Minimum Register Instruction Sequencing to Reduce Register Spills in Out-of-Order Issue Superscalar Architectures

R. Govindarajan, Hongbo Yang, José Nelson Amaral, Chihong Zhang, Guang R. Gao*

Abstract

In this paper we address the problem of generating an optimal instruction sequence S for a Directed Acyclic Graph (DAG), where S is optimal when it uses a minimum number of registers. We call this the Minimum Register Instruction Sequence (MRIS) problem. The motivation for studying the MRIS problem stems from several modern architecture innovations/requirements that put the instruction sequencing problem in a new context.

We develop an efficient heuristic solution for the MRIS problem. This solution is based on the notion of an *instruction lineage* — a set of instructions that can definitely share a single register. The formation of lineages exploits the structure of the dependence graph to facilitate the sharing of registers not only among instructions within a lineage, but also across lineages. Our efficient heuristics to "fuse" lineages further reduce the register requirement. This reduced register requirement results in generating code sequence with fewer register spills.

We implemented our solution in the MIPSpro production compiler and measured the performance on the SPEC95 floating point benchmark suite. Our experimental results demonstrate that the proposed instruction sequencing method significantly reduces the number of spill loads and stores inserted in the code, by more than 50% in each of the benchmarks. Our approach reduces the average number of dynamic loads and stores executed by 10.4% and 3.7%, respectively. Further, our approach improves the execution time of the benchmarks on the average by 3.2%.

In order to evaluate how efficiently our heuristics find a near-optimal solution to the MRIS problem, we develop an elegant integer linear programming formulation for the MRIS problem.

*R. Govindarajan is with the Supercomputer Education & Research Centre, Dept. of Computer Science & Automation, Indian Institute of Science, Bangalore, India. H. Yang and G. R. Gao are with the Electrical and Computer Engineering, University of Delaware, Newark, DE, USA. J. N. Amaral is with the Dept. of Computing Science, University of Alberta, Edmonton, AB, Canada. C. Zhang is with Conexant Systems, Inc., Newport Beach, CA, USA.

Using a commercial integer linear programming solver we obtain the optimal solution for the MRIS problem. Comparing the optimal solution from the integer linear programming tool with our heuristic solution reveals that in a very large majority (99.2%) of the cases our heuristic solution is optimal. For this experiment, we used a set of 675 dependence graphs representing basic blocks extracted from scientific benchmark programs.

Keywords

Compiler Optimization, Code Sequence Optimization, Register Allocation, Instruction Scheduling, Code Generation, Superscalar architectures, Instruction Level Parallelism.

1 Introduction

In this paper we revisit the *optimal code generation* problem [1, 9] also known as the *evaluation-order determination* problem [42]: 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 uses the minimum number of registers. We define the *Minimum Register Instruction Sequence* (MRIS) problem as:

> 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.

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 [37]. Out-of-order (o-o-o) instruction issue is facilitated by the runtime scheduling hardware and by the register renaming mechanism existent in superscalar architectures. An o-o-o processor has more physical (*i.e.,* not logical or architected) registers at its disposal for register renaming at runtime, that are not visible to the compiler. Due to these hardware mechanisms, o-o-o issue processors have the capabilities to uncover the instruction level parallelism obscured by anti- and output-dependences (together known as *false dependences* or *name dependences*). Hence it is important in these processors to reduce the number of register spills, even at the expense of not exposing instruction level parallelism [40, 36]. Reducing register spills reduces the number of loads and stores executed, which in turn is important:

- from a performance viewpoint in architectures that either have a small cache or a large cache miss penalty;

- from a memory bandwidth usage viewpoint as the elimination of spill instructions frees instruction slots to issue other useful instructions;

- from a power dissipation viewpoint, as load and store instructions consume a significant portion of the power budget.

As another argument to why the MRIS problem is relevant, in code generation for threads in fine-grain multi-threaded architectures, it is often important to minimize the number of registers used in a thread in order to reduce the cost of thread context switch [14].

The MRIS problem is related to, but different from, the conventional instruction scheduling [1, 16, 17, 28, 41] and register allocation [1, 7, 8, 10, 11, 15, 28, 30, 32, 39] problems. In traditional instruction scheduling, the main objective is to minimize the total time (or length) of the schedule. Thus such problem formulation must take into account the execution latencies of each instruction in the DDG. In contrast, the latency of operations in the DDG and the availability of functional unit resources are not a part of the MRIS problem formulation. To highlight this difference, we distinguish the use of the terms "*instruction schedule*" and "*instruction sequence*".

The MRIS problem is closely related to the *optimal code generation (OCG) problem* [1, 9, 34] or the *evaluation-order determination* problem [42]. For the case in which the dependence graph is a tree, an algorithm that produces an optimal sequence (in terms of code length) for the OCG problem exists. For a general DAG, the problem is known to be NP-Complete since 1975 [34]. An important difference between these traditional code generation approaches (OCG and evaluation order determination) and our MRIS problem is that the former emphasize reducing the code schedule length while the latter focuses on minimizing the number of registers used.

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 an *instruction lineage* evolves from the notion of *instruction chains* [19] which allows the sharing of a register among instructions along a (flow) dependence chain in a DDG. In other words, a lineage is a set of nodes in the DDG. Instruction lineages model the DDG register requirement more accurately than instruction chains (see Section 6).

- *Lineage Interference Graph:* The notion of a lineage interference graph captures the *definite overlap* relation between the live ranges of lineages even before the instructions in lineages are scheduled, and its use to facilitate sharing of registers across lineages.

We implemented our heuristic approach (MRIS) in the SGI MIPSpro compiler suite and used that implementation to evaluate our approach on the SPEC95 floating point benchmarks. For comparison we also measured the performance of a baseline version of the compiler that performs traditional compiler optimizations, but that does not optimize the instruction sequence with the goal of reducing register pressure. We also compared our approach with an Optimized version of the MIPSpro compiler that performs integrated instruction scheduling and register allocation. As the emphasis of our work is on sequencing the instructions to reduce the register requirements and the number of spill instructions executed, we measured the number of static as well as dynamic loads and stores in each benchmark under different versions of the compiler. We also compare the execution time of the benchmarks. Our experimental results are summarized as follows.

- When compared to the baseline version of the MIPSpro compiler, the heuristic approach significantly reduces the number of spill instructions inserted in the (static) code, on the average by 63.14%, and by more than 50% in each of the benchmarks. The heuristic approach also drastically reduces the number of basic blocks that require spills.

- The number of dynamic load and store instructions executed reduces respectively, by as much as 20.9% and 11.2%, and on the average by 10.4% and 3.7%, in comparison to the baseline version of the compiler.

- The heuristic method also marginally improves the execution time, on the average by 3.2%. In one of the applications, the performance improvement is as high as 14.2%.

- Lastly, even compared to the optimized version, our heuristic approach performs better in terms of all the above parameters, although the percentage improvement is relatively low.

The baseline version of the compiler performs the same optimizations included in the optimized compiler, including the same instruction scheduling and register allocation algorithms. In the optimized compiler, when the local register allocation requires spilling, the local instruction scheduler is called again using more accurate estimations for the register pressure. This second attempt at instruction scheduling optimization, after the first attempt at register allocation, is disabled in the baseline compiler.

In order to assess the optimality and efficiency of the proposed heuristic we formulate the MRIS problem as an integer linear programming problem and solve it using a commercial integer linear programming solver. This implementation produces the optimal solution for the MRIS problem for DDGs with a small number of nodes. Comparing this optimal solutions with the heuristic ones reveals that, for a very large majority (99.2%) of the cases, our heuristic solution was optimal. For this experiment, we used a set of 675 (small sized) DDGs representing basic blocks in scientific benchmarks programs.

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 for the lineage formation and a sequencing method to construct a near-optimal minimum register instruction sequence[1]. Section 4 deals with the formulation of the MRIS problem as an integer linear programming problem. We report static and dynamic performance measures of our approach in Section 5. Related work and conclusions are presented in Sections 6 and 7.

2 Motivating Example

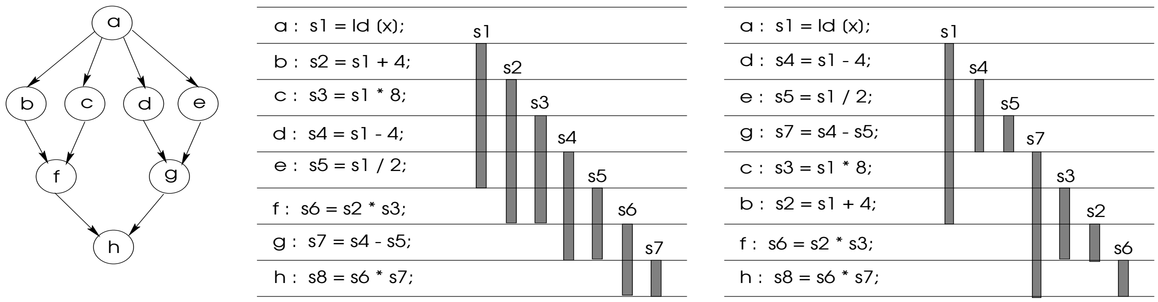
We first use an example to motivate the MRIS problem. Later we illustrate our heuristic approach using the same example.

