Published in the Proceedings of International Parallel and Distributed Processing Symposium(IPDPS)'2001 San Francisco, Apr 23--27, 2001. ©2001 IEEE

Minimum Register Instruction Sequence Problem: Revisiting *Optimal Code Generation* for DAGs

R. Govindarajan †, H.Yang ‡, J. N. Amaral ♯, C. Zhang ‡, G. R. Gao ‡

† Supercomputer Education & Research Centre
Dept. of Computer Science & Automation
Indian Institute of Science
Bangalore, 560 012, INDIA
govind@csa.iisc.ernet.in

‡ Electrical and Computer Engineering
University of Delaware
Newark, DE 19716
Delaware, U.S.A.
{hyang,czhang,ggao}@capsl.udel.edu

♯ Dept. of Computing Science
University of Alberta
Edmonton T6G 2E8
Alberta, CANADA
amaral@cs.ualberta.ca

Abstract

We revisit the optimal code generation *or* evaluation order determination *problem — the problem of generating an instruction sequence from a data dependence graph (DDG). In particular, we are interested in generating an instruction sequence S that is optimal in terms of the number of registers used by the sequence S. We call this MRIS (Minimum Register Instruction Sequence) problem.*

We developed an efficient heuristic solution for the MRIS problem based on the notion of instruction lineage*. This solution facilitates the sharing of registers among instructions within a lineage and across lineages by exploiting the structure of a DDG. We implemented our solution on a production compiler and measured the reduction in the number of (spill) loads and (spill) stores and the wall-clock execution time for the SPEC95 floating point benchmark suite. On average our method reduced the number of loads and stores by 11.5% and 15.9 %, respectively, and decreased the total execution time by 2.5%.*

1 Introduction

The objective of the the *optimal code generation* problem [1, 24] — also known as the *evaluation order determination* problem [30] — is the generation of an instruction sequence for a data dependence graph (DDG). In particular, we are interested in generating an instruction sequence *S* that uses a minimum number of registers. We call this MRIS (Minimum Register Instruction Sequence) problem:

> Given a data dependence graph *G*, derive an *instruction sequence S* for *G* that is optimal in the sense that its register requirement is minimum.

The MRIS problem is related to but different from the conventional problem formulation of instruction scheduling [1, 11, 12, 18, 29] and register allocation [1, 6, 7, 10, 18, 20, 22, 28]. We highlight these differences in Section 5.

Our study of the MRIS problem is motivated by the challenges faced by modern processor architectures. For example, a number of modern superscalar processors support out-of-order instruction issue and execution [27]. Out-of-order (o-o-o) instruction issue is facilitated by a runtime scheduling hardware and by the register renaming mechanism. An o-o-o processor has a larger number of physical (*i.e.,* not architected) registers at its disposal for register renaming at runtime. Recent studies [16, 26] indicate that compilers for o-o-o issue processors should emphasize reducing the number of register spills over exposing instruction level parallelism. Reducing register spills reduces the number of loads and stores executed and is important:

- from a performance viewpoint in architectures that either have a small cache or a large cache miss penalty;
- from a power dissipation viewpoint, as the contribution of load/store instructions consume a significant portion of the power budget.

As another argument to why the MRIS problem is relevant, consider Intel's IA-64 architecture [4] where a variable sized register window is allocated in the register stack for each function, and register spilling is automatically performed by the register stack engine (RSE). In that architecture the problem faced by the local register allocator is to minimize the number of registers allocated to avoid unnecessary spilling by the RSE, and not to optimize the use of a fixed number of registers. Lastly, in code generation for fine-grain multithreaded architectures [9], it is often more important to minimize the number of registers used in a thread in order to reduce the cost of thread context switch.

In this paper we present a simple and efficient heuristic method to address the MRIS problem. The proposed method is based on the following:

- *Instruction lineage formation:* The concept of *instruction lineage* evolves from the notion of *instruction chains* [13] which allows the sharing of a register among instructions along a (flow) dependence chain in a DDG. However, the instruction lineage approach more accurately models the register requirement than an instruction chain (as will be discussed in Section 5).

- The notion of a lineage interference that captures the *must* overlap relation between the live ranges of lineages even before the lineages are scheduled, which facilitates sharing registers across lineages.

We implemented our heuristic approach in the SGI MIPSpro compiler suite and evaluated our approach on the SPEC95 floating point benchmarks. For comparison we also measured the performance of a baseline version (base) of the compiler that performs traditional compiler optimizations, but does not optimize the instruction sequence. As the emphasis of our work is on sequencing the instructions to reduce the register requirements and spill code, we measured the number of loads and stores graduated in each benchmark under each version of the compiler. We also report wall-clock execution time. The results are summarized as follows.

