**University of Delaware**
**Department of Electrical and Computer Engineering**
**Computer Architecture and Parallel Systems Laboratory**

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

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

**CAPSL Technical Memo 36**
November 12, 1999

Copyright © 1999 CAPSL at the University of Delaware

†Supercomputer Education and Research Centre and Dept. of Computer Science and Automation, Indian Institute of Science Bangalore 560 012, India. E-mail: govind@csa.iisc.ernet.in

**Abstract**

In this paper we address the problem of generating an instruction sequence $S$ from a data dependence graph (DDG) that is optimal in terms of the number of registers used by the sequence $S$. This problem is closely related to the traditional optimal code generation (OCG) problem, which minimizes the code length, and is known to be NP-Complete for DAGs. While traditional code generation methods and instruction-level parallelism compilation techniques emphasize exposing parallelism to reduce the code length, there are at least two execution environments — out-of-order issue superscalar processors and low-power embedded architectures — where optimizing the number of registers used takes precedence, even at the expense of limiting the amount of instruction-level parallelism exposed at compile-time.

We present two solution methods for the Minimum Register Instruction Sequence (MRIS) problem. First, we present a heuristic solution which introduces the notion of *instruction lineage* that facilitates the sharing of registers among instructions within a lineage and across lineages by exploiting the structure of a DDG. Our second solution is through an elegant integer linear programming (ILP) formulation for the MRIS problem.

We compared the number of registers used by our heuristic solution, with that used in the solution generated by an implementation of the Sethi-Ullman algorithm, and with the minimum register requirement (MRR) computed by the ILP method for 852 DDGs extracted from standard benchmark sets. Our heuristic solution found the optimal solution in 99.4% of the 671 DDGs where the ILP method found the MRR. For the remaining 181 DDGs, our heuristic method performed better than Sethi-Ullman's algorithm in 100 DDGs, and worse in only 6 DDGs.

# Contents

**1 Introduction** **1**

**2 Motivating Example** **3**
2.1 Open Problems . . . . . 4

**3 Heuristic Approach to MRS Problem** **5**
3.1 Lineage Formation . . . . . 5
3.2 Lineage Interference Graph and Heuristic Register Bound . . . . . 7
3.3 Instruction Sequence Generation . . . . . 8
3.4 Heuristics Employed . . . . . 9

**4 Exact Approach to MRS Problem** **10**

**5 Experimental Results** **12**
5.1 Experimental Framework . . . . . 12
5.2 Performance of Heuristic Methods . . . . . 13
5.2.1 Optimality of Heuristic Solutions . . . . . 13
5.2.2 Effectiveness of Heuristic Methods in Larger DDGs . . . . . 14
5.2.3 Comparison with OCG Method . . . . . 14
5.3 Summary of Main Results . . . . . 14

**6 Related Work** **15**

**7 Conclusions** **16**

**A Details of Heuristic Algorithms** **19**

# List of Figures

1 Motivating Example . . . . . 3
2 Data Dependence Graph for the Motivating Example . . . . . 4
3 Lineage Formation Algorithm . . . . . 20
4 Scheduling Algorithm . . . . . 20

# List of Tables

1 Tightness of Heuristic Register Bound . . . . . 13
2 Performance of Heuristic Methods vs. Sethi-Ullmann on 181 Large Loops . . . . 15

# 1 Introduction

Instruction scheduling [2, 10, 11, 17, 29] and register allocation [2, 7, 8, 9, 17, 18, 20, 28] play a major role in generating efficient code for modern high-performance processors. In this paper we study the following *Minimum Register Instruction Sequence (MRIS)* problem:

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

In contrast to the traditional register allocation problem [2, 8, 17], the input to the MRIS problem is a DDG instead of a totally ordered sequence of instructions. The partial order specified by a DDG does not impose any unnecessary overlap of live ranges, and hence facilitates obtaining a sequence that requires fewer number of registers. The total ordering of instructions, on the other hand, makes the traditional register allocation problem for basic blocks easier, for which efficient solutions exist [20, 28]. The MRIS problem is also quite different from the traditional *instruction scheduling problem* [2, 10, 11, 17]. In traditional instruction scheduling, the main objective is to minimize the total time (or length) of the schedule: the problem formulation and its solutions must take into account the execution latencies of each operation (instruction) in the DDG. In contrast, the latency of operations in the DDG and the availability of function unit resources are not a part of the MRIS problem formulation! To highlight this difference, we distinguish the use of the two terms "*instruction schedule*" and "*instruction sequence*".