2.1 Motivating Example

Consider the computation represented by a data dependence graph (DDG)[2] 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 in this example 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 one of the minimum register sequences.

[1] We use the term *near-optimal* to indicate that in our empirical investigation, our heuristic algorithm compared well with known optimal solutions. We make no claims about near-optimality from an approximation theory stand point.

[2] Since our focus in this paper is on generating an instruction sequence that reduces register pressure, 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.



(a) DDG (b) Instruction Sequence 1 (c) Instruction Sequence 2

Figure 1: Motivating Example

The MRIS problem can be stated as follows: given a set of instructions and the data dependences among them, build an instruction sequence that requires a minimum number of registers. The input for the MRIS problem is a Data Dependence Graph (DDG) where the nodes represent instructions and the directed edges, also referred to as flow arcs, are data dependences.[3] The edges of the DDG impose a partial order among the instructions. In this paper we restrict our attention to acyclic DDGs and hence do not consider any loop-carried dependences.

We say that multiple instructions share a single register if they use the same register as a destination for the values that they produce. We need to identify which nodes in the DDG can share the same register in a legal sequence. Although a complete answer to this question is hard to determine, the data dependences in the DDG 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 use 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 node f. Similarly nodes e and g can share the same register. Next, can nodes f and g share the same register? The answer is no, because, in any legal sequence, the values produced by these instructions must be live simultaneously so that the computation in node 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 it 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 nodes c and d do not overlap, and hence they can share the same register. In the following subsection, we outline our approach which uses some efficient heuristics to arrive at a near-optimal solution to the MRIS problem.

[3]Although we present the MRIS formulation for the DDG of a basic block, our method is also applicable to superblocks [23].

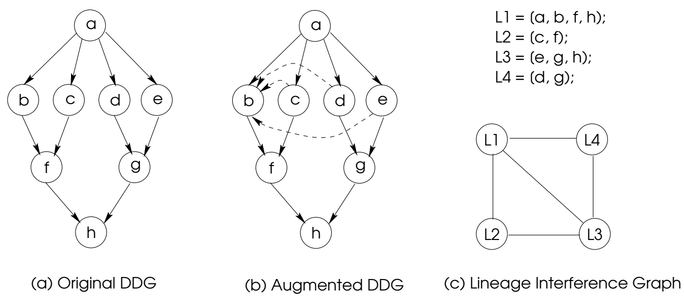


Figure 2: Data Dependence Graph for the Motivating Example

2.2 Overview of the Lineage Based Algorithm

Our solution to the MRIS problem uses the notion of *instruction lineages*. If S_i is a node in a DDG and S_i has one or more descendents, then S_i produces a value and must be assigned a register.[4] If we have a sequence of instructions in the DDG $S_1, S_2, S_3, \cdots, S_n$ where S_2 is the descendant of S_1, S_3 is the descendant of S_2, *etc.,* then we can form a *lineage* of these instructions in such a way that an instruction sequencing in which all the instructions in the lineage share the same register can be generated. 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, and, hence, if the live range of the variable defined by S_1 ends at S_2, then S_2 can definitely reuse the same register allocated to S_1.

What if S_1 has more than one descendant? 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 sequenced before S_2. Thus the selection of one of the descendents of S_1 to be the *heir* of the register creates sequencing constraints among the descendents of S_1. Such sequencing constraints are explicitly represented in the *augmented DDG* by means of directed *sequencing edges* from each descendant 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. These sequencing edges are shown as dotted arrows in Figure 2(b). In Section 3 we introduce a simple but efficient heuristic to select heirs. We will also show that the introduction of sequencing edges does not introduce cycles in the DDG.

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

[4]A node of the DDG without descendents must either be a store instruction or else it produces a value that is live-out of the basic block. For simplicity, we do not consider live-out registers in the presentation of our solution. A simple extension of our solution would insert a dummy sink node to capture live-out registers as in [27].

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 sequence the lineages in such a way that they share a register. Based on the overlap relation we construct a Lineage Interference Graph (LIG). Figure 2(c) shows the LIG for our example. This lineage interference graph is colored using traditional graph coloring algorithms to compute the number of registers required for the DDG [10, 11]. We refer to this number as the *Heuristic Register Bound (HRB)*. Once we color the lineage interference graph, we apply a sequencing 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.

As formulated, MRIS optimizes an instruction sequence based only on the number of registers used in the sequence. MRIS does not takes into consideration the size of the instruction window in a superscalar processor. Therefore an optimal solution to MRIS may be an instruction sequence that minimizes the register pressure, but result in sub-optimal schedule at runtime because instructions that could execute in parallel are not in the same instruction windows.

3 Heuristic Approach to the MRIS Problem

In this section, we present our heuristic approach to find a good approximation to the minimum register sequence for an acyclic DDG. First, we formally introduce the concept of lineage and describe our lineage formation algorithm. Further, we establish a condition for lineage overlapping, introduce the lineage interference graph (LIG) and compute the HRB. In Section 3.3, we introduce the concept of lineage fusion to simplify the LIG and improve register sharing. Finally we describe the sequencing algorithm for instruction sequence generation based on the coloring of the LIG.

3.1 Lineage Formation

We use the notion of *instruction lineage* to identify sequences of operations that can share registers.

Definition 3.1 *An* **instruction lineage** $L_v = [v_1, v_2, \cdots, v_n)$ *is a set of nodes such that there exist flow arcs* (v_1, v_2), (v_2, v_3), …, (v_{n-1}, v_n) *in the DDG. We say that* v_2 *is the* **heir** *of* v_1 *in lineage* L_v, v_3 *is the heir of* v_2 *in* L_v, *and so on.*

Although a node can have many descendants in the DDG, only one of these descendants can be the node's heir. The heir of a node v_i is a node v_j that inherits the destination register of v_i to store its own result. As a consequence the formation of a lineage imposes sequencing constraints among the descendants of a node v_i: the chosen heir v_j must be the last among all descendants of v_i to be executed because v_j is the *last use* of the value produced by v_i.

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 might be a store instruction that does not need a register. Therefore the last node does not use the same register as the other nodes in the lineage. We emphasize this property of the last node by denoting a lineage with semi-open intervals as $[v_1, v_2, \cdots, v_{n-1}, v_n)$. Thus, in the DDG of Figure 2, the four lineages that cover all the def-last_use relations are $[a, b, f, h)$, $[c, f)$, $[e, g, h)$, and $[d, g)$.

Our lineage formation algorithm attempts to form as long a lineage as possible, in the sense that if $\{v_1, v_2, \cdots, v_n\}$ is a lineage then either v_n is a node with no descendants or v_n is already associated with some other lineage. Because the heir is always the last descendant to be executed in an acyclic sequencing, 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. In order to ensure that the heir is the last use of the value produced by its parent node, we introduce *sequencing edges* in the DDG, from each descendant of a node to the chosen heir, as shown in Figure 2(b). A sequencing edge from a node v_i to a node v_j imposes the constraint that node v_i must be listed before node v_j is listed. This constraint implies that all nodes that can reach v_i must be listed before any node that can be reached from v_j is listed.

If the introduction of sequencing edges were to make the graph cyclic, then it would be impossible to obtain a sequence for 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:

$$ht(u) \quad = \quad \begin{cases} 1 & \text{if } u \text{ has no descendants} \\ 1 + \max_{v \in D(u)}(ht(v)) & \text{otherwise} \end{cases}$$

where $D(u)$ is the set of all the immediate descendants of u. In the DDG of Figure 2(a) in Section 2, the heights of the nodes 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 descendants, then we chose a descendant node with the smallest height to be the heir of v_i. If multiple descendants have the same lowest height, then the tie is broken arbitrarily. In order to ensure that cycles are not introduced in the lineage formation process, we recompute the height of each node after introducing sequencing edges between the descendants of a node.

Each flow arc (u, v) in a DDG corresponds to a true data dependency, and hence, to a definition-use (def-use) relationship between the nodes u and v. Hence each dependence edge is associated with a register. Later we will assign registers to lineages of nodes of the DDG. Therefore it is important that the live range of each node of the DDG (except for sink nodes) be associated with exactly one lineage. The lineages are formed by the arcs between a node and its chosen heir, *i.e.*, each arc that form a lineage is a def-*last_use* arc in the DDG. By using a greedy algorithm to form lineages, our algorithm is conservative in the number of lineages formed, thus reducing the size of the *lineage interference graph* and the complexity of coloring that graph.

