described in this paper.

Examples of register interference graphs for the **early** and **late** register assignment of Figures 2 and 3 are depicted in Figure 4.



Register Interference Graph for Figure 2



Register Interference Graph for Figure 3

Figure 4: Register Interference Graphs Comparison

3 Related Work

There are obvious pros and cons to doing register assignment early or late. Since these two compiler phases are antagonistic, it has been suggested that the two phases should be merged. A basic technique

is to have a pool of registers being managed by the scheduler. In this way, the scheduler can take into account the number of available registers when it is scheduling. Several researchers have tried more involved techniques.

Goodman and Hsu [10] compared two different methods of integration against both early register assignment and late register assignment. In one method, which they call integrated prepass scheduling, they perform late register assignment while restricting the scheduler to use a fixed number of registers. Their other method manipulates the scheduler's DDD so that the "width" of the DAG is no greater than the number of registers available. Using the altered DDD, the method performs early register assignment. They found both techniques outperformed strictly early or strictly late register assignment.

Bradlee [3] discusses a method of mixing the two phases in which initial passes of an instruction scheduler get estimates of the schedule cost given a certain number of registers. The scheduler is run locally with a very limited number of registers and then again with the maximum number of registers in the machine. These values are then used to allocate a certain number of registers for each basic block.

Freudenberger [8] describes how register assignment is integrated into trace scheduling in the MultiFlow compilers. The scheduler drives the register assignment process to place the heavily used values (i.e. the values in the heavily-used traces) in registers. Since trace scheduling starts scheduling on the crucial traces first, the scheduler, which uses a pool of registers, takes as many registers from the pool as it needs. Since traces have multiple entry and exit points, information is stored about which registers contain which values at each entry and exit point. When other, less crucial, traces "hook up" to this trace, this information is used to minimize the amount of code for data movement that is needed.

Norris and Pollock [12] perform early register assignment, but add edges to the interference graph to estimate the re-ordering effect of instruction scheduling. Their basic goals are to construct a plan which does not require alteration of the scheduling phase of compilation, and to require only a single pass of instruction scheduling. They achieve these goals by building the interference graph from the Data Dependence DAG (DDD) rather than from either a linear listing of intermediate code (as "traditional" early assignment does) or the scheduled code (as late assignment does.) Since the DDD represents more parallelism than either the initial linear code or the final schedule, this could lead to interference graphs which would be hard to color. Norris and Pollock evaluate several heuristic techniques to limit the DDD parallelism (and thus the number of registers needed) while hopefully retaining a good schedule. Their experiments show a significant improvement over strictly early register assignment for livermore loops. We feel CRAIG is at least as easy to implement and like Norris and Pollock's work does not require any alteration to the instruction scheduler. CRAIG will sometimes require two passes of instruction scheduling, but, unlike Norris and Pollock we do not see this as a significant problem.

Pinter's work [14] realizes that the register interference graph for early register assignment has fewer edges, thus allowing extraneous anti-dependences to occur. This, in turn, leads to a schedule that is too conservative. To avoid this, her algorithm creates a parallelizable interference graph. To generate such a graph, the schedule graph (i.e. similar to a DDD except that nodes consist of only the destination for each operation) is analyzed and the transitive closure of all the directed edges are placed into a graph as undirected edges. Any machine dependences (resource conflicts) are then added to this graph. As an example, if a machine has only one divide unit, and two operations require the division unit, an edge is placed between the nodes corresponding to these operations. The graph's complement is then constructed and referred to as the false dependence graph. The union of the register interference graph and a false dependence graph is created. This new graph is the parallel interference graph, and represents all the true interference that exists between values. It can be colored to give a register assignment which does not retard any available ILP. Since it is likely that this graph is not colorable with the available registers, Pinter carefully chooses which edges to remove in order to avoid creating anti-dependences which might retard the final schedule.