The MRIS problem is closely related to the *optimal code generation (OCG) problem* [2, 22]. For the case in which the dependence graph is a tree, an optimal algorithm, optimal in terms of code length, for the OCG problem exists. For a general DAG, the problem is known to be NP-Complete since 1975 [22]. An important difference between traditional code generation methods and our MRIS problem is that the former emphasizes on reducing the code length (or schedule length) while the latter minimizes the number of registers. Further the code generation problem is specifically for generating a minimum schedule length code for a *given* number of registers.

Our interests to study the MRIS problem are motivated by the requirements of at least two classes of architectures:

- In embedded systems, power is conserved if the number of switching cycles in the processor-memory bus is minimized [1, 26, 27]. Typically these systems have a limited amount of cache memory, and thus there is a high probability that a temporary value spilled to the memory will be evicted from the cache before it is reloaded. Therefore in such architectures, the consumption of power is reduced when the amount of spill code introduced by the compiler is minimized, which can be achieved by minimizing the number of registers used for the computation.
- A number of modern superscalar processors support out-of-order instruction issue and execution [25]. Out-of-order (o-o-o) instruction issue is facilitated by a runtime scheduling

hardware and by the register renaming mechanism. An o-o-o processor has a larger number of physical (*i.e.,* not architected) registers at its disposal for register renaming at runtime. Recent studies [15, 24] indicate that compilers for o-o-o issue processors should emphasize reducing the number of register spills than exposing the instruction level parallelism.

In this paper, we have developed a simple and efficient heuristic method to solve the MRIS problem. The proposed method is based on the following:

- *Instruction lineage formation:* The concept of *instruction lineage* evolves from the notion of *instruction chains* [12] which allows the sharing of a register among instructions along a (flow) dependence chain in a DDG. However, the instruction lineage approach more accurately models the register requirement than an instruction chain (as will be discussed in Section 6).
- The notion of a lineage interference graph which captures the *definite* overlap relation between the live ranges of lineages even before the lineages are scheduled, and its use to facilitate sharing of registers across lineages.
- The notion of *node affinity* and its use to determine the priority order of the *ready* instructions in generating the instruction sequence. Affinity captures the "closeness" between instructions in different lineages so they should be scheduled or listed together in order to facilitate the register sharing and thus reducing the register pressure.

In order to assess the optimality and efficiency of the proposed heuristic method, we have developed an efficient integer linear programming (ILP) formulation for the MRIS problem. We have implemented our heuristic approach as well as the integer programming approach using the MOST framework [3] and have evaluated its performance on a set of 852 DDGs derived from scientific benchmark programs. The results are summarized as follows.

- For 671 DDGs, both the heuristic and the exact method have reached a solution. The heuristic method found an optimal solution in 99.4% of cases.
- In the remaining 181 cases, the ILP method does not obtain an optimal solution within a reasonable time limit. In these DDGs, our heuristic solution fared very well and used only a few extra registers above the register bound obtained from the lineage interference graph.
- Lastly, a comparison with the OCG algorithm reveals that our approach performed better (in terms of number of registers) in 100 out of the 181 DDGs, equal in 75 DDGs, and worse in only 6 DDGs. Moreover, in the 100 cases in which our register solution performed better, it used, on average, 2.6 fewer registers per DDG.

The rest of the paper is organized as follows. In the following section we motivate MRIS problem with the help of an example. In Section 3, we present our heuristic solution. In

Section 4, we formulate the MRIS problem as an ILP problem. Experimental results are discussed in Section 5. Related work and conclusions are presented in Sections 6 and 7.

## 2 Motivating Example

In this section we use an example to motivate the MRIS problem and illustrate our solution to it. Consider the computation represented by a data dependence graph (DDG)[1] shown in Figure 1(a). Two possible instruction sequences for this DDG are also shown in the figure along with the live ranges of the variables $s1$–$s7$ (for simplicity, we assume 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 $d$ and $e$, 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.