The Lineage Formation algorithm is essentially a depth-first graph traversal greedy algorithm by identifying a heir for each node using the height priority. If a node has multiple descendants, sequencing edges are introduced after the heir is selected, and the heights of all nodes in the DDG are recomputed. The detailed algorithm is presented in Figure 3. The application of the lineage formation algorithm to the DDG of Figure 2(a) results in four lineages as shown in Figure 2(c). We refer to the DDG with the additional sequencing edges as the *augmented DDG* (refer to Figure 2(b)). Notice that in steps 8 and 10 of the algorithm, we distinguish flow edges from sequencing edges. That is, if there is a sequencing edge from node v_i to node v_j, v_j is not considered a descendant of v_i for the purpose of lineage formation. Only arcs that represent data dependences are considered for the lineage formation.

Next we show that the introduction of sequencing edges does not introduce any cycle. Formally,

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

Proof: Since only the lowest descendant of a node is chosen as the heir, all sequencing edges inserted will be from nodes with higher heights to nodes with lower heights. Also, if the lowest descendant is already in a different lineage, then the current lineage ends. Furthermore, the heights of all the nodes in the graph are recomputed (in Steps 23 and 24) at the start of each new lineage[5]. Thus in the augmented

[5]Although one may think that the re-computation of the heights of all nodes is needed every time after choosing a heir in a lineage, the heights of descendants of the chosen heir will not be affected by the sequencing edge. Hence the re-computation of heights is performed, if necessary, when a new lineage is started.

```
LineageFormation(V, E)
mark all nodes in the DDG as not in any lineage
compute the height of every node in the DDG
while there is a node not in any lineage do
        recompute height ← false
        v_i ← highest node not in lineage
        start a new lineage containing v_i
        mark v_i as in a lineage
        while v_i has a descendant do
                v_j ← lowest descendant of v_i
                if v_i has multiple descendants
                    recompute height ← true
                    for each descendant v_k ≠ v_j of v_i do
                         add sequencing edge from v_k to v_j
                    endfor
                endif
                add v_j to lineage
                if v_j is already marked as in a lineage
                    // end lineage with v_j as the last node
                    break;
                endif
                mark v_j as in a lineage
                v_i ← v_j
        end while
        if recompute height = true
                recompute the height of every node in the DDG
        endif
end while
```

Figure 3: Lineage Formation Algorithm.

DDG, there can be no path from a lower to a higher node, and therefore no cycle can be formed. □

3.2 Lineage Interference Graph and Heuristic Register Bound

In this section we discuss how to determine whether the live ranges of two lineages always overlap and how to compute the heuristic register bound.

3.2.1 Overlapping of Live Ranges

In order to determine whether two lineages can share a register, we need to verify if the live ranges of these lineages overlap. To define 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 *If the first instruction of a lineage $L = [v_1, v_2, \cdots, v_n)$, v_1 is in position t_i, and the last instruction v_n is in position t_j in the instruction sequence, then the* **live range** *of the lineage L starts at t_{i+1} and ends at t_j.*

Our goal is to find an instruction sequence that results in a minimal register usage over all legal instruction sequences. Therefore we must identify lineages that can share the same register, *i.e.* lineages with non-overlapping live ranges. Moreover we must identify such lineages before a complete instruction sequence is produced. While we are identifying these lineages, the instructions within each lineage are totally ordered, but instructions from multiple lineages will be interleaved in a yet to be determined way to form the final instruction sequence. Thus we can think that the position of instructions in the sequence, and as a consequence, the live ranges of different lineages are *floating*. The live range of a lineage is always contiguous, irrespective of interleavings of instructions from multiple lineages. Thus once the first instruction in a lineage is listed, its live range is active until the last instruction of the lineage is listed into the sequence.

In order to determine whether the live ranges of two lineages must necessarily overlap, we define a condition based on the existence of paths between the lineages. First, we define two set of nodes: S is the set of nodes that start lineages, and E is the set of nodes that end lineages. Next we define the *reach* relation R.

Definition 3.3 *The reach relation $R : S \to E$ maps S to E. For all $v_a \in S$ and $v_b \in E$, node v_a reaches node v_b, $R(v_a, v_b) = 1$, if there is a path in the augmented DDG from v_a to v_b, otherwise $R(v_a, v_b) = 0$.*

The reach relation is used to determine whether the live ranges of two lineages must necessarily overlap in *all* legal instruction sequences for the augmented DDG.

Definition 3.4 *Let t_{s_i} represent the position of instruction s_i in an instruction sequence. Two lineages $L_u = [u_1, u_2, \cdots, u_m)$ and $L_v = [v_1, v_2, \cdots, v_n)$* **overlap** *in an instruction sequence if $t_{u_1} < t_{v_1} < t_{u_m}$ or $t_{v_1} < t_{u_1} < t_{v_n}$.*

Definition 3.5 *Two lineages* $L_u = [u_1, u_2, \cdots, u_m)$ *and* $L_v = [v_1, v_2, \cdots, v_n)$ **definitely overlap** *if they overlap for all possible instruction sequences.*

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* u_1 *reaches* v_n *and* v_1 *reaches* u_m*.*

Proof: Since there exists a directed path (consisting of flow and sequencing edges) from u_1 to v_n, in any instruction sequence u_1 must be listed before v_n, $t_{u_1} < t_{v_n}$. Similarly, since there exist a directed path from v_1 to u_n, $t_{v_1} < t_{u_m}$. Further since L_u is a lineage, $t_{u_1} < t_{u_m}$ and similarly $t_{v_1} < t_{v_n}$. If u_1 is listed before v_1, $t_{u_1} < t_{v_1}$, we obtain that $t_{u_1} < t_{v_1} < t_{u_m}$ and the lineages overlap. Likewise if v_1 is listed before u_1, $t_{v_1} < t_{u_1}$, we obtain $t_{v_1} < t_{u_1} < t_{v_n}$ and the two lineages also overlap.Thus we proved that whether the L_u starts before L_v, or L_v starts before L_u, the two lineages overlap. Hence the live ranges of L_u and L_v definitely overlap. □

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 $a \to d \to g$, and node d can reach node h through the path $d \to g \to h$. Therefore, the live ranges of these two lineages must overlap. Similarly the live range of lineage L_2 must overlap with L_1, L_3 must overlap with L_4, L_1 must overlap with L_3, and L_2 must overlap with L_3.

3.2.2 Constructing and Coloring the Lineage Interference Graph

Next we construct a lineage interference graph (LIG), an undirected graph whose vertices's represent the lineages. Two vertices are connected by an interference edge if and only if the live ranges of the lineages represented by them definitely overlap. Using Theorem 3.1, the lineage interference graph can be computed from the transitive closure of the augmented DDG. Fortunately the reach relation defined earlier significantly reduces the cost of generating the LIG by requiring only the verification of the paths between nodes that start and nodes that end lineages.

There is an edge in the LIG between two lineages $L_u = [u_1, u_2, \cdots, u_m)$ and $L_v = [v_1, v_2, \cdots, v_n)$ if and only if $R(u_1, v_n) = 1$ and $R(v_1, u_n) = 1$. In our motivating example the set of start nodes is $S = \{a, c, d, e\}$, and the set of end nodes is $E = \{f, g, h\}$. For the presentation of our algorithm, the reach relation can be represented by a binary matrix where each row is associated with a node in S and each column is associated with a node in E. The reach relation for this example is shown in Figure 4(a). The LIG for this example is shown in Figure 2(c).

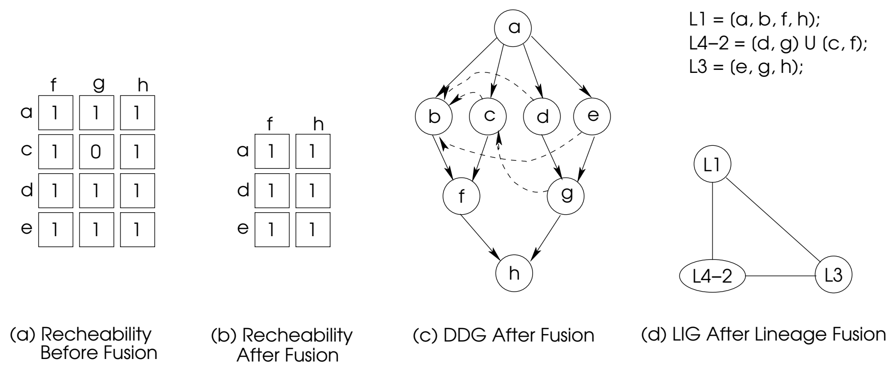


Figure 4: Reach Relation. DDG and LIG after lineage fusion for Motivating Example.

The lineage interference graph can be colored using a heuristic graph coloring algorithm [10, 11]. We refer to the number of colors required to color the interference graph as the Heuristic Register Bound (HRB). It should be noted that due to the heuristics involved in coloring the interference graph and due to the sequencing order of descendant nodes in the DDG, the HRB computed is a near-optimal solution. For our motivating example, shown in Figure 1(a), the lineage interference graph is depicted in Figure 2(c). This interference graph can be colored using 3 colors resulting in a HRB of 3 for our motivating example.

3.3 Lineage Fusion

Before we proceed to describe our instruction sequence generation method, in this section, we present an optimization that *fuses* two lineages into a single one. Lineage fusion increases the success of the instruction sequence generation algorithm on finding an instruction sequence that requires no more than HRB registers.