By generating a parallelizable interference graph, Pinter is approaching the problem of incurred false dependences by removing all of them and then adding them back only as needed. Scheduling is done after register assignment, but only after the register assignment process has effectively "massaged" the information in a way that allows the scheduler more freedom to re-order code. It is this work that inspired the CRAIG algorithm presented in this paper. However, we are concerned that Pinter's method considers all false dependences. This could lead to graphs of unreasonable size. We prefer to consider only those anti-dependences which limit the instruction scheduler. Also, we feel that Pinter's combining edges from different types of graphs leads to unnecessary complication. CRAIG limits itself to combining edges in one type of graph, namely the register interference graph, and thus, we believe, is easier to implement.

4 CRAIG

In this section, we describe our framework for combining instruction scheduling and register assignment called CRAIG (Combining Register Assignment Interference Graphs). The basis of CRAIG is to exploit the knowledge gained from an initial pass of the scheduler before it has additional constraints placed on the code due to register assignment. The information gained from scheduling with **late** register assignment is an upper bound on the "best" schedule the scheduler can produce if the cost of additional register pressure is not prohibitive.

CRAIG mediates the "tug-of-war" between register assignment and instruction scheduling by providing a mechanism to decrease anti-dependences (thus increasing scheduling freedom) even to the extent of adding spill code. CRAIG incorporates this "mediation" as a schedule cost considering both schedule efficiency and register pressure. This schedule cost is a heuristic designed to meet the goals of the code generator. If the initial schedule cost is too high, CRAIG goes back to the original linear code and attempts **early** register assignment. The intuition is that the original code will have a less busy interference graph and will therefore have a lower cost due to register pressure. If this schedule cost is still too high, CRAIG accepts this schedule based upon the assumption that it is the best that we can do under the circumstances. If, however, the schedule cost is not too high it is likely that anti-dependences have been added, and thus, the schedule can be improved. CRAIG will attempt to reclaim some of this lost efficiency by removing as many of these anti-dependences as possible, up to the point where the schedule cost is too high. By adding edges found exclusively in the late register assignment interference graph, we are creating interference between those values which the scheduler forced to be in different registers. If they are mapped to the same register in the early register assignment interference graph, then we have identified and removed an anti-dependence that potentially inhibits a more efficient schedule. Figure 5 gives an overview of CRAIG.

Possibly the most attractive feature of CRAIG is its robust method to choose among early assignment, late assignment or something in between. By trying late register assignment first, small compilation units would generally have efficient schedules since there are no anti-dependences from register assignment to im-

Attempt late register assignment
If the schedule cost is acceptable then exit
Attempt early register assignment
RIG = *early register interference graph*
While schedule cost is not too high do
    Update RIG *by adding edge(s) from the late interference graph*
    Re-color and evaluate code
EndWhile
Output code based on RIG

Figure 5: Combining Register Assignment Interference Graphs

pede progress. However, for those times when additional register pressure is too costly, CRAIG will choose to attempt early register assignment. In the cases where this is too conservative, the generation of a less restrictive register assignment attempts to reclaim the forfeited loss in parallelism.

The compilation cost of CRAIG is low. The scheduler need only be run a maximum of two times: once to create the late interference graph and once to generate the final code. In addition, determining the schedule cost should be efficient. Finally, experimentation with an instance of CRAIG gives evidence that it often requires fewer passes of register assignment than strictly late register assignment. Thus, we expect CRAIG to require slightly more compilation time than early register assignment and require about the same time as late register assignment.

4.1 An Instance of CRAIG

In this section, we present CRAIG_0, an initial instance of CRAIG. CRAIG_0 first attempts to generate the best schedule it can without generating spill code. Therefore, a schedule cost is determined to be "too high" when spill code is inserted. When adding edges back from the late register assignment interference graph to the early register assignment graph, we add those edges between values which are mapped to the same register (i.e. colored the same color) but whose values interfered in the first graph. By using this heuristic, we are attempting to exploit the ILP that the scheduler found. Additionally, a heuristic choice must be used to determine which edges from among those mapped to the same register can remove the anti-dependences that most negatively effect the final schedule. Currently, we pick these edges arbitrarily. Possible future heuristics may be based on :

- a function of the priority used during the scheduling of the operation containing a value's definition while instruction scheduling was done late,
- a function that recognizes if the values used are on the critical path through the DDD,
- a function based on the live range of the values through the DDD, or
- a function of the number of variables that were live at the time the definition of the value is scheduled.

