Spill-Free Parallel Scheduling of Precedence Graphs

Balas Natarajan, Computer Systems Laboratory
Michael Schlansker, Compiler and Architecture Research
HPL-95-131
November, 1995

balas@hpl.hp.com
schlansk@hpl.hp.com

VLIW scheduling,
register allocation

This paper concerns the problem of spill-free scheduling of acyclic precedence graphs on a processor with multiple functional units and a limited number of registers. The problem of minimizing the schedule length is well known to be computationally intractable. We present a heuristic for the problem, a general divide-and-conquer paradigm that converts any insensitive scheduling algorithm—one that is insensitive to register constraints—to one that respects register constraints. We estimate the goodness of the heuristic by relating its performance to that of the insensitive algorithm. We also present experimental results obtained by applying the heuristic to basic blocks from the SPEC benchmarks programs, for several machine models.

Internal Accession Date Only
A condensed version of this report will be published in and presented at the *28th Annual IEEE/ACM International Symposium on Microarchitecture,* November 28 - December 1, 1995, Ann Arbor, Michigan.

1 Background

Compilers that produce code for processors with instruction level parallelism need to jointly minimize the schedule length and the number of registers required. The development of quality heuristics for this problem is a difficult one, [7], [15], and is the focus of ongoing research. To gain an understanding of the tradeoffs involved between the schedule length and the number of registers, it is useful to analyze simplified abstractions of the problem. In this paper we examine a class of algorithms that aim to minimize schedule length in the face of a bounded number of available registers, in the absence of spill. The class of algorithms is characterized by a divide-and-conquer paradigm that converts any scheduling algorithm that is insensitive to register constraints, to one that respects register constraints. We study the performance of our algorithm on basic blocks drawn from the SPEC bechmarks, and a simple machine model of heterogeneous non-pipelined units with various choices of functional units and registers.

1.1 The Data Dependency Graph

It is the task of a compiler to convert a program into a sequence of instructions suitable for a target computer. Specifically, the compiler must decompose the program into instructions for the target computer, storing values on registers and manipulating these values as dictated by the semantics of the program. To do so, a typical compiler rewrites program fragments into an intermediate representation as follows. Each line in the intermediate representation would correspond to a single operation on the target machine. The operands in the intermediate representations would be virtual registers and memory locations, there being no limit on the number of virtual registers available. Assuming that each operation produces a single result from zero or more operands, the intermediate representation can be presented as a graph, with one vertex per operation, and an edge from vertex i to vertex j if the result of operation i is used in operation j. If the initial program code is a basic block, [1], or is superblock/hyperblock, [12], then the graph would be a directed acyclic graph or DAG.

In our discussion, we will ignore the process of constructing the intermediate representation graph, and assume that the graph is given to us as input. In particular, we limit ourselves to the following abstraction. We consider directed acyclic graphs (DAGs) where each vertex corresponds to an instruction, and is labeled with the virtual register defined in that instruction if any, an operation code, and a latency. The latency of a vertex is the time taken to complete the corresponding operation.

We use G to denote a graph, with vertex set V and edge set E. The indegree of a vertex is the number of edges entering a vertex. A DAG is said to be binary if its indegree is at most 2. The outdegree of a vertex is the number of edges leaving a vertex. We limit our discussions to binary DAGs, without limits on the outdegree. Superblocks and hyperblocks may carry vertices with indegree greater than two, since side exits may have a number of values that are live-out. These can be converted into binary graphs by adding dummy-zero latency vertices. A source in a DAG is a vertex with no incoming edges, and a sink is a vertex with no outgoing edges. A tree is a DAG with a single sink (the root of the tree) and the outdegree of any vertex in a tree is at most 1. A binary tree is a tree with indegree at most 2. The depth of a vertex in a DAG is the maximum sum of the latencies along any path from that vertex to a sink in the graph. Let v be a vertex in a DAG G, and let $U \subseteq V$ be such that for every $u \in U$, there exists a path from u to v. The subgraph G_v rooted at v is the subgraph of G induced by U.

1.2 The Machine Model

Our discussion is based on a machine model that is made as simple as possible to facilitate analysis, while retaining the inherent computational complexity of the problem. In particular, we study a machine model with a specified set of functional units, and a specified number of registers. All the registers are considered identical. With each functional unit comes a specification of the set of operations that can be carried out on it. The functional units may be pipelined, with distinct latencies. An operation on a functional unit can be a load from a memory location to a register, a store from a register to a memory location, a unary operation from one register to a second register, or a binary operation on two registers placing the result in a third register.