3.3.1 Conditions for Lineage Fusion

Let $L_u = [u_1, u_2, \cdots, u_m)$ and $L_v = [v_1, v_2, \cdots, v_n)$. If u_1 reaches v_n but v_1 does not reach u_m, then the live ranges of L_u and L_v do not necessarily interfere with each other, *i.e.,* it is possible to generate a legal sequence where these two live ranges do not overlap. In this case, it is possible to use the same register for both lineages. However if we start listing nodes of L_v in the instruction sequence before we finish listing all the nodes of L_u, the two lineages will interfere and we will not be able to use the same register for both lineages. An instruction sequencer with a fixed number of registers will deadlock in such a situation because it will not be able to release the register used for nodes of L_u for usage by L_v. This lineage interference, and consequent sequencing deadlock, is avoidable.

A simple solution that prevents this interference and the consequent sequencing deadlock involves introducing a new sequencing constraint in the DDG that forces all the nodes of L_u to be listed before any node of L_v is listed. We call this operation a *lineage fusion* because it treats lineages L_u and L_v as if they were a single lineage. An additional advantage of lineage fusion is a reduction in the number of nodes in the lineage interference graph and a consequent reduction in the cost of coloring that graph. After the lineages are formed as described in Section 3.1, pairs of lineages that satisfy the *Lineage Fusion Condition* can be fused into a single lineage. Formally,

Definition 3.6 *Two instruction 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. u_1 reaches v_n, i.e., $R(u_1, v_n) = 1$;

ii. v_1 does not reach u_m, i.e., $R(v_1, u_m) = 0$;

When the lineages L_u and L_v are fused together, a sequencing edge from u_m to v_1 is inserted in the DDG, the lineages L_u and L_v are removed from the lineage set and a new lineage $L_w = [u_1, u_2, \ldots, u_m) \cup [v_1, v_2, \ldots, v_n)$ is inserted in the lineage set. Note that the last node of the first lineage, u_m, does not necessarily use the same registers as the other nodes in the new L_w lineage. Thus it is important to represent the lineage resulting from the fusion as a union of semi-open sequences as we did in this paragraph. 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. In fusing lineages, the reach relation should be updated after each fusion. We discuss how to update the reach relation in the following subsection. Before that we establish that the sequencing edges introduced by lineage fusion do not result in cycles.

Lemma 3.2 *The fusion of lineages does not introduce any cycle in the augmented DDG.*

Proof: We use proof by contradiction. Let $L_u = [u_1, u_2, \ldots, u_m)$ and $L_v = [v_1, v_2, \ldots, v_n)$ be two lineages such that (1) u_1 reaches v_n, and (2) v_1 does not reach u_m. Let us assume that when we fuse L_u and L_v, the inclusion of the sequencing edge from u_m to v_1 causes the formation of a cycle in the DDG. If this edge forms a cycle, this cycle must include a path from v_1 to u_m. But that contradicts condition 2 for fusion. Hence, the fusion of two lineages under the conditions of definition 3.6 cannot introduce a cycle in the DDG. □

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. It would be also legal to fuse the two lineages L_u and L_v when $R(v_1, u_m) = R(u_1, v_n) = 0$. However such fusions would impose an unnecessary constraint in the sequencing order of L_u and L_v restricting the freedom of the sequencing algorithm. The sharing of registers by completely independent lineages is indicated by the coloring of the LIG.

3.3.2 Implementing Lineage Fusion

While it may seem at first that it would be necessary to recompute the transitive closure of the augmented DDG after each lineage fusion, we can obtain all possible lineage fusions from the reach relation already computed for the construction of the lineage interference graph.

To find candidates for fusion, we search for a pair of lineages $L_u = [u_1, u_2, \cdots, u_m)$ and $L_v = [v_1, v_2, \cdots, v_n)$ such that $R(u_1, v_n) = 1$ and $R(v_1, u_m) = 0$. The fusion condition says that we can create the new lineage $L_w = L_u \cup L_v$. The creation of this new lineage will require the addition of a scheduling edge from u_m to v_1. The algorithm updates the reach relation in such a way that all nodes that could reach u_m before the fusion now can also reach v_n. In other words, $R(u_1, v_b) = 1$, for some $v_b \in E$, if either $R(u_1, v_b) = 1$ before the lineage fusion, or $R(v_1, v_b) = 1$. After the fusion v_1 can no longer be in the set of nodes that start lineages, S. Notice that v_1 is no longer the start node of the fused lineage L_w, and that each node can start at most one lineage in the lineage formation algorithm. Therefore the row corresponding to v_1 should be eliminated from the reach relation. Furthermore, if u_m does not terminate any other lineage, its column can also be eliminated from the reach relation.[6] Further, node v_1 is removed from S. Likewise, node u_m can be removed from E, the set of nodes that end a lineage, if u_m does not end any other lineage other than L_u.

When there are multiple candidates for lineage fusion, we arbitrarily select a pair of lineages to be fused. The fusion results in the updating of the sets S and E, and of the reach relation. We keep

[6] A node might end multiple lineages because the last node of the lineage does not share a register with the other nodes in the lineage. For instance, in our motivating example node h ends lineages $L1$ and $L3$.

searching the reach relation representation for new pairs of lineages that can be fused until none is found. For instance, a lineage L_u may be first fused with a lineage L_v to create a new lineage $L_u \cup L_v$, and then this compound lineage may be fused with a third lineage L_w to form $L_u \cup L_v \cup L_w$.

In our motivating example the initial analysis of the reach relation finds that lineage $L4 = [d, g)$ can be merged with lineage $L2 = [c, f)$ because $R(d, f) = 1$ and $R(c, g) = 0$. The new lineage is $L4\text{–}2 = [d, g) \cup [c, f)$ and the new sequencing edge in the DDG is (g, c). The updated reach relation is shown in Figure 4(b). Because there are no more zeros in the reach relation, no more lineage fusions are possible. The DDG and the LIG after fusion are shown in Figures 4(c) and 4(d).

In the following subsection, we present an heuristic method to derive an instruction sequence for the augmented DDG.

3.4 Instruction Sequence Generation

The coloring of the lineage interference graph associates a register with each lineage. For example, a register assignment A with 3 colors for the nodes in our motivating example is:

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

where R_1, R_2, and R_3 are general purpose registers. Our solution assumes that registers that are live-in and live-out in the DDG will be assigned by a global register allocation procedure. However, as discussed in Section 2, our method can account for live-in and live-out registers through the addition of dummy *source* and *sink* nodes.

Our sequencing method is based on the list scheduling 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 instruction sequence generation method is shown in Figure 5. The sequencing algorithm takes as inputs G', the DDG augmented with sequencing edges; L, a list of lineages obtained from the lineage formation algorithm with lineage fusion applied; and A, the register assignment for the nodes from the coloring of the lineage interference graph. The availability of register needs to be checked only if the node v_i is the first node in its lineage. Otherwise, the predecessor of v_i will pass the register assigned to it, which will be free when v_i is ready to be listed. The algorithm uses the augmented DDG with the sequencing edges, which ensures that v_i is the *last use* of its predecessor. When two

SEQUENCING(G', L, A)
ReadyList $\leftarrow \{(v_i, R_j)$ such that v_i has no predecessors $\}$
RegAvailable $\leftarrow \{R_1, R_2, \ldots, R_N\}$
while ReadyList $\neq \emptyset$ **do**
  for each node v_i in the ReadyList in decreasing height order **do**
    if ($v_i \notin S$) **or** ((($v_i, R_j) \in A$) **and** ($R_j \in$ RegAvailable))
      // either v_i is not the start node of a lineage or
      // the register assigned to v_i in A is available
      Remove R_j from RegAvailable
      Remove (v_i, R_j) from ReadyList
      List(v_i, R_j)
      Add to the ReadyList all successors of v_i
        that have all its predecessors listed
      if ($v_i \in E$) **and** $(v_i, R_j) \in A$
        // node v_i ends a lineage which
        // is assigned register R_j
        Return R_j to RegAvailable
      endif
    endif
  endfor
end while

Figure 5: Sequencing Algorithm.

lineages $L_u = [u_1, u_2, \cdots, u_m)$ and $L_v = [v_1, v_2, \cdots, v_n)$ are fused together, they are represented by a single lineage in the lineage interference graph and a single register is assigned to the nodes $u_1, u_2, \cdots, u_{m-1}, v_1, \cdots, v_{n-1}$. Because the fusion algorithm introduced a sequencing edge from u_m to v_1, and removed nodes u_m and v_1 from E and S respectively, there is no need to check for the availability of a register when v_1 is listed.

Unfortunately, the above sequencing algorithm could result in a deadlock due to two reasons [19]. First, the order of listing 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*, as they often can be avoided through the lineage fusion process. We illustrate this with the help of an example in the following discussion. The second kind of deadlocks referred to as *unavoidable* deadlocks are caused due to the underestimation of HRB. This could happen because the condition used to test if two live ranges definitely overlap (stated in Theorem 3.1) is sufficient but not necessary. As illustrated below, lineage fusion helps to reduce the occurrences of both avoidable and unavoidable deadlocks.