4.2 A CRAIG_0 Example

This section shows an example of the execution of CRAIG_0 on the following code.

```
x_pos = x_pos + (x_vel * t);
y_pos = y_pos + (y_vel * t);
```

The target machine is an ILP architecture with 2 identical functional units and 4 shared machine registers set aside to hold scalar values. Assume also, that `ld` and `st` instructions take two cycles to execute and all others take one cycle. Additionally, the variables `x_pos` and `y_pos` are used globally and therefore need to be stored out to memory once they are computed.

CRAIG_0 attempts to perform **late** register assignment. Figure 6(a) shows the schedule that is created. Each of the operations references pseudo-registers since register assignment has not yet mapped them to machine registers. When the live ranges of these pseudo-registers are analyzed, the register interference graph in Figure 7(a) is created. Note that the graph is not colorable with the 4 machine registers given. Thus, CRAIG_0 saves this interference graph and attempts to do **early** register assignment.

The **early** register interference graph for the example code, complete with its 3-coloring, can be found in Figure 7(b). Although the graph was colored with fewer colors, several anti-dependences were created. As a result, the less efficient schedule depicted in Figure 6(b) would be created.

Since spill code was not required, CRAIG_0 tries to combine the **early** register interference graph with the **late** interference graph. By adding edges from the **late** interference graph (Figure 7(a)), to the **early** interference graph (Figure 7(b)), the algorithm attempts to exploit the inherent parallelism. Figure 7(c) shows the resulting combined register interference graph. The combined interference graph will be 4-colorable since CRAIG_0 will stop adding edges before spill code is required.

The resulting schedule is shown in Figure 6(c). It requires 8 cycles to complete, however, there is no spill code, and thus, no additional memory delays will be incurred. For the sake of comparison, Figure 6(b) shows the schedule using the **early** register interference graph (Figure 7(b)). This schedule takes 12 cycles to complete.

As shown in the example, when register assignment is done **late**, more ILP can be exploited, yet the interference graph can be hard to color and spill code can be generated. When register assignment is done **early**, spill code is less likely to be required, but at the cost of losing parallelism. CRAIG_0 attempts to get the best of both early and late register assignment by trying to exploit the parallelism, but only up to the point where spill code would be required. Although CRAIG_0 cannot guarantee that **all** code will be scheduled without spill code, it does ensure that any code which does not require spilling with early register assignment, will not require spill code.

5 Experimental Evaluation of CRAIG_0

To evaluate CRAIG_0 we have implemented it in Rocket, a retargetable C compiler for ILP architectures[15]. Although Rocket is easily targeted to either superscalar or long-instruction-word (LIW) machines, we chose an LIW target for the experiments described here because that allows us to statically measure execution cycles necessary for a program, based upon profile information. The LIW model used is a load-store architecture which allows up to four operations to be initiated in each instruction. Integer computation is assumed to require 1 cycle to complete, while pipelined floating point operations require two cycles to complete. Loads and stores all require three cycles.

Given the chosen machine model, the total execution time of the program can be statically computed with the following formula

1	pr1 ← ld x_vel	pr2 ← ld t
2	pr6 ← ld y_vel	pr4 ← ld x_pos
3	pr3 ← pr1 * pr2	pr8 ← ld y_pos
4	pr7 ← pr6 * pr2	pr5 ← pr3 + pr4
5	pr9 ← pr8 * pr7	x_pos ← st pr5
6	y_pos ← st pr9	nop
7	nop	nop