- Our heuristic approach reduces the number of loads and stores executed on the average by 11.5% and 13.9% respectively (compared with a baseline version of the compiler).

- The heuristic method also results in an average execution time improvement of 2.5% on the SPEC95 floating point benchmarks.

- The heuristic method performs competitively, in terms of execution time, compared to a combined instruction scheduling/register allocation algorithm. The proposed heuristic approach also results in a marginal reduction (2.8% and 1.7%, respectively) in the number of loads and stores.

The rest of the paper is organized as follows. In the following section we motivate the MRIS problem with the help of an example. In Section 3, we present our heuristic solution to the MRIS problem. Experimental results are discussed in Section 4. Related work and conclusions are presented in Sections 5 and 6.

2 Motivating Example

In this section we use an example to motivate the MRIS problem and illustrate our solution to it. Consider the computation represented by a data dependence graph (DDG)[1] shown in Figure 1(a). Two possible instruction sequences for this DDG are also shown in the figure along with the live ranges of the variables $s1$–$s7$ (for simplicity, we assume that all the variables are dead at the end of the basic block). For the instruction ordering shown in Figure 1(b) we have four variables simultaneously live in statements e and f, therefore four registers are required. However, with the sequencing shown in Figure 1(c) only three variables are simultaneously live and therefore we may use only three registers. In this particular example, the minimum register requirement is three. Hence the sequence shown in Figure 1(c) is a minimum register sequence.

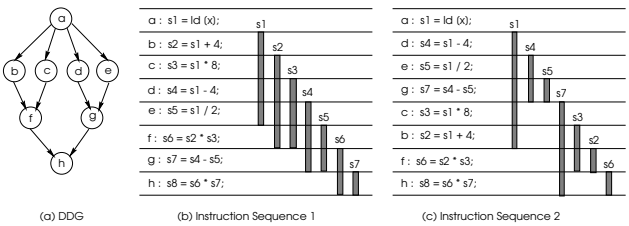

Figure 1. Motivating Example

The input for the MRIS problem is a DDG that defines only a partial order among instructions. By restricting our attention to acyclic DDGs we do not consider any loop carried dependencies. We identify nodes (instructions) in the DDG that can share the same register (*i.e.,* use the same register as destination) in a legal sequence. Although a complete answer to this question is hard to determine, the data dependence in the DDG does provide a partial answer. For instance, in the DDG of Figure 2(a), since there is a data dependency from node b to node f, and there is no other node that uses the value produced by node b, we can definitely say that in any legal sequence, the register associated with node b can be shared by f. Similarly nodes e and g can share the same register. Can nodes f and g share the same register? The answer is no, as, in any legal sequence, the values produced by these instructions must be live simultaneously so that the computation of h can take place.

Another interesting question is whether nodes c and d can share the same register. The data dependence in the DDG neither requires their live ranges to definitely overlap (as in the case of nodes f and g) nor implies that they definitely will not overlap (as in the case of nodes b and f). Hence to obtain a minimum register instruction sequence, one must order the nodes in such a way that the live ranges of c and d do not overlap, and thus they can share the same register. In the following subsection, we outline our first approach which uses efficient heuristics to arrive at a near-optimal solution to the MRIS problem.

[1] Since our focus in this paper is on register allocation, we consider only flow dependences in our DDG. Other dependences due to memory (such as store–load dependences), while important from a scheduling viewpoint, do not influence register allocation and hence need not be considered for computing the register requirements.

3 Lineage Based Instruction Sequencing

In this section, we present our heuristic approach to find a minimum register sequence for a given acyclic DDG.

3.1 Overview of Our Approach

Our approach to the MRIS problem uses the notion of *instruction lineages* that is derived from the notion of *chains* introduced in [13]. If a node S in a DDG has one or more descendents, it produces a value and thus must be assigned a register. If we have a sequence of instructions in the DDG $S_1, S_2, S_3, \cdots, S_n$ where S_2 is the descendent of S_1, S_3 is the descendent of S_2, *etc.,* then we can form a *lineage* of these instructions in such a way that all the instructions in the lineage can share the same register. That is, the register assigned to S_1 is passed on to S_2 (S_1's *heir*) which is passed on to S_3, and so on. Due to the data dependency between pairs of instructions in the lineage, any legal sequence will order the instructions as S_1, S_2, $\cdots$, S_n. Hence, if the live range of the variable defined by S_1 ends at S_2, then S_2 can definitely share the same register allocated to S_1.