Applying the sequencing algorithm to our motivating example, first we list node a. If lineage fusion is not used, the listing 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 listed before node f (listing node f will release the register R_2 currently used by c to d). On the other hand, in order to list f we must first list b, but b cannot be listed 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 sequencing 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 have to list node e. The nodes g, c, b, f, and h are listed subsequently in that order. The 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 heuristic register bound (HRB) computed from 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. To overcome the deadlock problem (both avoidable and unavoidable deadlocks) we follow a simple heuristic that increases the HRB by 1. The algorithm then picks one of the nodes (the one with the maximum height) in the Ready List and change its register assignment (as well as that of the remaining nodes in that lineage) to a new register and list the node. This strategy overcomes a deadlock by gradually increasing the HRB and trying to obtain a sequence that uses as few extra register as possible. We measure the performance of our heuristic approach in Section 5.

4 Exact Approach to the MRIS Problem

In this section we formulate the MRIS problem as an integer linear programming (ILP) problem. Our ILP formulation must obtain an ordering of the DDG nodes such that each node u is assigned a unique integer f_u that represents its position in the instruction sequence. If there are n nodes in the DDG, then f_u can assume values from 1 to n. That is,

$$0 < f_u \leq n \tag{1}$$

Further since we are interested only in a sequential ordering of the nodes, no two nodes can have the same position, *i.e.,* $f_u \neq f_v$ for all $u \neq v$. We represent this relation as an integer linear constraint using the technique proposed by Hu [22]. We use a 0-1 integer variable $w_{u,v}$ and write the condition $f_u \neq f_v$ as a pair of constraints:

$$f_u - f_v > -n * w_{u,v} \tag{2}$$

$$f_v - f_u > -n * (1 - w_{u,v}) \tag{3}$$

where n is the number of nodes in the DDG. Intuitively $w_{u,v}$ represents the sign of $(f_v - f_u)$, and $w_{u,v}$ is 1 if the sign is positive and 0 otherwise.

Next we derive conditions that ensure that the instruction sequence obeys true data dependences. If there exists a flow dependence arc (u, v) in the DDG, then clearly node u must be listed ahead of node v. That is

$$f_v - f_u > 0 \quad \text{for all } (u, v) \text{ in the DDG} \tag{4}$$

Assume that node u has k successors: $v_1, v_2, \cdots, v_k$. The live range of the value produced by node u is the interval $[f_u + 1, \max(f_{v_1}, f_{v_2}, \cdots, f_{v_k})]$. Thus the value produced by node u is live at position t iff (i) $f_u + 1 \leq t$ and (ii) $(f_{v_1} \geq t) \vee (f_{v_2} \geq t) \vee \cdots \vee (f_{v_k} \geq t)$. For t ranging from 1 to n, the value of the 0-1 integer variable $d_{u,t}$ represents whether $f_u \leq t$ is true (see inequalities 5 and 6). Similarly, the value of the 0-1 integer variable $e_{i,u,t}$ indicates whether $f_{v_i} > t$ (see inequalities 7 and 8). The constraints in the position of all nodes that have true dependences on node u, *i.e.,* nodes $v_1, \cdots, v_k$ are represented in the similar way.

$$t - f_u \geq -n * (1 - d_{u,t}) \tag{5}$$

$$t - f_u < n * d_{u,t} \tag{6}$$

$$t - f_{v_i} \geq -n * e_{i,u,t} \tag{7}$$

$$t - f_{v_i} < n * (1 - e_{i,u,t}) \tag{8}$$

The 0-1 integer variable $e_{u,t}$ is 1 if and only if any of the $e_{1,u,t}, \cdots, e_{k,u,t}$ variables is 1. This condition is imposed by the following two constrains.

$$e_{1,u,t} + \cdots + e_{k,u,t} \geq e_{u,t} \tag{9}$$

$$e_{1,u,t} + \cdots + e_{k,u,t} \leq k * e_{u,t} \tag{10}$$

Finally, the 0-1 integer variable $s_{u,t}$ indicates if variable u is live at time t. Thus $s_{u,t}$ is 1 if and only if both $d_{u,t}$ and $e_{u,t}$ are nonzero. This relation is represented in the following two constraints.

$$d_{u,t} + e_{u,t} \geq 2 * s_{u,t} \tag{11}$$

$$d_{u,t} + e_{u,t} \quad \leq \quad 2 * s_{u,t} + 1 \tag{12}$$

The interference relations between the live ranges within a basic block form an acyclic interval graph. Therefore it is optimally colorable with the same number of colors as the maximum width of the interval graph [21]. The width of the interval graph at position t can be represented by $s_{1,t} + s_{2,t} + \cdots + s_{n,t}$. Hence the objective function is to **minimize** z where

$$z \quad \geq \quad \sum_{i=0}^{n} s_{i,t} \quad \text{for all } \; t \; \in \; [1, n] \tag{13}$$

Thus the ILP problem is to minimize z subject to inequalities 1 to 13.

5 Experimental Results

We present results from two sets of experiments designed to evaluate the performance of our lineage based algorithm. In the first set of experiments (see Section 5.1) we compare the register requirement of the instruction sequence generated by our lineage-based sequencing algorithm with the minimum register requirement computed by the ILP formulation presented in Section 4. Because of the time complexity of the ILP algorithm, the ILP method only can find a solution in a reasonable time for DDGs with a limited number of nodes (in spite of the use of an efficient commercial ILP solver, *viz.,* CPLEX). Therefore instead of SPEC benchmarks, we use a set of loops extracted from a collection of benchmark programs for this comparison. We were able to compare the results for 675 DDGs and determined that in 99.2% of them our sequencing method found a sequence that uses the minimum number of registers.

For our second set of experiments, we implemented our method in the SGI MIPSpro compiler suite, a set of highly-optimizing compilers for Fortran, C, and C++ on MIPS processors. We report both static and dynamic performance measures for four different versions of the compiler on SPEC-95 Floating Point suite [7]. The static performance measures include the number of basic blocks that required register spilling, the average spills per basic block as well as the total number of spill instructions inserted in the code. We report execution time of the compiled program on a MIPS R10000, and the number of dynamic loads and stores that graduate from the pipeline as dynamic performance measures. The static and dynamic performance measures are reported in Section 5.2

[7] Our experiments with the SPEC Int suite resulted in no measurable differences in execution time between the MRIS and the optimized compiler. This is possibly because of the facts that the basic blocks in the Spec Int benchmarks are smaller in size and our implementation of the MRIS approach performs instruction sequencing only within the basic block.

Estimate's Relation	Register Requirement			Number of DDGs		
	ILP	HRB	Sequence	DDGs	Total	Percentage
ILP = HRB = Sequence	1-8	1-8	1-8	650	650	96.3%
ILP = Sequence $>$ HRB	3	2	3	10		
	4	3	4	5		
	5	4	5	1		
	7	3	7	1		
	8	7	8	2	19	2.8%
ILP = HRB $<$ Sequence	4	4	5	2		
	6	6	7	1	3	0.4%
ILP $<$ HRB = Sequence	2	3	3	1		
	3	4	4	2	3	0.4%

Table 1: Comparison between the number of register required for MOST DDGs from ILP (exact), HRB (estimation), and an actual sequence. We present the number of DDGs and the register requirements for each estimate relation.

5.1 Comparing the Heuristic with the Exact Results

In order to compare the heuristic register bound (HRB) found by our method with the exact register requirement obtained with the ILP approach, we use a set of loops from Erik Altman's MOST framework [2]. The MOST framework is a collection of 1200 loops extracted from SPEC92 (integer and floating point), linpack, livermore, and the NAS kernels benchmarks. These DDGs were extracted with an optimizing research compiler. However many of these DDGs had a large number of nodes and edges; when the exact solution was tried for these DDGs, the ILP solver couldn't get a solution even after a reasonably long execution time. In the interest of time, we eliminated such DDGs and used a set of 675 DDGs in our experiments. The DDGs considered in our experiments vary widely in size with a median of 10 nodes, a geometric mean of 12 nodes, and an arithmetic mean of 19 nodes per DDG.

Out of the 675 DDGs, in 650 (96.3% of the total) the estimated HRB is exact and our sequencing algorithm obtains an instruction sequence that uses the minimum number of registers. In Table 1 we present the exact minimum register requirement found by the ILP formulation, the estimated register requirement found by the HRB approach, and the number of registers used by our lineage-based sequencing algorithm. We separate the loops in which these estimates are identical to the minimum register requirement (the 650 loops in the first row of the table) from those in which they are not (the last three rows in the table). In 19 loops (2.8% of total) although the HRB is an underestimation of the number of registers required, the sequencing algorithm based on lineage coloring used the minimum number of registers required. In three DDGs (0.4% of total) although the HRB is a correct estimation of the minimum number of registers, the sequencing algorithm used one extra register. And in other three loops, both the HRB estimate and the number of registers required by our heuristic sequencing algorithm are one more than the optimal value. Thus, out of the 675 DDG tested, only in 6 of them (0.8%) the heuristic sequencing used one more register than the minimum required. These results give us confidence in the quality of the instruction sequence produced by the lineage algorithm.