(a) Late Schedule

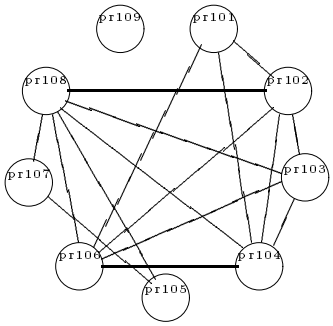
1	r1 ← ld x_vel	r0 ← ld t
2	nop	nop
3	r2 ← r1 * r0	r1 ← ld x_pos
4	nop	nop
5	r1 ← r2 + r1	nop
6	x_pos ← st r1	r1 ← ld y_vel
7	nop	nop
8	r1 ← r1 * r0	r0 ← ld y_pos
9	nop	nop
10	r0 ← r0 * r1	nop
11	y_pos ← st r0	nop
12	nop	nop

(b) Early Schedule

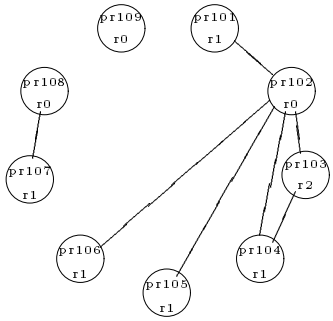
1	r2 ← ld x_vel	r0 ← ld t
2	r3 ← ld x_pos	r1 ← ld y_vel
3	r2 ← r2 * r0	nop
4	r3 ← r2 + r3	r2 ← ld y_pos
5	r1 ← r1 * r0	x_pos ← st r3
6	r0 ← r1 * r2	nop
7	y_pos ← st r0	nop
8	nop	nop

(c) CRAIG Schedule

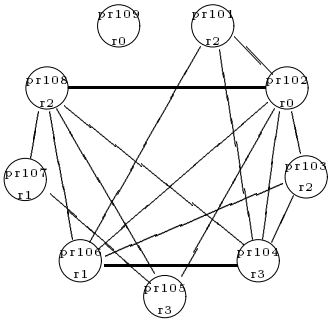
Figure 6: Early, Late and CRAIG$_0$ Schedules



(a) Late Graph



(b) Early Graph



(c) CRAIG Graph

Figure 7: Early, Late and CRAIG$_0$ Register Interference Graphs

:

$$\sum_{b \in basic\ blocks\ in\ program} length(b) \times frequency(b) \qquad (1)$$

where $length(b)$ is the number of instructions in block *b* and $frequency(b)$ is the number of times block *b* is executed. Profile data is only used for the purpose of computing schedule execution times. That is, the information is not used by the compiler to make instruction scheduling decisions. The URM machine simulator[13] was used to actually simulate execution and to help generate profile information for the test code.

To test CRAIG$_0$, we simply alter the number of available registers. By controlling the number of available registers, attempts at late register assignment can be forced to spill while attempts with the same machine description will allow early register assignment to have leftover registers, and thus, a combined register assignment interference graph will be generated.

5.1 Preliminary Results

This evaluation of CRAIG$_0$ is based upon three C programs whose results are representative of the trends we have seen. The programs are:

8q (3 functions) - The standard recursive method for finding a placement of 8 queens on a chess board such that no queen can attack another.

gauss (9 functions) - 20 × 20 Gaussian elimination.

livermore (3 functions) - Livermore loops 1 through 14 which represent standard loop kernels in scientific code.

Table 1 compares early and late register assignments when enough registers are permitted to obviate the need for spill code. The table indicates how many register were used in each of early and late register assignment, the number of execution cycles required with each technique and the % improvement gained by performing late register assignment. The first thing to notice is that the degree to which late register assignment allows a better schedule varies considerably. This is not unexpected. There are two factors which might limit the gain expected by late register assignment; first, it may well be that the additional freedom allowed late assignment does not lead to better schedules. However, it is often the case that the improved schedules of late assignment are counteracted by the additional register save-restore code required for the extra registers used in late assignment. In fact, when the save-restore code is not considered, the improvements due to late register assignment for *8q*, *gauss*, and *livermore* are 4.6%, 18.6%, and 5.2%. Still, for two of the three programs this doesn't seem to leave CRAIG$_0$ with much window for opportunity.