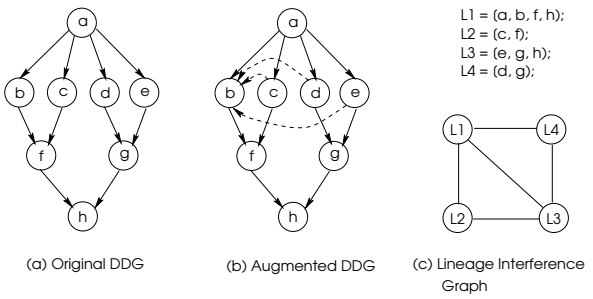

Figure 2. Data Dependence Graph for the Motivating Example

What if S_1 has more than one descendent? In order for S_2 to use the same register that S_1 used, we must ensure that the other descendents of S_1 are scheduled before S_2. Thus the selection of one of the descendents of S_1 to be the *heir* of the register creates scheduling constraints among the descendents of S_1. Such sequencing constraints are explicitly represented in the *augmented DDG* by directed *sequencing edges* from each descendent node to the selected heir. For instance, the DDG for the motivating example is shown in Figure 2(a). The definition of the lineage $L1 = \{a, b, f, h\}$ creates scheduling constraints between the descendents of a as shown by dotted arrows in Figure 2(b). In Section 3.2 we introduce a simple but efficient heuristic to select the heir among multiple descendents.

It is clear that all the nodes in a lineage share the same register. But can two lineages share the same register? To determine the interference between two lineages, we have to determine whether the live ranges of the lineages overlap in all legal instruction sequences. The live range of an instruction lineage is the concatenation of the live ranges of all the values defined by the instructions in the lineage. If the live ranges of two lineages overlap in all legal sequences, then the lineages cannot share the same register. However if they do not overlap in at least one of the legal sequences, then we may be able to schedule the lineages in such a way that they share a register. Given the DDG and the partial ordering of instructions, how do we determine whether or not the live ranges of two lineages will or will not overlap? We address this in Section 3.3. Based on the overlap relation we construct a lineage interference graph.

This lineage interference graph is colored using traditional graph coloring algorithms [7] to compute the number of registers required for the DDG. We refer to this number as the *Heuristic Register Bound (HRB)*. Once we color the lineage interference graph, we apply a modified list scheduling method that uses this coloring as a guideline to generate an instruction sequence that uses the minimum number of registers. Our heuristic-based algorithm produces a near-optimal solution for the MRIS problem.

3.2 Lineage Formation

First we define the notion of an instruction lineage.

Definition 3.1 *An* **instruction lineage** *is a sequence of nodes* $\{v_1, v_2, \cdots, v_n\}$ *such that there exist arcs* (v_1, v_2), (v_2, v_3), $\cdots$, (v_{n-1}, v_n) *in the DDG. Further in the above lineage,* v_2 *is the* **heir** *of* v_1, v_3 *is the heir of* v_2 *and so on.*

A node may have many descendents, but it has at most a single heir. Because the heir is always the last descendent to be scheduled (it is the *last use* of the value produced by its parent), in an acyclic scheduling, the live range of the nodes in a lineage will definitely not overlap with each other and hence all the nodes in the lineage can share the same register. To ensure that the heir is the last use of the value produced by its parent, we introduce *sequencing edges* in the DDG. We need a sequencing edge from each descendent of a node to the chosen heir, as shown in Figure 2(b).

If the introduction of sequencing edges were to make the graph cyclic, then it would be impossible to sequence the instructions represented in the DDG. Hence some care should be taken in the selection of a heir. During the formation of the instruction lineages, we use a simple height priority to choose the heir of each node. The height of a node is defined as follows:

$$ht(u) = \begin{cases} 1 & \text{if } u \text{ has no descendents} \\ 1 + \max_v(ht(v)) & \text{for } v \in \text{descendents}(u) \end{cases}$$

The heights of the nodes in the DDG of Figure 2(a) are:

$$ht(a) = 4; \quad ht(b) = 3; \quad ht(c) = 3; \quad ht(d) = 3;$$

$$ht(e) = 3; \quad ht(f) = 2; \quad ht(g) = 2; \quad ht(h) = 1;$$

During the lineage formation, if a node v_i has multiple descendents, then we choose a descendent node with the smallest height. If multiple descendents have the same height, the tie is broken arbitrarily. In order to ensure that cycles are not introduced in the lineage formation process, our approach recomputes the height of each node after introducing a set of sequencing edges between the descendents of a node.