1.3 Problem Statement

Suppose that we are given a DAG and a machine model, and seek to translate the DAG into a schedule of instructions for the machine. The schedule would have to obey the precedence constraints specified by the edges in the DAG, as well as the latencies labeling the vertices of the DAG. Formally, a schedule for a DAG is an assignment of a time t_i and functional unit p_i to each vertex i satisfying

(1) If $i \rightarrow j$ is an edge of the DAG, $t_j > t_i + latency(i)$.

(2) Processor p_i is capable of carrying out the operation corresponding to vertex i.

(3) For any two vertices i and j such that $p_i = p_j$, $t_i + d(i) \leq t_j$ or $t_j + d(j) \leq t_i$, where $d(i)$ is the delay of operation i on the functional unit, i.e., the minimum time required after the start of i on the unit, to the start of a subsequent operation on the same unit.

For a given schedule, the lifetime of a virtual register v is the time interval $[s, f]$ where s is the time at which v is defined, and f is the time at which the last use of v terminates. We could equally well stipulate that the lifetime ends when the last use of v begins execution, but the difference between the two models is not central to the problem. A register allocation for a schedule is a mapping of the virtual registers to the physical registers. A register allocation is valid for a schedule if no two virtual registers with overlapping lifetimes are mapped to the same physical register. With the above preliminaries established, we can state the following problems.

Register Constrained Scheduling Problem: For a given DAG on a non-pipelined machine model of one functional unit and specified number of registers, construct any schedule with a valid register allocation, if such exists.

Minimum Length Scheduling: For a given DAG on a given machine model, construct a minimum length schedule.

Register Constrained Minimum Length Scheduling: For a given DAG on a given machine model, construct a minimum length schedule with a valid register allocation.

1.4 Prior Work

All three problems listed above are computationally intractable, even for the restricted case where all the functional units are identical and capable of all operations, and every operation has unit latency. Specifically, the Register Constrained Scheduling problem is NP-hard, [18], [8]; the Minimum Length Scheduling Problem is NP-hard, [8]; Register Constrained Minimum Length Scheduling is at least as hard as the Register Constrained Scheduling problem.

On the other hand, a number of algorithms are known for restricted cases of the above problems. In [19] an algorithm is given for scheduling a data dependency tree using a minimum number of registers, thereby addressing the Register Constrained Scheduling problem for trees. For the case

of a DAG, Klein et al., [13], give an approximate algorithm for the Register Constrained Scheduling problem that finds a schedule using at most $r^* log^2(N)$ registers, where N is the number of vertices, and r^* is the minimum number of registers for which a schedule exists. Specifically, their algorithm makes a minimum balanced directed cut of the DAG, i.e., splits the DAG G into two roughly equal pieces G_1 and G_2, so that the number of registers required to carry live values across the cut is minimized, and so that there are edges from G_1 to G_2, but no edges from G_2 to G_1. All of G_1 is then scheduled before G_2 by recursively applying the same procedure.

For the Minimum Length Scheduling problem, a number of good heuristics are known under the restrictions of unit latency and identical non-pipelined functional units. For example, the greedy scheduling algorithm is known to be optimal within a factor of $2 - 1/m$, [9], where m is the number of function units in the machine model. That is, the length of the schedule produced by the greedy algorithm is no more than $2 - 1/m$ greater than the shortest schedule. The critical path schedule given below, is known to be optimal for trees with unit latency, [11], and within a factor of $2 - 1/m$ for general DAGs on m functional units, [9]. Finally, the Coffman-Graham scheduling algorithm is within a factor of $2 - 2/m$ for unit latency DAGs on $m > 1$ functional units, [14], [5], and hence is optimal for $m = 2$.