Table 2 compares execution cycles needed with each of early, late, and combined interference graphs. Each row of the table represents the cycles required with a given number of available registers. The available registers allowed ranged from those required by early assignment to one less than the number required to do late assignment without spilling. The entries marked *f* indicate that register assignment failed to converge in 10 iterations. Whenever late register assignment fails to converge within 10 iterations, Rocket resorts to early register assignment, based upon the premise that additional spill code of late assignment will degrade performance more than a tighter schedule could possibly compensate for. Thus, we could replace the *f* entries of the table with the cycles associated with early register assignment for that program.

Comparing the cycles required using the combined graph to those required with early assignment, we can see that the combined

	Registers Required				Execution Cycles		
	Early		Late		Early	Late	Improved
Benchmark	Ints	Floats	Ints	Floats			
8q	6	0	7	0	350933	346050	1.4
gauss	8	3	11	4	99837	82634	17.2
livermore	13	25	15	32	1390943	1319122	5.2

Table 1: Early vs. Late Register Assignment — No Spilling

Benchmark / Register Restriction	$CRAIG_0$ Total Execution time (cycles)	Early Total Execution time (cycles)	Early % improvement	Late Total execution time (cycles)	Late % improvement
8q					
6 integers	350933	350933	0.0	553035	36.5
gauss					
8 integers	89802	99837	10.1	f	
9 integers	83692		16.2	f	
10 integers	82374		17.5	88009	6.4
livermore					
13 integers	1319606	1390943	5.1	1319875	0.0
14 integers	1319364		5.1	1319790	0.0
25 floats	1320201		5.1	1319705	0.0
26 floats	1320119		5.1	1319620	0.0
27 floats	1327243		4.6	1319705	-0.1
28 floats	1326763		4.6	1319620	-0.1
29 floats	1326523		4.6	1319537	-0.1
30 floats	1326203		4.7	1319454	-0.1
31 floats	1325603		4.7	1319371	0.0

Table 2: $CRAIG_0$ Execution Time Results

graph technique, $CRAIG_0$, seems to be able to improve on early assignment about as much as late assignment does with unlimited registers. For *8q*, in which late assignment provided little improvement over early, $CRAIG_0$ provided little difference as well. For *livermore*, where late's improvement (with "unlimited" registers) over early is modest, $CRAIG_0$ achieves nearly the same improvement, with substantially fewer registers. And for *gauss*, in which late assignment allowed a significantly better schedule than early, $CRAIG_0$ was able to obtain a little more than half that improvement with no additional registers required over those needed for early assignment. By allowing $CRAIG_0$ 10 registers, (one less than the number used by late assignment with unlimited registers), we actually obtained a better execution time than late register assignment with 11 registers. We attribute the difference to the one less register to be saved and restored.

Looking at the cycles required when late register assignment is forced to spill shows several interesting effects. First, while $CRAIG_0$ gained nothing over early assignment for *8q*, it shows a 36% improvement over late assignment when late assignment is allowed only 1 register less than a number sufficient to require no spilling at all. Clearly there are times when spilling dramatically increases the execution time well beyond any scheduling gains obtained by late register assignment. In contrast, for *livermore* with 7 fewer registers than needed to ensure non-spilling, the cost of spilling was minimal (much less than 1%.) This suggests that, at least for some programs, substantial spilling costs virtually nothing. In this case, it was a program which didn't show significant improvement due to late register assignment even without spilling. For *gauss*, the spill costs (with 10 registers) were not enough to overshadow the benefits of scheduling without additional anti-dependences, but $CRAIG_0$ was able to show a marked (6.4%) improvement over late register assignment spilling.

Summarizing our results, $CRAIG_0$ shows promise of being as good or better than a policy of either strictly early or strictly late register assignment. The *livermore* results certainly suggest the need to allow at least the possibility of adding interference edges which would require spilling but allow more parallelism. Of course the CRAIG framework allows such possibilities, even if $CRAIG_0$ does not.