The last node v_n in a lineage is the last one to use the value in the register (defined by $v_{(n-1)}$) assigned to that lineage. The last node v_n might belong to another lineage or it might be a store instruction that does not need a register. Therefore the last node does not share its register with the other nodes in the lineage. We emphasize this property of the last node by representing a lineage as a semi-open interval $[v_1, v_2, \cdots, v_{(n-1)}, v_n)$. In the DDG of Figure 2, the four lineages are $[a,b,f,h)$, $[c,f)$, $[e,g,h)$, and $[d,g)$. Lastly, in forming lineages, we ensure that the live range of each node (except the sink nodes) is included in exactly one of the lineages.

The Lineage Formation algorithm is essentially a depth-first graph traversal algorithm, identifying a heir for each node (using the height priority). In cases where a node has multiple descendents, sequencing edges are introduced, and the heights of all nodes in the DDG are recomputed. The detailed algorithm is presented in [14].

Lemma 3.1 *The introduction of sequencing edges during lineage formation does not introduce any cycle in the augmented DDG.*

The reader is referred to [14] for a proof of this and other lemmas/theorems.

3.3 Lineage Interference Graph and Heuristic Register Bound

In defining the live range of a lineage we use the fact that each instruction has a unique position t in the sequence of instructions.

Definition 3.2 *For a lineage $L = [v_1, v_2, \cdots, v_n)$, if v_1 and v_n are at positions t_1 and t_n, respectively, in an instruction sequence, then the* **live range** *of the lineage L starts at t_1 and ends at t_n.*

While identifying lineages with non-overlapping live ranges, the instruction sequence is not yet fixed, and as a consequence, the live ranges of different lineages are all *floating*. In this context, determining whether two lives ranges must necessarily overlap in any instruction sequence can be accomplished using the following condition. Note that the live range of a lineage is always contiguous, and once the first instruction in a lineage is listed, the lineage's live range is active until the position of the last node of the lineage in the instruction sequence.

Theorem 3.1 *The live ranges of two lineages $L_u = [u_1, u_2, \cdots, u_m)$ and $L_v = [v_1, v_2, \cdots, v_n)$* **definitely overlap** *if there exist directed paths from u_1 to v_n and from v_1 to u_m in the augmented DDG.*

Consider the lineages $L_1 = [a,b,f,h)$ and $L_4 = [d,g)$ in our motivating example in Figure 2(b). Node a can reach node g through the path $a \rightarrow d \rightarrow g$, and node d can reach node h through the path $d \rightarrow b \rightarrow f \rightarrow h$. Therefore, the live ranges of these two lineages must overlap. Similarly the live range of lineage L_2 overlaps with L_1, L_3 overlaps with L_4, L_1 overlaps with L_3, and L_2 overlaps with L_3.

Next we construct a lineage interference graph I which is an undirected graph, where the vertices represent the lineages. Two vertices are connected by an interference edge only if the live ranges of the lineages represented by them definitely overlap. The lineage interference graph can be colored using a heuristic graph coloring algorithm [7]. We refer to the number of colors required to color the interference graph as the Heuristic Register Bound (HRB). The lineage interference graph for the motivating example is shown in Figure 2(c). Its HRB is 3.

3.4 Lineage Fusion

In this section we discuss an optimization that helps to reduce HRB. After the lineages are formed as described in Section 3.2, pairs of lineages that satisfy the following condition can be fused into a single lineage:

Definition 3.3 *Two lineages $L_u = [u_1, u_2, \ldots, u_m)$ and $L_v = [v_1, v_2, \ldots, v_n)$ can be fused into a single lineage if:*

i. there exists a directed path from u_1 to v_n;

ii. there is no directed path from v_1 to u_m;

When lineages L_u and L_v are fused together, a sequencing edge from u_m to v_1 is inserted in the DDG, the lineage L_u and L_v are removed from the lineage set, and a new lineage $L_w = [u_1, u_2, \ldots, u_m) \cup [v_1, v_2, \ldots, v_m)$ is inserted in the lineage set. The last node of the first lineage, u_m, does not necessarily uses the same registers as the other nodes in the new L_w lineage. Fusing two lineages L_u and L_v causes the respective nodes u and v in the interference graph to be combined into a single node, say w. Every edge incident on u or v is now incident on w. Lineage fusion might result in additional interference edges which need to be added to the lineage interference graph.

Lineage fusion helps to reduce the number of partially overlapping live range pairs, and thereby reduces the register requirements. It accomplishes this by fusing the two lineages corresponding to the partially overlapping live ranges into one, and forcing an instruction sequence order on them. Lineage fusion is applied after lineage formation and before the coloring of the lineage graph. Therefore the interference graph to be colored after lineage fusion has fewer vertices.