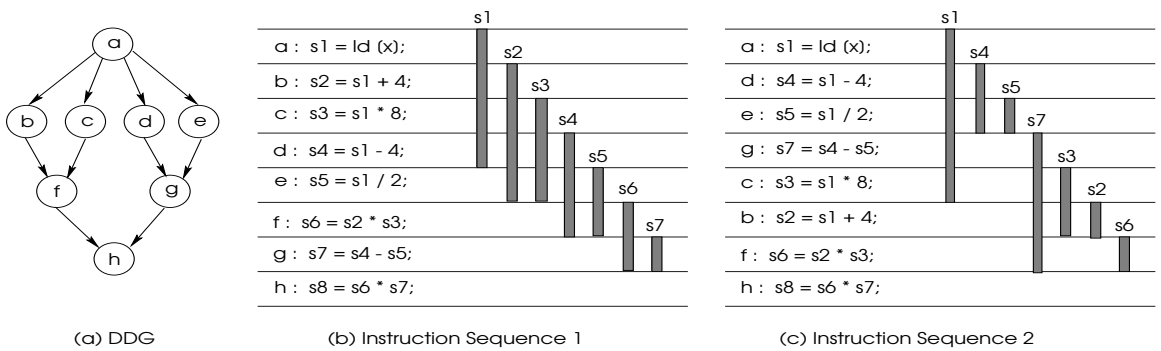


Figure 1: Motivating Example

The MRIS problem can be stated as: given a set of instructions and the data dependencies among them, build an instruction sequence that requires the minimum number of registers. To solve the MRIS problem, we consider a Data Dependence Graph (DDG) for the basic block which impose only a partial order among instructions. However, our method is applicable to superblocks [14] and basic blocks with predicated instructions. The nodes in the DDG represent instructions and the arcs represent data dependency. In this paper we restrict our attention to acyclic DDGs and hence do not consider any loop carried dependencies.

We need to identify which nodes (instructions) 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 dependence in the DDG does provide a partial answer. For instance, in the DDG of Figure 2(a), since there is a data dependency from node $b$ to $f$, and there is no other node to use the value

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

produced by node $b$, we can definitely say that in any legal sequence, the register associated with node $b$ can be shared by $f$. Similarly nodes $e$ and $g$ can share the same register. Next, can nodes $f$ and $g$ share the same register? The answer is no, as, in any legal sequence, the values produced by these instructions must be live simultaneously so that the computation of $h$ can take place.