The Critical Path Schedule
While there remains an unscheduled vertex
    On each idle functional unit, amongst vertices
    whose predecessors have already been scheduled
    schedule the one with the greatest depth first.
end

Given that the Minimum Length Scheduling problem is NP-hard, it is clear that the Register Constrained Minumum Length problem is NP-hard as well. The problem remains NP-hard even for trees, with heterogeneous processors as in [2], or homogeneous non-pipelined processors, [6]. However, for the very simple non-pipelined machine model of a single memory unit and a single arithmetic unit,[3], a heuristic is known that performs very close to optimal for trees. The heuristic is not extensible to multiple functional units, or to DAGS. There have been a number of hueristics proposed for the general case of the problem, as reviewed in [10]. Broadly speaking these decouple the problem into the subproblems of register allocation and minimum length scheduling. Prepass algorithms carry out the scheduling prior to register allocation, to scheduling, and postpass algorithms carry out the scheduling after register allocation. Goodman and Hsu, [10], suggest a heuristic to integrate register allocation and scheduling, but offer no estimate of goodness. In brief, their algorithm runs a scheduling strategy they call CSP, that attempts to minimize schedule length without regard to the number of registers used. When the number of registers crosses a threshold, the algorithm switches to a scheduling strategy they call CSR that attempts to minimize the number of registers used. When the number of registers in use falls sufficiently, the algorithm switches back to CSP. Pinter, [17], presents another heuristic for the Register Constrained Minimum Length Schedule problem. His strategy attempts to allocate registers in a manner that is guaranteed to be valid for all possible schedules for the input DAG. If the number of registers available is insufficient for such an allocation, scheduling-order constraints or spill is introduced in an ad hoc fashion. Global register allocation is the problem of partitioning the physical registers amongst the various blocks of a procedure so as to minimize the overall execution time of the procedure. This is discussed in [4], [16], wherein local schedulers–which are the subject of our study–are assumed to be given, and global allocators are built around them.

2 Theoretical Results

We denote a DAG G by its vertex set V and edge set E. The directed edge from u to v is denoted as the ordered pair (u, v) in E. If a DAG G has more than one sink, we combine these repeatedly with zero-latency vertices of indegree two. The edges used will not be dataflow edges, and will not participate in the definitions that follow. Henceforth, we assume that the DAG has only one sink, and we refer to this sink as the root.

We now define two properties that we will use in developing our scheduling algorithm. The first property is the thickness of a DAG $G = (V, E)$, which is a measure of the deviation of the graph from a tree. Let $v \in V$, and let U be the set of vertices in the subgraph rooted at v. Let t be the number of vertices in U with edges terminating in $V - U$, i.e., vertices $u \in U$ such $(u, \widehat{u}) \in E$ and $\widehat{u} \in V - U$. The thickness of G is the maximum value of t over all vertices v. It is easy to see that every tree has unit thickness. For our purposes, only the data flow edges of the precedence graph will be used in the definition of the thickness of the graph, and for the rest of the paper, we will retain this restriction tacitly. The second property is the notion of the weight $W(v)$ of a vertex v in a DAG G. As shown in Claim A.1 of the Appendix, the weight of a vertex is an upper bound on the number of registers required to compute the vertex on a processor with infinitely many functional units, in the shortest possible time. Let G_v denote the subgraph rooted at v.

$W(v) = \#(\text{sources } s \text{ in } G_v) +$
$\#\ (\text{edges in } G_v) - \#\ (\text{vertices in } G_v) + 1\ .$

If G is a tree, the weight of a vertex is the number of leaves in the subtree rooted at that vertex. We are now ready for our scheduling algorithm for DAG's.

Algorithm
input: DAG G, register bound r,
number of processors m.
begin
sched (root of G);
end

```
sched( vertex a)
    {
    if all registers are in use then fail;
    Schedule subgraph rooted at a using the
    insensitive algorithm without register constraints.
    if enough registers are available for this schedule then
        output schedule;
        delete scheduled vertices from G;
        free all registers containing dead values.
    else
        By depth-first-search find vertex b
        such that 1/4W(a) ≤ W(b) ≤ 1/2W(a);
        sched (b);
        sched (a);
    end
    }
```

The algorithm converts an insensitive scheduling algorithm, i.e., one that attempts to minimize schedule length, but is insensitive to register constraints, to one that attempts to minimize schedule length respecting register constraints. Initially, the algorithm attempts to schedule the entire DAG without regard to register constraints, using the insensitive algorithm. If the number of available registers is insufficient, the algorithm finds a vertex whose weight is roughly half that of the root of the DAG–in Claim A.3 of the Appendix, we show that it is always feasible to find such a vertex. The subgraph rooted at that vertex is scheduled recursively, and then the remaining portion of the graph is scheduled. The weights of the vertices are updated dynamically as the algorithm progresses.