3.5 Instruction Sequence Generation

Coloring the lineage interference graph associates a register with each lineage. For example, the register assignment A for the nodes in our motivating example is:

$$A = \{(a,R_1);(b,R_1);(f,R_1);(c,R_2);(d,R_2);(e,R_3);(g,R_3)\}$$

Our sequencing method is based on the list scheduling algorithm. Refer to [14] for the detailed algorithm. It uses the register allocation information obtained by coloring the lineage interference graph. The sequencing method *lists* nodes from a Ready List based on height priority and on the availability of registers assigned to them. The sequencing algorithm uses the augmented DDG with sequencing edges. The availability of register needs to be checked only if node v_i is the first node in its lineage. Otherwise, the predecessor of v_i would pass the register assigned to it, which would be free when v_i is ready to be listed.

Unfortunately, the above sequencing algorithm could result in a deadlock due to two reasons. First, the order of listing of two nodes belonging to two different lineages that are assigned the same color may result in a deadlock. We refer to these deadlocks caused by wrong ordering of nodes as *avoidable deadlocks*. The second kind of deadlocks referred to as *unavoidable* deadlocks are caused due to the underestimation of HRB.

Applying the sequencing algorithm to our motivating example, first we list node a. If lineage fusion is not used, the scheduling of node a causes nodes c, d, and e to be added to the Ready List. Since register R_2 is available, either node d or node c can be listed next. There is no criterion to choose c or d and therefore the tie is broken arbitrarily. Unfortunately, if node c is listed before d, a cycle of dependences is created because node d cannot be scheduled before node f (scheduling node f will release the register R_2 currently used by c to d). On the other hand, in order to schedule f we must first schedule b, but b cannot be scheduled before d because b must be the last use of R_1 assigned to a. This is an avoidable deadlock that can be solved by lineage fusion.

If lineage fusion is employed, lineages $L4$ and $L2$ are fused together and a scheduling edge from node g to node c is added to the graph. Thus after node a is listed (using R_1), only nodes d and e can be listed. Suppose that node d is listed next (using R_2). Now the only available register is R_3. Hence we list node e. The nodes g, c, b, f, and h are then listed in that order. This instruction sequence requires only three registers as shown in Figure 1(c).

Unfortunately even with the application of lineage fusion, *unavoidable deadlocks* occur when the HRB computed by coloring the lineage interference graph is lower than the actual number needed. In this case there does not exist a legal instruction sequence that uses HRB or fewer registers. We overcome the deadlock problem by gradually increasing the HRB by 1. We estimate the performance of our heuristic approach in Section 4.

4 Experimental Results

In this section, we describe our experimental framework, and present the experimental results.

4.1 Experimental Framework

We implemented the instruction sequencing algorithm described in Section 3 in the SGI MIPSpro compiler suite, a set of highly-optimizing compilers for Fortran, C, and C++ on MIPS processors.

The base compiler performs several optimizations including copy propagation, dead-code elimination, if-conversion, loop unrolling, cross-iteration optimization, recurrence breaking, instruction scheduling, and register allocation. It also implements an integrated global-local scheduling algorithm [15] that is invoked before and after register allocation. Subsequently a global register allocation based on graph coloring [8, 6] followed by local register allocation, and a postpass scheduling are performed in the base compiler.

Our heuristic register bound based algorithm, described in Section 3, is used to optimize the instruction sequence at the basic block level. This local optimization is applied only to basic blocks that require spill code under the initial local register allocation. After the instruction sequence is optimized, the local register allocation is invoked on the new instruction sequence. We refer to this version of the compiler as the HRB approach

The performance of the HRB optimized version is evaluated against the baseline version. We also measure the HRB approach against an optimized version of the MIPSpro compiler which includes a combined instruction scheduling and register allocation algorithm. This version is referred to as the Optimized MIPSpro version.

We performed our experiments on a Silicon Graphics O2 machine with a 300 MHz MIPS R12000 processor, 32 KB instruction cache, 32 KB data cache, and 1MB secondary unified instruction/data cache. We configured the register file to use 32 integer registers, and 16 floating point registers. We report results for the benchmarks in the SPEC95 Floating Point suite. We measured the wall-clock time for the execution of each benchmark under the IRIX 6.5 operating system with the machine running in a single user mode. As the emphasis of our work is on sequencing the instructions to reduce the register requirements and spill code, we used the R12000 hardware counters and the `perfex` tool to measure the number of loads and stores graduated in each benchmark under each version of the compiler. Since the baseline and HRB versions of the compiler are identical except for the instruction reordering at the basic block