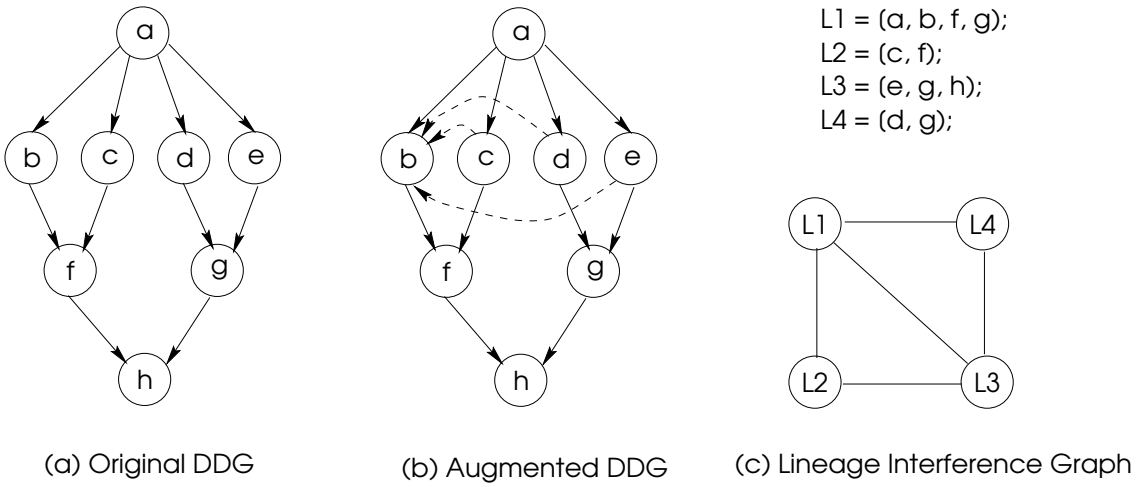


Figure 2: Data Dependence Graph for the Motivating Example

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

## 2.1 Open Problems

Our approach to the MRIS problem uses the notion of *instruction lineages* that is derived from the notion of *chains* introduced in [12]. If a node $S$ in a DDG has one or more descendents, it produces a value and thus must be assigned a register.[2] If we have a sequence of instructions in the DDG $S_1, S_2, S_3, \cdots, S_n$ where $S_2$ is the descendent of $S_1$, $S_3$ is the descendent of $S_2$, *etc.,* then we can form a *lineage* of these instructions in such a way that all the instructions in the lineage can share the same register. That is, the register assigned to $S_1$ is passed on to $S_2$ ($S_1$'s *heir*) which is passed on to $S_3$, and so on. Due to the data dependency between pairs of instructions in the lineage, any legal sequence will order the instructions as $S_1$, $S_2$, $\cdots$, $S_n$, and,

[2]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 our solution. A simple extension of our solution would insert a dummy sink node to capture live-out registers as in [16].

hence, if the live range of the variable defined by $S_1$ ends at $S_2$, then $S_2$ can definitely share the same register allocated to $S_1$.

What if $S_1$ has more than one descendent? In order for $S_2$ to use the same register that $S_1$ used, we must ensure that the other descendents of $S_1$ are scheduled before $S_2$. Thus the selection of one of the descendents of $S_1$ to be the *heir* of the register creates scheduling constraints among the descendents of $S_1$. Such sequencing constraints are explicitly represented in the *augmented DDG* by means of directed *sequencing edges* from each descendent node to the selected heir. For instance, the DDG for the motivating example is shown in Figure 2(a). The definition of the lineage $L1 = \{a, b, f, h\}$ creates scheduling constraints between the descendents of $a$. These sequencing edges are shown as dotted arrows in Figure 2(b). In Section 3.1 we introduce a simple but efficient heuristic to select the heir for a node having multiple descendents.

It is clear that all the nodes in a lineage share the same register. But can two lineages share the same register? To determine the interference between two lineages, we have to determine whether the live ranges of the lineages overlap in all legal instruction sequences. The live range of an instruction lineage is the concatenation of the live ranges of all the values defined by the instructions in the lineage. If the live ranges of two lineages overlap in all legal sequences, then the lineages cannot share the same register. However if they do not overlap in at least one of the legal sequences, then we may be able to schedule the lineages in such a way that they share a register. Based on the overlap relation we construct a lineage interference graph. This lineage interference graph is colored using traditional graph coloring algorithms [8] to compute the number of registers required for the DDG. We refer to this number as the *Heuristic Register Bound (HRB)*. Once we color the lineage interference graph, we apply a modified list scheduling method that uses this coloring as a guideline to generate an instruction sequence that uses the minimum number of registers. Our heuristic-based algorithm produces a near-optimal solution for the MRIS problem.

# 3 Heuristic Approach to MRS Problem

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

## 3.1 Lineage Formation

First we define the notion of an instruction lineage.

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

Although a node can have many descendents, it has at most a single heir. In our approach we form maximal lineages. A lineage is maximal in the sense that if $\{v1, v2, \cdots, v_n\}$ is a lineage

then either $v_n$ is a node with no descendents or $v_n$ is already associated with some other lineage. Because the heir is always the last descendent to be scheduled (it is the *last use* of the value produced by its parent), in an acyclic scheduling, the live range of the nodes in a lineage will definitely not overlap with each other and hence all the nodes in the lineage can share the same register. However, in order to ensure that the heir is the last use of the value produced by its parent, we introduce *sequencing edges* in the DDG, from each descendent of a node to the chosen heir, as shown in Figure 2(b).