We now analyze the performance of our algorithm. Let L_r be the length of the schedule produced by the algorithm, for r registers in the machine model. In this notation, L_∞ is the length of the schedule produced by the algorithm, for unboundedly many registers, and hence is the length of the schedule produced by the insensitive algorithm.

Claim 1: Let G be a binary DAG of thickness t, $R = W(root)$ the weight of the root of G, and r the number of registers available. Then, the length of the schedule constructed by the algorithm satisfies

$$L_r \leq 2^j L_\infty \ ,$$

where j is the smallest integer such that $(3/4)^j R \leq r - jt$. If no such j exists, the algorithm may fail.

Proof: If the algorithm does not call *sched* recursively, then the length of the schedule achieved is L_∞. If the algorithm calls *sched* recursively, the length of the schedule is at most doubled for each level of recursion. Let G be the graph at entry to a level and let $\widehat{G}$ be the graph at the entry to the next level when a recursive call is made to *sched*(a). Let $W(a)$ and $\widehat{W}(a)$ be the corresponding weights of a. By the definition of W, $W(a) = W(b) + \widehat{W}(a) + k$, where k is the number of edges from vertices in the subgraph of G rooted at b, to vertices in the subgraph of $\widehat{G}$ rooted at a. Since $W(b) \geq 1/4W(a)$, it follows that $\widehat{W}(a) \leq 3/4W(a)$. By Claim A.1 of the appendix, the number of registers required to schedule the subgraph rooted at a is at most $W(a)$. It follows that the recursion will bottom out at a level at which $W(a)$ is at most the number of free registers. When the recursion is j deep, at most jt registers can be in use at earlier levels in the recursion, since at each level at most t registers could be holding values all of whose uses are yet to occur. Hence, at least $r - jt$ registers are free at that level. Thus, the depth of the recursion cannot exceed the smallest integer satisfying $(3/4)^j R \leq r - jt$. □

We now give a corollary to the above claim for the particular case of identical non-pipelined units, and all operations being of equal latency. In this simple situation, we can relate L_∞ to L^*_∞, the optimal schedule length without register constraints, thereby relating the achieved schedule length L_r to L^*_∞.

Corollary: If the insensitive algorithm is the critical path scheduler, and all functional units are identical and non-pipelined, and all operations have equal latency, then

$$L_r \leq 2^{j+1} L^*_\infty \ .$$

Proof: The critical path scheduler is known to be within a factor of 2 of the optimal, [9], for identical, non-pipelined, equal latency units. Hence $L_\infty \leq 2L^*_\infty$ and the corollary follows. □

2.1 Interpreting the Theorem

Suppose that the number of available registers r is large compared to the fewest number of registers at which the DAG can be scheduled, and smaller than the number of registers sufficient to schedule the DAG using the insensitive algorithm. In this range, as a first approximation, the bound established in the theorem says that the achieved schedule length L_r varies as $(R/r)^{2.4}$. Hence, on the average a 10% increase/decrease in the number of available registers would increase/decrease the schedule length by roughly 24%.

2.2 Run Time

Claim 2: If the insensitive algorithm costs $T(n+e)$ time on a graph of n vertices and e edges, our algorithm costs $O(T(n+e)log(n+e))$ time.

Proof: The entire DAG could be scheduled by the insensitive algorithm once for each level of the recursion. Thus each level of the recursion costs $T(n+e)$. The number of levels of recursion is at most $O(log(R))$, which is $O(log(n+e))$. Hence the claim. □

2.3 An Improved Algorithm

Our basic algorithm cuts the DAG in two in the recursive step, and calls *sched* on each of the two pieces successively. While the simplicity of this approach aids in analysis and understanding, functional units may be idle during the tail of the schedule of the first piece, units that could have been used for the second piece thereby reducing the overall schedule length. We can modify the algorithm to overcome this limitation, as given below.