Benchmark	Loads			Stores			Execution Time		
	Baseline (in Billions)	HRB	Reduc. (%)	Baseline (in Billions)	HRB	Reduc. (%)	Baseline (in seconds)	HRB	Improv. (%)
tomcatv	10.08	8.51	15.5	4.3	3.42	20.5	429	426	0.7
swim	10.17	8.04	20.9	4.26	3.08	27.7	338	333	1.5
su2cor	7.47	7.04	5.7	3.29	2.87	12.8	248	247	0.4
hydro2d	10.17	10.17	0	3.39	3.39	0	800	812	-1.5
mgrid	21.06	20.63	2.1	1.92	1.48	22.6	373	378	-1.3
applu	16.29	13.69	15.9	7.77	6.84	11.9	503	490	2.6
turb3d	20.61	17.91	13.1	16.51	13.98	15.3	381	366	3.9
apsi	7.23	6.31	12.8	4.05	3.19	21.1	191	188	1.6
fpppp	58.42	44.51	23.8	23.40	23.50	−0.4	332	279	16.0
wave5	5.70	5.26	7.7	4.18	3.84	8	230	228	0.9
mean	16.7	14.2	11.8	7.3	6.56	13.9	382	375	2.5

Table 1. Comparison between the HRB approach and the baseline version of the compiler.

level, the reduction in the number of loads/stores executed in each benchmark program correspond to the number of spill loads/stores reduced by the HRB approach.

4.2 Reduction in Loads/Stores

Table 1 shows the number of loads and stores graduated and the execution time for each benchmark, when compiled using the HRB approach and with the baseline compiler. The percentage reduction in the number of loads for a benchmark for the HRB sequencing is computed as follows:

$$\%\text{age Load Reduction}(HRB) = \frac{loads(Baseline) - loads(HRB)}{loads(Baseline)} \times 100$$

When compared with the baseline compiler, our HRB approach reduced the number of loads by as much as 24%, and the number of stores by as much as 28%. Further, the average reduction in loads is 11.8% and stores is 13.9%. The reduction of loads and stores is significant and very encouraging considering the arguments about out-of-order execution and power consumption laid out in the introduction.

4.3 Improvements in Execution Time

Next, we present the comparison on the total (wall-clock) execution time for all benchmarks between the HRB approach and the baseline compiler. The improvement over the baseline is as high as 16% for the `fpppp` benchmark. On average the reduction in the spill code required by the lineage based algorithm results in an improvement of 2.5% over the baseline version of the compiler. The two exceptions are `hydro2d` and `mgrid`. A possible explanation for this is that the reordering of instructions performed by the HRB approach creates resource conflicts that cannot be resolved by the dynamic scheduler of the R12000.

4.4 Comparison with the Optimized Version

Lastly, we compare our HRB approach with the optimized implementation in the MIPSpro compiler in terms of graduated loads and stores and execution time in Table 2. The HRB approach results in less loads and stores than the optimized version. The execution times of the two versions are comparable. In all but one case, the HRB approach has a slightly lower execution time than the optimized compiler. The increase in the execution time for `mgrid` might be caused by an interference between the HRB instruction reordering and the out-of-order issuing mechanism of the R12000. These results are encouraging, considering that the HRB sequencing did not use resource constraints. We expect that using resource constraints in the sequencing phase of the HRB approach will further improve its performance.

5 Related Work

Instruction scheduling [11, 18] and register allocation [1, 6, 7, 10, 18, 20, 22, 28] are important phases in a high performance compiler. The ordering of these phases and its implications on the performance of the code generated have been studied extensively for in-order issue superscalar processors and Very Long Instruction Word (VLIW) processors. In such processors it is often necessary to expose enough instruction-level parallelism even at the expense of increasing the register pressure and, to some extent, the amount of spill code generated. Integrated techniques that try to minimize register spills while focusing on exposing parallelism were found to perform well [5, 3, 17, 19, 21]. All of the above approaches work on a *given* instruction sequence and attempt to improve register allocation and/or instruction schedule. In contrast our MRIS approach, generates an instruction sequence from a DDG where the precise order of instructions is not yet fixed.