6 Conclusions and Future Work

We have presented a framework for combining register assignment and instruction scheduling called CRAIG. CRAIG adds interference graph edges from a graph constructed after instruction scheduling to a complementary interference graph constructed before instruction scheduling. CRAIG is robust enough to allow for growth and through its combined approach, it manages to exploit the advantages of both late and early register assignment. In the future, as more aggressive global scheduling techniques and compiler optimizations (e.g. scalar replacement, loop unrolling) force late register assignment to spill, CRAIG will still be able to use the information from late register assignment to improve schedules.

Within the context of CRAIG, we have implemented and evaluated an initial instance of the framework called $CRAIG_0$. $CRAIG_0$ attempts to generate schedules without spill code while still allowing for as much code motion as possible. $CRAIG_0$ shows promise of outperforming either strictly early or strictly late register assignment.

In the future, we will implement and evaluate the heuristics for choosing edges to copy from the late register assignment interference graph mentioned in Section 4.1. We will also consider heuristics to guide when and how CRAIG should insert additional interference edges when such edges would result in spill code.

Acknowledgments

We would like to thank Preston Briggs for his help in reviewing initial iterations of this framework. We would also like to thank the

National Science Foundation for helping to fund this work through Grant CCR-9308348.

References

[1] AUSTIN, T. M., AND SOHI, G. S. Dynamic dependency analysis of ordinary programs. *1992 IEEE 19th Annual International Symposium on Computer Architecture* (1992), 342–351.

[2] BANERJEE, U., SHEN, S., KUCK, D., AND TOWLE, R. Time and parallel processor bounds for FORTRAN-like loops. *IEEE Transactions on Computers C-28*, 9 (Sep 1979), 660–670.

[3] BRADLEE, D., EGGERS, S., AND HENRY, R. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, California, April 1991).

[4] BUTLER, M., YEH, T.-Y., PRATT, Y., ALSUP, M., SCALES, H., AND SHEBANOW, M. Single instruction stream parallelism is greater than two. *IEEE 18th Annual Symposium on Computer Architecture* (1991), 276–286.

[5] CHAITIN, G. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction* (June 1982), pp. 201–207.

[6] CHAITIN, G., AUSLANDER, M., CHANDRA, A., COCKE, J., HOPKINS, M., AND MARKSTEIN, P. Register allocation via coloring. *Computer Languages 6* (1981).

[7] DEWITT, D. *A Machine-Independent Approach to the Production of Optimal Horizontal Microcode*. PhD thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, MI, 1976.

[8] FREUDENBERGER, S. M., AND RUTTENBERG, J. C. Phase ordering of register allocation and instruction scheduling. In *Code Generation - Concepts, Tools, Techniques: Proceedings of the International Workshop on Code Generation* (London, May 1992), R. Giegerich and S. L. Graham, Eds., Springer-Verlag, pp. 146–172.

[9] GAREY, M., AND JOHNSON, D. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, CA, 1979.

[10] GOODMAN, J., AND HSU, W. Code scheduling and register allocation in large basic blocks. In *Proceedings of the ACM SIGPLAN 88 Conference on Program Language Design and Implementation* (1988).

[11] LANDSKOV, D., DAVIDSON, S., SHRIVER, B., AND MALLETT, P. Local microcode compaction techniques. *ACM Computing Surveys 12*, 3 (September 1980), 261–294.

[12] NORRIS, C., AND POLLOCK, L. A scheduler-sensitive global register allocator. In *Proceedings of Supercomputing '93* (Portland, OR, Nov. 1993).

[13] PETERMAN, C. L. An analysis of instruction-level parallelism in several common benchmarks. Master's thesis, Michigan Technological University, 1993.

[14] PINTER, S. S. Register allocation with instruction scheduling: A new approach. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation* (1993), 248–257.

[15] SWEANY, P. H., AND BEATY, S. J. Overview of the ROCKET retargetable C compiler. Tech. Rep. CS-94-01, Department of Computer Science, Michigan Technological University, Houghton, January 1994.