Modified Algorithm

```
input: DAG G, register bound r,
   number of processors m.
begin
   sched (root of G);
end

msched( vertex a)
   {
   if all registers are in use then fail;
   Schedule subgraph rooted at a using the
   insensitive algorithm without register constraints.
   if enough registers are available for this schedule then
      output one cycle of this schedule;
      delete the scheduled vertices from G;
      free all registers containing dead values.
   else
      By depth-first-search find vertex b
      such that 1/4W(a) ≤ W(b) ≤ 1/2W(a);
      msched (b);
      msched (a);
   end
   }
```

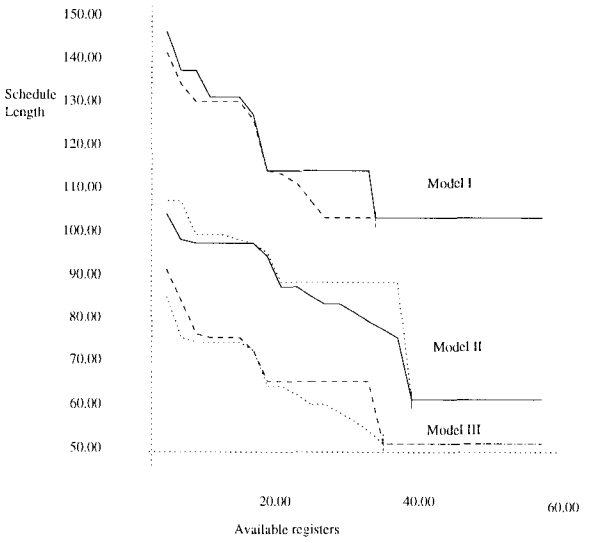

Figure 1: *Length of the schedule constructed by the algorithm as a function of the number of available registers for three different machine models, for a sample DAG of 137 vertices. For each model, the upper plot corresponds to the basic algorithm and the lower plot to the modified algorithm. The vertical tick marks on the plots mark the fewest registers sufficient for the critical path schedule.*

In the modified algorithm, the split vertex b can be selected dynamically so that as scheduling progresses, the split will remain balanced with respect to the weights of the two parts. The rebalancing is achieved by restricting $sched(b)$ to schedule just one instruction at each call, after which $sched(a)$ will be called again. At this point, the weights are updated and a new split vertex b is selected. This increases the computational complexity of the modified algorithm to $O(T^2(n+e)log(n+e))$ from the $O(T(n+e)log(n+e))$ of the basic algorithm.

3 Experimental Results

We now examine the performance of our basic and modified algorithms on some sample DAGS. Since critical path scheduling is a popular and effective scheduling algorithm in the absence of register constraints, we used critical path scheduling as the insensitive algorithm embedded in both our basic and modified scheduling algorithms. We applied both algorithms to DAGS generated by the IMPACT compiler on SPEC benchmark programs, with the optimizer turned on. We studied three machine models. Model I has 1 integer unit, 1 floating point unit and 1 memory unit for load and store instructions. Model II has 2 integer units, 1 floating point unit and 1 memory unit. Model III has 2 integer units, 2 floating point unit and 2 memory units. In each model, an integer operation takes 1 cycle, a floating point operation takes 4 cyles, loads take 2 cycles and stores 1 cycle. Since the behavior of our algorithms is essentially similar over the DAGS that we tested, we report our results on sample DAGs. Figure 1 shows a plot of the schedule length achieved by both the basic and modified algorithms as a function of the number of registers available, parametrized by the machine model on a DAG of 137 vertices. For each model, the upper plot is obtained from the basic algorithm and the lower plot from the modified algorithm. It is clear that the modified algorithm performs consistently better than the basic algorithm. Referring to Figure 1, the vertical tick marks on each of the the plots is the point at which the number of available registers

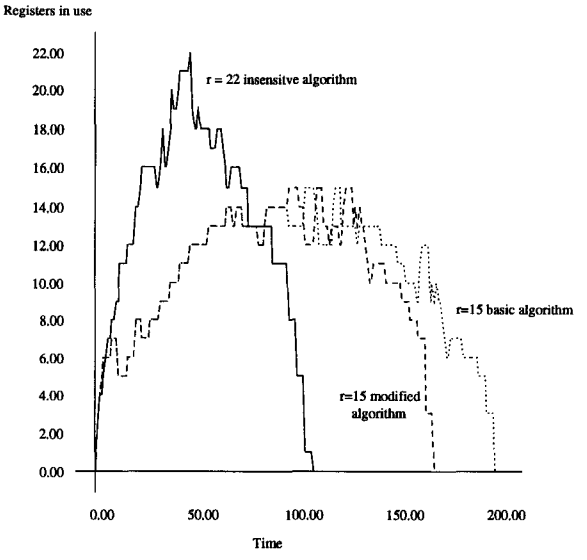

Figure 2: *Number of registers in use as a function of time for a DAG of 189 vertices, machine model III, and available registers of 15 and 22 respectively.*