Benchmark	Loads			Stores			Execution Time		
	MIPSpro (in Billions)	HRB	Reduc. (%)	MIPSpro (in Billions)	HRB	Reduc. (%)	MIPSpro (in seconds)	HRB	Improv. (%)
tomcatv	8.80	8.51	3.33	3.61	3.42	5.4	430	426	0.9
swim	8.40	8.04	4.25	3.08	3.08	0.0	333	333	0.0
su2cor	7.04	7.04	0.0	2.87	2.87	0.0	248	247	0.4
hydro2d	10.17	10.17	0.0	3.39	3.39	0.0	812	812	0.0
mgrid	20.63	20.63	0.0	1.48	1.48	0.0	367	378	−3.0
applu	13.66	13.69	−0.28	6.41	6.84	−6.78	493	490	0.6
turb3d	19.18	17.91	6.66	14.15	13.98	1.17	373	366	1.9
apsi	6.64	6.31	5.03	3.52	3.19	9.33	191	188	1.6
fpppp	46.32	44.51	3.92	24.11	23.50	2.56	291	279	4.1
wave5	5.52	5.26	4.8	4.07	3.84	5.57	230	228	0.9
mean	14.6	14.2	2.8	6.67	6.56	1.7	377	375	0.7

Table 2. Comparison between the HRB approach and the MIPSpro optimized version

The Minimum Register Instruction Sequence (MRIS) problem studied in this paper is different from the traditional register allocation problem [1, 6, 7, 10, 18, 20]. The input to the MRIS problem is the partial order specified by a DDG instead of a totally ordered sequence of instructions. Although the absence of a total order of instructions makes the MRIS problem harder, it also enables the generation of an instruction sequence that requires less registers. The MRIS problem is also quite different from the traditional *instruction scheduling problem* [1, 11, 12, 18, 29]. In the traditional instruction scheduling problem, the main objective is to minimize the total time (length) of the schedule, taking into account the execution latencies of each operation (instruction) in the DDG and the availability of function unit resources. This is in contrast to the MRIS problem, where only the true dependence constraints are observed.

The MRIS problem is closely related to the *optimal code generation (OCG)* problem [1, 24, 23]. An important difference between traditional code generation methods and our MRIS problem is that the former emphasizes reducing the code length (or schedule length) for a fixed number of registers, while the latter minimizes the number of registers.

The unified resource allocator (URSA) method deals with function unit and register allocation simultaneously [2]. The method uses a three-phase *measure–reduce–assign* approach, where resource requirements are measured and regions of excess requirements are identified in the first phase. The second phase reduces the requirements to what is available in the architecture, and the final phase carries out resource assignment. More recently, they (Berson, Gupta and Soffa) have used *register reuse dags* for measuring the register pressure [3]. A register reuse dag is similar to a lineage discussed in this paper. They have evaluated register spilling and register splitting methods for reducing the register requirements in the URSA method.

Lastly, the lineage formation and the heuristic list scheduling methods proposed in this paper are major improvements over, respectively, the chain formation and the modified list scheduling method discussed in [13]. The chain formation method allocates, at least conceptually, one register for each arc in the DDG, and must cover all arcs. That is, it must include each def-use, not just def-last-use, in a chain. Hence, the instruction lineage approach more accurately models the register requirement. Secondly, the lineage formation overcomes an important weakness of instruction chains, namely allocating more than one register for a node. Further, a number of heuristics have been incorporated into the sequencing method to make it more efficient and obtain near-optimal solutions.

6 Conclusions

In this paper we address the problem of generating an instruction sequence for a computation that is optimal in terms of the number of registers used by the computation. This problem is motivated by requirements of modern processors. We proposed a heuristic solution that uses lineage formation, lineage interference graph and a modified and efficient list scheduling method. We evaluated the performance of our heuristic method by implementing it on the MIPSpro production compiler, and on SPEC95 floating point benchmarks. Our experimental results demonstrate that our instruction reordering method which attempts to minimize the register requirements, reduces the number of loads and stored executed, on the average, by 11.5% and 13.9% respectively. This reduction also results in an execution time improvement of 2.5% on the average. Lastly, our heuristics method performs competitively compared to a combined instruction scheduling/register allocation algorithm.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley Publishing Co., Reading, MA, corrected edition, 1988.

[2] D. Berson, R. Gupta, and M. L. Soffa. URSA: A Unified ReSource Allocator for registers and functional units in VLIW architectures. In *Proc. of the Conf. on Parallel Architectures and Compilation Techniques, PACT '98*, Paris, France, June 27–29, 1998.

[3] D. Berson, R. Gupta, and M. L. Soffa. Integrated instruction scheduling and register allocation techniques. In *Proc. of the Eleventh International Workshop on Languages and Compilers for Parallel Computing, LNCS, Springer Verlag*, Chapel Hill, NC, Aug. 1998.

[4] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the IA-64 Architecture. *IEEE Micro* 20(5): 12-23, 2000.

[5] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Proc. of the Fourth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, Santa Clara, CA, Apr. 8–11, 1991.