5.2 Comparison with a Production Compiler

In a second set of experiments we implemented our heuristic instruction sequencing method in the SGI MIPSpro compiler. This compiler performs extensive optimizations including copy propagation, dead-code elimination, if-conversion, loop unrolling, cross-iteration optimization, recurrence breaking, instruction scheduling, and register allocation. The compiler also implements an integrated global-local scheduling algorithm [26] that is invoked before and after register allocation. The global register allocation, based on graph coloring [12, 8], is followed by local register allocation. After register allocation, the data dependence graph is rebuilt and a post-pass scheduling is invoked. During local register allocation, the MIPSpro compiler inserts spill code according to a cost function that estimates the number of load/store spill instructions that each spill candidate will incur. This estimation only takes into consideration uses within the same basic block.

In our implementation, the 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 again on the new instruction sequence. The goal of the instruction sequencing algorithm is to reduce the amount of spill code executed by reducing the register pressure of the generated code. We refer to this version of the compiler as HRB-based sequencing, or simply as the HRB approach[8].

[8]It should be noted that although the original objective of the MRIS problem is to arrive at an instruction sequence that uses a minimal number of registers for a basic block, in our implementation the HRB-based sequencing is applied only to those basic blocks whose register requirement is greater than the available number of registers. Thus, the HRB sequencing approach is only used to reduce the register pressure for basic blocks which do incur register spills.

The performance of the HRB optimized version is evaluated against a baseline version of the MIPSpro compiler. We also measure the HRB approach against an optimized version of the MIPSpro compiler which includes a combined instruction scheduling and register allocation algorithm. Thus, in the experimental results presented in this section we compare four versions of the compiler:

Baseline The baseline compiler includes several traditional optimizations, such as copy propagation, dead-code elimination, *etc.,* listed earlier in this subsection. In fact all four compiler versions apply these optimizations. In addition, the baseline compiler traverses the instructions of a basic block in reverse order to perform local register allocation, but does not try to optimize the instruction sequencing when the local register allocator requires spill code.

Optimized In this version of the compiler, in addition to the traditional optimizations, there is a combined instruction scheduling and register allocation algorithm that is implemented in the MIPSpro compiler. The instruction sequencing algorithm used for this optimization is a depth-first traversal algorithm that takes resource constraints into consideration.

HRB The HRB compiler version also includes traditional optimizations, but when the local register allocator detects the need to spill code in a basic block, the instruction sequencing and register allocation algorithm based on the formation of lineages and on the heuristic register bound described in this paper is applied. Register allocation is performed in the new instruction sequence.

HRB (No Fusion) Identical to HRB, except that the lineage fusion algorithm is not implemented.

The baseline compiler is an optimized implementation including the integrated global and local scheduling, global and local register allocation and optimized spilling. However, when register pressure is high (which is the case for some of the SPEC FP benchmarks), a more sophisticated instruction scheduling geared toward reducing register pressure is needed. The optimized version of the compiler uses this additional instruction scheduling, while the baseline does not.

We present our performance results for a machine with 32 integer and 32 floating point architected registers. Because the HRB algorithm is more effective in application with high register pressure, we will also present the performance results for a machine with 32 integer and 16 floating point architected registers. This is the same target processor, but in the machine description file in the compiler we restrict the available registers to 16, so that the compiler cannot use half of the FP register file. This in effect increases the register pressure and, hence, spills.

5.2.1 Static Measurements

First we report static performance measures such as the number of spill instructions inserted statically in each benchmark by different versions of the compiler. Columns 2, 4, and 7 in Table 2 reports the number of basic blocks in which spill code was introduced by the different versions of the compilers.[9] We report the total number of spill instructions introduced by the different versions of the compiler in each application in columns 3, 5, 8, and 11. Lastly, columns 6, 9, and 12 represent, respectively, the percentage reduction in the number of (static) spill instructions introduced by the Optimized, HRB (No Fusion), and HRB versions of the compiler compared to the baseline version. We report the percentage reduction in the spill instructions compared to the baseline version of the compiler. For instance, for the HRB version of the compiler, the percentage reduction is defined as:

$$\%Reduction(P, HRB) = 100 \times \frac{(Spills(P, Baseline) - Spills(P, HRB))}{Spills(P, Baseline)} \tag{14}$$

where $Spills\ (P, HRB)$, for example, refers to the number of spills in benchmark P under the HRB version of the compiler. The top or bottom section of the table reports the performance for a target machine with 32 or 16 FP registers. In each section, the last row reports the average number of basic blocks that incurred spill, the average number of spill instructions per benchmark, and the percentage reduction in the average spill instructions compared to the baseline version. The average percentage reductions in this row are computed as the percentage reduction in the average number of spills inserted in all the benchmarks, and not as the average of the percentage reduction for the different benchmarks.

In Table 2, the total number of basic blocks that require spilling for a specific version of the compiler can be obtained by adding the entries in the column "Blocks with Spill". For a machine with 16 FP registers, the total number of blocks that require any spilling is reduced from 210 in the Baseline compiler to 94 in the HRB compiler, *i.e.,*55% of the blocks that required spill operations before no longer do. Compared to this, the Optimized MIPSpro inserts spill code in 152 blocks (a reduction of only 28%). In a machine with 32 FP registers, the baseline compiler inserts spills in 88 blocks compared with 32 in the HRB (a reduction of 63.6%) and 51 blocks in the optimized compiler (a reduction of 42.1%).

A comparison of the average number of spills in each benchmark, reveals that the HRB version reduces the spills by 63.1% and 55.89%, respectively, in the 32 FP and 16 FP register machines,

[9] Besides the SPEC-95 FP benchmarks reported in the paper, we also conducted our experiments for `swim`, `mgrid`, and `hydro2d`; but for these benchmarks there are no basic blocks with spills even under the baseline version of the compiler. Hence we do not include them in our results.

Benchmark	Compiler Version										
	Baseline		Optimized			HRB(No Fusion)			HRB		
	Blocks w/spills	Total Spills	Blocks w/spills	Total Spills	% reduc.	Blocks w/spills	Total Spills	% reduc.	Blocks w/spills	Total Spills	% reduc.
A Machine with 32 INT and 32 FP Registers											
tomcatv	1	1	0	0	100.00	0	0	100.00	0	0	100.00
su2cor	8	36	1	6	83.33	1	5	86.11	1	5	86.11
applu	22	1080	14	454	57.96	11	337	68.80	10	340	68.52
turb3d	10	67	7	69	-2.99	1	28	58.21	0	0	100.00
apsi	20	287	11	135	52.96	9	119	58.54	9	113	60.63
fpppp	17	1180	13	782	33.73	12	818	30.68	12	560	52.54
wave5	10	111	5	50	54.95	1	1	99.10	0	0	100.00
Average	**12.57**	**394.57**	**7.29**	**213.71**	**45.84**	**5.00**	**186.86**	**52.64**	**4.57**	**145.43**	**63.14**
A Machine with 32 INT and 16 FP Registers											
tomcatv	2	38	1	22	42.11	2	16	57.89	1	13	65.79
su2cor	13	80	2	22	72.50	5	44	45.00	1	5	93.75
applu	63	1678	51	1130	32.66	43	945	43.68	35	819	51.19
turb3d	23	530	18	342	35.47	14	325	38.68	10	169	68.11
apsi	42	694	30	332	52.16	25	213	69.31	17	146	78.96
fpppp	23	1939	18	1391	28.26	16	1540	20.58	14	1276	34.19
wave5	44	827	32	492	40.51	26	223	73.04	16	124	85.01
Average	**30.00**	**826.57**	**21.71**	**533.00**	**35.52**	**18.71**	**472.29**	**42.86**	**13.43**	**364.57**	**55.89**

Table 2: Static count of the total number of spill operations inserted (32 INT and 16 FP registers).

compared to the baseline compiler. More specifically, for a machine with 16 FP registers, in terms of the total number of spills, the reduction due to HRB ranges from 34% to 94%. For a machine with 32 FP register, the percentage reduction varies from 52.5% to 100%. In comparison, MIPSpro's optimizing compiler reduces the average of number of spills by only 45.8% and 35.5% respectively for the two machines. The most dramatic improvement when comparing the HRB with the optimized compiler is observed for `fpppp`, `applu`, and `wave5` benchmarks, for a machine with 32 and 16 FP registers, where the optimized version incurs 200 or more static spill instructions compared to the HRB approach. Thus even compared to the optimized compiler, the HRB approach produces code that significantly reduces number of spills.

The fusion of lineages reduces the number of spill operations inserted in the code in relation to a version of the HRB algorithm that does not perform lineage fusion. Compared to the HRB version which reduced the average spills by 63.1% and 55.9%, respectively, for machines 32 and 16 FP registers, the HRB version without lineage fusion, resulted in a reduction of only 52.6% and 42.9%.