r is just sufficient for the schedule constructed by the insensitive algorithm, i.e., the critical-path scheduler. At this value of r, neither algorithm recurses. If the number of registers is reduced by one from this value, the length of the schedule produced by the basic algorithm is sharply increased. This is because the basic algorithm breaks up the DAG into two pieces and schedules the pieces successively, potentially idling functional units. On the other hand, the schedule produced by the modified algorithm suffers a smooth degradation. Figure 2 is a plot of the register profile generated by the scheduling algorithms, i.e., the number of registers in use as a function of time, with respect to Model III, on a DAG of 189 vertices. There are three plots in the figure. The first plot corresponds to the insensitive scheduler i.e., without register constraints, and uses 22 registers. The second plot corresponds to the basic algorithm, constrained to use no more than 15 registers, and the third corresponds to the modified algorithm with the same constraint. Notice that the schedules constructed by both the basic and modified algorithms are longer than the unconstrained schedule constructed by the insensitive algorithm. Also, the schedule constructed by the modified algorithm is shorter than that of the basic algorithm. Both the constrained schedules have similar register profiles comprising an initial region with a build-up of register use, followed by a region of maximum register use, and then followed by a region of decaying register use.

4 Conclusion

We presented a heuristic for spill-free scheduling of acyclic predence graphs under register constraints. We also obtained estimates of the goodness of the heuristic. The incorporation of spill is an important extension to the heuristic in order to make it practicable.

Acknowledgements

Thanks to R. Motwani, B. Rau and N. Young for the discussions, and to R. Johnson for generating sample graphs with the IMPACT compiler.

References

[1] Aho, A., Sethi, R., and Ullman, J.D., *Compilers Principles, Techniques and Tools*, Addison Wesley, Reading, MA, 1988.

[2] Bernstein, D., Rodeh, M., and Gertner, I., "On the complexity of scheduling problems for parallel/pipelined machines," *IEEE Trans. Comp.*, Vol. 38, No. 9, pp. 1308-1313, 1989.

[3] Bernstein, D., Jaffe, J. M., and Rodeh, M., "Scheduling arithmetic and load operations in parallel with no spilling," *SIAM J. on Computing*, Vol. 18, No. 6, pp. 1098-1127, 1989.

[4] Bradlee, D.G, Eggers, S.J., and Henry, R.R., "Integrating register allocation and instruction scheduling for RISCs," *ACM Conf. on Arch. Support for Prog. Lang. and Op. Systems,* April, 1991.

[5] Braschi, B., and Trystram, D., "A new insight into the Coffman-Graham algorithm," *SIAM J. on Computing*, Vol. 23, No. 3, pp. 662-669, 1994.

[6] Chekuri, C., Private Communication, 1995.

[7] Ellis, J. R., *A compiler for VLIW architectures*, MIT Press, Cambridge, MA, 1985.

[8] Garey, M.R., and Johnson, D.S., *Computers and Intractability: A guide to the theory of NP-completeness,* Freeman, San Francisco, 1979.

[9] Graham, R., "Bounds on multiprocessor timing anomalies," *SIAM J. on Applied Math.*, Vol. 17, No. 2, pp. 416-429, 1969.

[10] Goodman, J.R., and Hsu, W-C, "Code scheduling and register allocation in large basic blocks," in *Proc. Intl. Conf. on Supercomputing*, pp. 442-452. 1988.

[11] Hu, T.C., "Paralell sequencing and assembly line problems," *Operations Research*, Vol. 9, pp. 841-848, 1961.

[12] Hwu, W.M., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringamann, R.A., Outllette, R.G., Hank, R.E., Kiyohara, T., Haab, G.E, Holm, J.G., and Lavery, D.M., "The superblock: An effective technique for VLIW and superscalar," *Journal of Supercomputing*, 7:229–248 (1993).

[13] Klein, P., Agrawal A., Ravi, R., and Rao, S., "Approximation through multicommidity flow," in *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pp. 726-737, 1990.

[14] Lam, S., and Sethi, R., "Worst case analysis of two scheduling algorithms," *SIAM J. on Computing*, Vol. 6, No. 3, pp518-536, 1977.