[6] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Trans. on Programming Languages and Systems* 16(3): 311–321, May 1994.

[7] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, Jan. 1981.

[8] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. on Programming Languages and Systems* 12(4): 501–536, Oct. 1990.

[9] *Advanced Topics in Dataflow Computing and Multithreading*. G. R. Gao, L. Bic and J-L. Gaudiot. (Editors) IEEE Computer Society Press, 1995.

[10] L. George and A. W. Appel. Iterated register coalescing. In *Conf. Record of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 208–218, St. Petersburg, FL, Jan. 21–24, 1996.

[11] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proc. of the SIGPLAN '86 Symp. on Compiler Construction*, pages 11–16, Palo Alto, CA, June 25–27, 1986.

[12] J. R. Goodman and W-C. Hsu. Code scheduling and register allocation in large basic blocks. In *Conf. Proc., 1988 Intl. Conf. on Supercomputing*, pages 442–452, St. Malo, France, July 4–8, 1988.

[13] R. Govindarajan, C. Zhang, and G.R. Gao. Minimum register instruction scheduling: A new approach for dynamic instruction issue processors. In *Proc. of the Twelfth International Workshop on Languages and Compilers for Parallel Computing*, San Diego, CA, Aug. 1999. (available at `http://csa.iisc.ernet.in/~govind/lcpc99.ps`)

[14] R. Govindarajan, H. Yang, J.N. Amaral, C. Zhang, and G.R. Gao. Minimum register instruction sequence problem: Revisiting optimal code generation for DAGs Technical Memo #36, Computer Architecture and Programming Systems Lab., University of Delaware, Newark, DE, 1999.

[15] S. Mantripragada, S. Jain, and J.ehnert. A New Framework for Integrated Global Local Scheduling In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp. 167-174, 1998.

[16] Madhavi G. Valluri and R. Govindarajan. Evaluating register allocation and instruction scheduling techniques in out-of-order issue processors. In *Proc. of the Conf. on Parallel Architectures and Compilation Techniques, PACT '99*, Newport Beach, CA, Oct. 1999.

[17] R. Motwani, K.V. Palem, V. Sarkar, and S. Reyan. Combining register allocation and instruction scheduling. Technical Report, Courant Institute of Mathematical Sciences, New York University, New York, NY, 1996.

[18] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1997.

[19] B. Natarajan and M. Schlansker. Spill-free parallel scheduling of basic blocks. In *Proc. of the 28th Ann. Intl. Symp. on Microarchitecture*, pages 119–124, Ann Arbor, MI, Nov. 29–Dec.1, 1995.

[20] C. Norris and L. L. Pollock. Register allocation over the Program Dependence Graph. In *Proc. of the ACM SIGPLAN '94 Conf. on Programming Language Design and Implementation*, pages 266–277, Orlando, FL, June 20–24, 1994.

[21] S. S. Pinter. Register allocation with instruction scheduling: A new approach. In *Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 248–257, Albuquerque, NM, June 23–25, 1993.

[22] M. Poletto and V. Sarkar Linear scan register allocation *ACM Trans. of Programming Langauges and Systems*, 1998.

[23] T.A. Proebsting and C.N. Fischer Linear-time, optimal code scheduling for delayed-load architectures. In *Proc. of the ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation*, pages 256–2671, Toronto, Canada, June 1991.

[24] R. Sethi. Complete register allocation problems. *SIAM Jl. on Computing*, 4(3):226–248, Sep. 1975.

[25] R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *Jl. of the ACM*, 17(4):715–728, Oct. 1970.

[26] R. Silvera, J. Wang, G. R. Gao, and R. Govindarajan. A register pressure sensitive instruction scheduler for dynamic issue processors. In *Proc. of the Conf. on Parallel Architectures and Compilation Techniques, PACT '97*, pages 78–89, San Francisco, CA, Nov. 1997.

[27] J.E. Smith and G. Sohi. The microarchitecture of superscalar processors. *Proc. of the IEEE*, 83(12):1609–1624, Dec. 1995.

[28] O. Traub, G. Holloway, and M.D. Smith. Quality and speed in linear-scan register allocation. In *Proc. of the ACM SIGPLAN '98 Conf. on Programming Language Design and Implementation*, pages 142–151, Montreal, Canada, June 1998.

[29] H. S. Warren, Jr. Instruction scheduling for the IBM RISC System/6000 processor. *IBM Jl. of Research and Development*, 34(1):85–92, Jan. 1990.

[30] W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs and C. M. Geschke. *The Design of an Optimizing Compiler* American Elsevier Publishing Co., New York, NY, 1975.