Lastly, notice that the reduced percentage improvement in average spills for the 32 FP register machine is less than that for the 16 FP register machine. This result may seem to be counter-intuitive, given that the HRB algorithm should perform well for programs with higher register pressure. Note, however, that in the 16 FP machine, on average 462 spills were eliminated, while for the 32 FP machine this number was 249. Thus, the lower percentage reduction is only due to the large denominator value (average spills in the baseline version).

5.2.2 Dynamic Measurements

To report dynamic performance, we conducted our experiments on a Silicon Graphics machine with a 194 MHz MIPS R10000 processor, 32 KB instruction cache, 32 KB data cache, 1 MB of secondary unified instruction/data cache, and 1 GB of main memory. 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 R10000 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 level, the reduction in the number of loads/stores executed in each benchmark program corresponds to the number of spill loads/stores reduced by the HRB approach.

First we report the dynamic measurements for spill instructions. The number of loads and stores graduated from the pipeline for each benchmark in each version of the compiler are shown in Tables 3 and 4. Each table shows the number of operations (in billions) as well as the percentage reductions in relation to the baseline compiler. The percentage reduction is computed using an equation similar to 14. The average number of loads reported in the last row of each section of a table is simply the arithmetic mean of the number of graduated loads in each of the benchmarks. However, the percentage reductions in these rows is the percentage reduction of the average number of loads. In other words, the average reported under the "% reduction" column is not the average of the percentage reductions. The same averaging method is used for reporting the graduated stores (see Table 4) and the execution time (see Table 5).

Benchmark	Compiler Version						
	Baseline	Optimized		HRB(No Fusion)		HRB	
	Loads	Loads	% reduc.	Loads	% reduc.	Loads	% reduc.
A Machine with 32 INT and 32 FP Registers							
tomcatv	6.51	6.37	2.1	6.07	6.7	6.19	5.0
su2cor	5.83	5.82	0.1	5.83	0.1	5.82	0.1
applu	8.07	7.56	6.3	7.51	7.0	7.51	7.0
turb3d	11.63	11.63	0.0	11.58	0.4	11.59	0.3
apsi	4.96	4.83	2.5	4.73	4.6	4.67	5.7
fpppp	28.40	24.65	13.2	23.13	18.6	22.45	20.9
wave5	3.72	3.72	-0.2	3.69	0.8	3.69	0.7
Average	**9.87**	**9.23**	**6.53**	**8.93**	**9.53**	**8.85**	**10.4**
A Machine with 32 INT and 16 FP Registers							
tomcatv	8.42	7.84	6.9	7.25	13.9	6.88	18.3
su2cor	6.29	5.92	5.9	6.48	-3.0	5.68	9.7
applu	8.77	8.42	3.9	8.19	6.5	8.12	7.4
turb3d	14.75	13.30	9.8	12.02	18.5	12.65	14.2
apsi	5.38	5.15	4.4	5.02	6.8	4.70	12.7
fpppp	35.04	28.00	20.2	30.52	12.9	26.81	23.5
wave5	3.88	3.75	3.3	3.69	5.0	3.73	3.9
Average	**11.79**	**10.34**	**12.3**	**10.45**	**11.33**	**9.8**	**16.9**

Table 3: Total number of graduated loads (in billions) for each benchmark, and percentage reduction in relation to the baseline Compiler (32 INT and 16 FP registers).

The HRB compiler reduces the average number of loads and stores executed, respectively, by 10.4% and 3.7% for a machine with 32 FP registers. These numbers for a machine with 16 registers are 16.9% and 3.5%, respectively. Even when compared with the optimized MIPSpro compiler, the reduction in the average number of loads and stores is significant (almost 400 and 200 Million instructions). Thus reducing the number of static spill instructions, results in a reduction in the number of dynamic spill instructions executed. The drastic improvement in a single benchmark is observed for `fpppp`, where 8.2 and 5.95 Billion loads are eliminated (compared to the baseline version), for machines with 16 and 32 FP registers respectively.

Benchmark	Compiler Version						
	Baseline	Optimized		HRB(No Fusion)		HRB	
	Stores	Stores	% reduc.	Stores	% reduc.	Stores	% reduc.
A Machine with 32 INT and 32 FP Registers							
tomcatv	2.34	2.34	0.0	2.34	0.0	2.34	0.0
su2cor	2.84	2.84	0.0	2.84	0.0	2.84	0.0
applu	5.24	4.75	9.3	4.65	11.3	4.65	11.2
turb3d	13.33	13.34	0.0	13.24	0.6	13.24	0.6
apsi	3.17	3.17	0.0	3.17	0.0	3.17	0.0
fpppp	20.78	20.68	0.5	19.67	5.3	19.55	5.9
wave5	3.76	3.76	0.0	3.76	0.0	3.76	0.0
Average	**7.35**	**7.27**	**1.1**	**7.1**	**3.5**	**7.08**	**3.7**
A Machine with 32 INT and 16 FP Registers							
tomcatv	3.22	2.63	18.2	2.54	21.2	2.73	15.2
su2cor	2.86	2.86	0.0	2.86	0.0	2.86	0.0
applu	5.88	5.37	8.6	5.31	9.7	5.16	12.3
turb3d	14.03	13.81	1.5	13.86	1.2	13.82	1.4
apsi	3.52	3.39	3.7	3.29	6.6	3.19	9.4
fpppp	21.37	22.09	-3.4	20.60	3.6	21.26	0.5
wave5	3.90	3.88	0.7	3.85	1.3	3.83	1.8
Average	**7.83**	**7.72**	**1.4**	**7.47**	**4.5**	**7.55**	**3.5**

Table 4: Total number of graduated stores (in billions) for each benchmark, and percentage reduction in relation to the baseline Compiler(32 INT and 16 FP registers).

Next we report whether the reduction in the number of loads and stores executed corresponds to a reduction in the execution time of the benchmarks. Table 5 presents the execution time for each benchmark under different versions of the compiler. This is the wall-clock time, measured in seconds, required to execute each benchmark. We also present the percentage reduction in the execution time. First, we notice that, by and large, there is a correlation between the improvement in execution time in Table 5 and the reduction in the number of loads and stores graduated. The drastic improvements in the number of loads and stores seen in `fpppp`, do translate into a significant improvement in execution time: 14.2% and 11.4%, respectively, for machines with 32 or 16 FP registers. Once again, although

Benchmark	Compiler Version						
	Baseline	Optimized		HRB(No Fusion)		HRB	
	Time	Time	% reduc.	Time	% reduc.	Time	% reduc.
A Machine with 32 INT and 32 FP Registers							
tomcatv	358.99	354.67	1.20	360.38	-0.39	354.17	1.34
su2cor	230.12	229.99	0.06	230.37	-0.11	229.76	0.16
applu	397.41	392.16	1.32	388.59	2.22	385.92	2.89
turb3d	393.94	394.06	-0.03	393.62	0.08	395.17	-0.31
apsi	248.22	249.79	-0.63	249.53	-0.53	248.31	-0.04
fpppp	391.82	361.39	7.77	365.46	6.73	336.11	14.22
wave5	198.84	199.28	-0.22	198.50	0.17	198.81	0.01
Average	**317.05**	**311.62**	**1.71**	**312.35**	**1.48**	**306.89**	**3.20**
A Machine with 32 INT and 16 FP Registers							
tomcatv	375.00	363.31	3.12	362.55	3.32	358.58	4.38
su2cor	232.22	229.29	1.26	232.42	-0.09	231.73	0.21
applu	402.41	404.22	-0.45	397.28	1.27	393.65	2.18
turb3d	415.63	411.09	1.09	409.44	1.49	404.41	2.70
apsi	253.25	253.18	0.03	252.48	0.31	249.81	1.36
fpppp	447.98	411.82	8.07	414.27	7.53	396.79	11.43
wave5	200.43	198.79	0.82	200.55	-0.06	200.10	0.16
Average	**332.42**	**324.53**	**2.37**	**324.14**	**2.49**	**319.30**	**3.95**

Table 5: Execution time (in seconds) and reductions in the execution time in relation to the baseline compiler.

the reduction in execution time compared to the respective baseline version is the same for different target machine configurations for which they were compiled, the lower percentage reduction in the 16 FP register case is due to the larger execution time of the baseline version.

From Table 5, it can be seen that the code produced by the HRB version of the compiler does, in fact, reduce the execution time, although the percentage reduction is somewhat low. However, we remark that one should not be discouraged by the somewhat low improvement in execution time due to the HRB approach. The HRB approach performs instruction sequencing solely with the objective of reducing the register requirements. Our implementation of the HRB algorithm does not take into account the resource constraints of the architecture, often sacrificing some instruction-level parallelism. The average execution time under the HRB version is comparable or better than that under the Optimized version in each of the benchmarks. Our arguments to support this approach is that modern superscalar processors with their out-of-order issue hardware should be able to uncover the instruction-level parallelism that is obscured by false register dependences. Compared to this, the MIPSpro Optimized version uses an integrated register allocation and instruction scheduler. Our experiments, in fact, demonstrate that the HRB sequencing approach minimizes the register requirements. At the same time, the execution of these programs resulted in a performance that is on par or better than the Baseline or Optimized MIPSpro version in terms of execution time. We have also witnessed a significant reduction (a few billions) in the number of loads and stores. This reduction is primarily due to the savings in spill code. As indicated by Cooper and Harvey [13], reducing the spill load/store operations results in less cache pollution and hence higher cache performance. Further reducing memory operations can significantly reduce the power dissipated. These are other advantages of the HRB approach in addition to achieving a reasonable performance improvement in execution time.