[15] Lowney G., et al., "The Multiflow Trace Scheduling Compiler," *Journal of Supercomputing*, Vol. 7, pp. 51-142, 1993.

[16] Norris, C., and Pollock, L.L., "A schedule-sensitive global register allocator," *Proc. Supercomputing*, pp. 804-813, 1993.

[17] Pinter, S., "Register allocation with instruction scheduling: a new approach," in *Proc. ACM SIGPLAN Conf. on Prog. Language Design and Implementation*, pp. 248-257, 1993.

[18] Sethi, R., "Complete register allocation problems," *SIAM J. on Computing*, Vol. 4, No. 3, pp. 226-248, 1975.

[19] Sethi, R., and Ullman, J.D., "The generation of optimal code for arithmetic expressions," *J. ACM*, Vol. 17, No. 4, pp. 715-728, 1970.

Appendix

Claim A.1: Let G be a DAG rooted at v. Every schedule of G can be achieved with $W(v)$ registers, independent of the number of functional units used.

Proof: By the definition of $W(v)$, we can reserve a register for every source vertex. We can also reserve a register for all but one outgoing edge of each vertex in G. At this point, executing any operation can only reduce the number of registers required. □

Claim A.2: For any interior vertex w,

$$W(w) \leq \sum_{(u,w)\in E} W(u) \; .$$

Proof: If there is only one vertex u such that (u, w) is an edge, then it follows from the definition of W that $W(w) = W(u)$ and the claim holds. Suppose that there are two vertices u and v with edges to w. Then,

$W(u) + W(v) =$
$\#(\text{sources only in } G_u) +$
$\#(\text{sources only in } G_v) +$
$2\#(\text{sources in both } G_u \text{ and } G_v) +$
$\#(\text{edges only in } G_u) +$
$\#(\text{edges only in } G_v) +$
$2\#(\text{edges in both } G_u \text{ and } G_v) -$
$\{\#(\text{vertices only in } G_u) +$
$\#(\text{vertices only in } G_v) +$
$2\#(\text{vertices in both } G_u \text{ and } G_v)\} + 2 \; .$

Now

$\#(\text{sources in } G_w) =$
$\#(\text{sources only in } G_u) +$
$\#(\text{sources only in } G_v) +$
$\#(\text{sources in both } G_u \text{ and } G_v) \; .$

and

$\#(\text{edges in } G_w) =$
$\#(\text{edges only in } G_u) +$
$\#(\text{edges only in } G_v) +$
$\#(\text{edges in both } G_u \text{ and } G_v) + 2,$

where the additional 2 is the contribution of the edges (u, w) and (v, w). And finally,

$\#(\text{vertices in } G_w) =$
$\quad \#(\text{vertices only in } G_u) +$
$\quad \#(\text{vertices only in } G_v) +$
$\quad \#(\text{vertices in both } G_u \text{ and } G_v) + 1,$

where the additional 1 is the contribution of w. It follows that

$W(u) + W(v) - W(w) =$
$\quad \#(\text{sources in both } G_u \text{ and } G_v) +$
$\quad \#(\text{edges in both } G_u \text{ and } G_v) -$
$\quad \#(\text{vertices in both } G_u \text{ and } G_v)$.

We now show that expression on the right hand side of the above equation is non-negative. To do so, it suffices to consider the vertices that are both in G_u and G_v. If such a vertex is a source, then it occurs in the first term as well, and its net contribution to the sum is zero. Else, such a vertex has incoming edges and these edges must occur in the second term and the net contribution must be non-negative. The claim follows. □

Claim A.3: If v is a vertex with $W(v) \geq 2$, there exists a vertex u in the subgraph rooted at v such that

$$1/4W(v) \leq W(u) \leq 1/2W(v) .$$

Proof: Set $w = v$. Let u be such $W(u)$ is a maximum among all vertices with an edge to v. If $W(u) \geq 1/2W(v)$, set $w = u$ and repeat until $W(u) \leq 1/2W(v)$. By Claim A.2,

$$W(w) \leq \sum_{(x,w) \in E} W(x) .$$

Since the DAG has indegree at most 2, if $W(u) \leq 1/4W(v)$, then $W(w) \leq 1/2W(v)$, which is not possible. Hence the claim. □