6 Related Work

Instruction scheduling [16, 28] and register allocation [1, 8, 10, 11, 15, 28, 30, 32, 39] 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 [6, 5, 27, 29, 31]. All these approaches work on a *given* instruction sequence and attempt to improve register allocation and/or instruction scheduling. In contrast, our MRIS approach generates an instruction sequence from a DDG where the precise order of instructions is not yet fixed.

Modern out-of-order issue superscalar processors, which have the capabilities to perform instruction scheduling and register renaming at runtime, shift the focus of the instruction-level compilation techniques. Studies on out-of-order issue processors indicate that reducing the register pressure, and hence the number of memory spill instructions executed is more crucial to the performance than increasing the instruction level parallelism [36, 40].

The Minimum Register Instruction Sequence (MRIS) problem studied in this paper is different from the traditional *register allocation problem* [1, 8, 10, 11, 15, 28, 30]. Recently, there have also been a few proposals on register allocation based on integer linear programming [3, 18, 25]. 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, 16, 17, 28, 41]. 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 also closely related to the *optimal code generation (OCG)* problem [1, 34, 33]. 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 [4]. 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, Berson, Gupta, and Soffa used *register reuse dags* for measuring the register pressure [5]. A register reuse dag is similar to the lineage discussed in this paper. They have evaluated register spilling and register splitting methods for reducing the register requirements in the URSA method for individual loops, rather than the whole application. Our work, in contrast, reports static and dynamic performance measures on SPEC-95 floating point benchmark suite.

In the experiments presented in Section 5, our lineage formation method was applied only to basic blocks for which an heuristic-based local register allocator could not perform the allocation using the number of registers available for the basic block. In other words, we only apply the lineage based algorithm for basic blocks that incur in spills. Thus the aggressive sequentialization in our algorithm does not prevent parallelism when an allocation is found that does not incur in spills. In an independent work, Touati proposes an approach to perform register allocation and code scheduling in a single pass by generating a schedule that maximizes the number of values live at the same time [38]. His reasoning is that when more values are simultaneously live the scheduler will find more opportunities to explore parallelism. Thus instead of creating long chains (lineages in our case), he attempts to create maximal anti-chains. When maximal anti-chains result in a register requirement that is greater than the number of available registers, Touati uses an approximate algorithm to find a quasi-optimal set of serialization arcs to reduce the number of registers used. In the few examples that we examined, his register saturation reduction algorithm and our lineage fusion algorithm created the same set of sequentialization edges. It is interesting to note that these two algorithms use quite contrasting approaches (maximizing vs. minimizing the register requirements) to achieve the same objective of minimizing the register spills.

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 [19]. 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.

7 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 out-of-order issue processors. In out-of-order issue processors, excessive register spills can potentially degrade the performance, and hence must be avoided even at the expense of reducing the parallelism exposed at compile time. Another motivation for the MRIS problem stems from the fact that register spills lead to memory accesses which are expensive in terms of power dissipation.

We proposed two solutions to the MRIS problem. First, an heuristic solution that uses lineage formation, lineage interference graph, and a modified and efficient list scheduling method to obtain efficient near-optimal solution to the MRIS problem. The second approach is based on an elegant integer linear programming formulation. Compared to the exact ILP-based approach for MRIS, the heuristic approach results in optimal solution for 99.2% of the DDGs used for our experiments, although these DDGs are small in size and have low register requirements.

We evaluated the performance of our heuristic method by implementing it in the MIPSpro production compiler, and running SPEC95 floating point benchmarks. Our experimental results demonstrate that our instruction reordering method, which attempts to minimize the register requirements, reduces the average number of basic blocks that require spilling by 62.5% and the average spill operations in the (static) code by 63.1%. As a consequence, the HRB approach also reduces the average number of loads and stores executed by 10.4% and 3.7% respectively for a machine with 32 integer and 32 floating point registers. This reduction in loads and stores also results in an improvement in the average execution time by 3.2%.

The MRIS approach to register allocation and instruction sequencing might allow a smaller cache, and reduce the traffic of data between the processor and the memory structure, thus contributing to the reduction of power consumption. It might also reduce the need for spilling from the register stack in an IA-64 machine. An experimental study to test these hypothesis will be welcomed by the community.

Acknowledgments

This research is supported by the National Science Foundation (NSF) and by the Natural Sciences and Engineering Research Council (NSERC) of Canada. We would also like to acknowledge current and former members of CAPSL for valuable discussions, and Jim Dehnert and Sun Chan, formerly with SGI, for discussions about the organization of the MIPSpro compiler. Special thanks to the anonymous reviewers whose comments helped us improve the quality of this manuscript.

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] E. R. Altman. *Optimal Software Pipelining with Function Unit and Register Constraints*. PhD thesis, McGill Univ., Montréal, Québec, Oct. 1995.

[3] A. W. Appel, L. George. Optimal spilling for CISC machines with few registers. In *Proc. of the ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation*, pages 243–253, Snowbird, UT, June 2001.

[4] 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.

[5] 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.

[6] 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.

[7] P. Briggs, K. D. Cooper, and L. Torczon. Rematerialization. In *Proc. of the ACM SIGPLAN '92 Conf. on Programming Language Design and Implementation*, pages 311–321, San Francisco, CA, June 17–19, 1992.

[8] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. on Programming Languages and Systems*, 16(3):428–455, May 1994.

[9] J. L. Bruno and R. Sethi. Code generation for a one-register machine. *J. of the ACM*, 23(3):502–510, Jul. 1976.

[10] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proc. of the SIGPLAN '82 Symp. on Compiler Construction*, pages 98–105, Boston, MA, June 1982.

[11] 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.

[12] 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.

[13] K. D. Cooper and T. J. Harvey. Compiler-Controlled Memory. In *Architectural Support for Programming Languages and Operating Systems*, pages 2-11, San Jose, CA, Oct, 1998

[14] G. R. Gao, L. Bic, and J.-L. Gaudiot. *Advanced Topics in Dataflow Computing and Multithreading*, IEEE Computer Society Press. Book contains papers presented at the Second International Workshop on Dataflow Computers held in May 1992 in Hamilton Island, Australia, 1995.

[15] 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.

[16] 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.

[17] 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.

[18] D. W. Goodwin and K. D. Wilken Optimal and near-optimal global register allocation using 0–1 integer programming. *Software—Practice and Experience* 26(8):929– 965, August 1996.

[19] 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`)

[20] R. Govindarajan, H. Yang, J.Nelson Amaral, C. Zhang, and G.R. Gao. Minimum register instruction sequence problem: Revisiting optimal code generation for DAGs In *Proceedings of the International Parallel and Distributed Processing Symposium*, (available at `http://www.capsl.udel.edu/COMPILER/IPDPS01_Govind.ps.gz`), San Francisco, CA, April 2001.

[21] L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs *The Journal of Programming Languages*, 1(3):155–185, 1993.

[22] T. C. Hu. *Integer Programming and Network Flows*, page 270. Addison-Wesley Pub. Co., 1969.

[23] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Jl. of Supercomputing*, 7:229–248, 1993.

[24] Intel. *Intel IA-64 Architecture Software Developer's Manual*, Jan 2000.

[25] C.W. Kessler. Scheduling expression DAGs for minimal register need. In *Proc. of 8th Intl. Symp. on Programming Languages: Implementations, Logics, and Programs (PLILP'96)*, Springer LNCS 1140, pp. 228-242, Aachen, Sept. 1996.

[26] S. Mantripragada, S. Jain, and J. Dehnert. A new framework for integrated global local scheduling. In *Proc. of the 1998 Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 167–174, Paris, France, Oct. 12–18, 1998. IEEE Comp. Soc. Press.

[27] 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.

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

[29] B. Natarajan and M. Schlansker. Spill-free parallel scheduling of basic blocks. In *28th Annual International Symposium on Microarchitecture*, pages 119-124, Ann Harbor, Michigan, December, 1995.

[30] 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.

[31] 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.

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

[33] 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.

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

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

[36] 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.

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

[38] S. A. A. Touati. Register Saturation in Superscalar and VLIW Codes In *10th International Conference on Compiler Construction*, pages 213-228, Genova, Italy, April 2001.

[39] 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.

[40] M. 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.

[41] 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.

[42] W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke. *The Design of an Optimizing Compiler*, Programming Languages Series. American Elsevier, New York, N. Y., 1975.