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

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

In the DDG of Figure 2 in Section 2, the heights of the nodes are:

$$ht(a) = 4;\ ht(b) = 3;\ ht(c) = 3;\ ht(d) = 3;\ ht(e) = 3;\ ht(f) = 2;\ ht(g) = 2;\ ht(h) = 1;$$

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

Each 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. In forming lineages, our aim is to *cover* all the edges in the DDG, so that all register requirements can properly be accounted for. However a node with multiple descendents have multiple outgoing edges, all of which are associated with a single register. This is because the data dependence from the node to its descendents is through a single target register. Thus it is sufficient if one of these edges, preferably the one representing the last-use of that value, is covered during the lineage formation. Fortunately, as sequencing edges are introduced during the lineage formation, the edge between a node and its selected heir represents the def-*last_use*. And the lineage formation needs only to cover all def-last_use dependences. That is, each def-last_use must be included in exactly one lineage. In forming lineages, we make each lineage a maximal one, which may result in a minimal number of lineages required to cover the DDG. This in turn reduces the size of the *lineage interference graph*.

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 share its register (*i.e.,* use the register as its destination) with the other nodes in the lineage. We emphasize this property of the last node by denoting a lineage as $[v_1, v_2, \cdots, v_{(n-1)}, v_n)$ (as

in semi-open intervals). 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 $[d, g)$.

The Lineage Formation algorithm is essentially a depth-first graph traversal algorithm, identifying a heir for each node (using the height priority) and forming maximal lineages. In cases where a node has multiple descendents, sequencing are introduced, and the heights of all nodes in the DDG are recomputed. The detailed algorithm is presented in Figure 3 (refer to Appendix A). 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 *augmented DDG* (refer to Figure 2(b)).

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

**Proof:** Since the lowest descendent of a node is chosen as the heir, all sequencing edges inserted by this choice will be from nodes with higher heights to nodes with lower heights. Hence there can be no path in the DDG from a lower to a higher node, and therefore no cycle can be formed. Furthermore, the height of all the nodes in the graph is recalculated after choosing a heir for each node. □

### 3.2 Lineage Interference Graph and Heuristic Register Bound

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 $t_1$, and the last vertex $v_n$ is in position $t_n$ in the instruction sequence, then the* **live range** *of the lineage $L$ starts at $t_1$ and ends $t_n - 1$.*

When we try to identify lineages with non-overlapping live ranges, the instruction schedule is not yet fixed. In other words, the position of instructions in the sequence, and as a consequence, the live ranges of different lineages are all *floating*. That is, although the instructions in a lineage are totally ordered, instructions from other lineages may be interleaved in between in the final instruction sequence. The live range of a lineage is always contiguous, irrespective of what interleavings have happened. Thus once the first instruction in a lineage is listed, its live range is active until the position of the last node of the lineage in the instruction sequence.

We define a condition C1 that determines whether the live ranges to two lineages must necessarily overlap in *all* legal instruction sequences for the augmented DDG.

**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*

**C1:** *There exist directed paths from $u_1$ to $v_n$ and $v_1$ to $u_m$ in the augmented DDG.*

**Proof:** Since there exists a directed path (consisting of dependence and sequencing edges) from $u_1$ to $v_n$, $u_1$ must be listed before $v_n$. If $t_{u_1}$ represents the position of instruction $u_1$ in the instruction sequence, then $t_{u_1} < t_{v_n}$. Similarly, $t_{v_1} < t_{u_m}$. Further since $L_u$ is a lineage, $t_{u_1} < t_{u_m}$ and similarly $t_{v_1} < t_{v_n}$. From the above inequalities it is clear that the live range of $L_u$ must end after $t_{v_1}$ and the live range of $L_v$ must end after $t_{u_1}$. 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 $a \rightarrow d \rightarrow g$, and node $d$ can reach node $h$ through the path $d \rightarrow b \rightarrow f \rightarrow h$. Therefore, the live ranges of these two lineages must overlap. Similarly the live range of lineage $L_2$ overlaps with $L_1$, $L_3$ overlaps with $L_4$, $L_1$ overlaps with $L_3$, and $L_2$ overlaps with $L_3$.

Next we construct a lineage interference graph $\mathcal{I}$ which is an undirected graph. The vertices in the graph represent the lineages. Two vertices are connected by an interference edge only if the live ranges of the lineages represented by them definitely overlap, *i.e.,* they satisfy condition C1. The lineage interference graph can be colored using a heuristic graph coloring algorithm [8]. 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 scheduling order of descendent nodes in the DDG, the HRB computed is a near-optimal solution. The lineage interference graph for the motivating example is shown in Figure 2(c). Its HRB is 3.

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

### 3.3 Instruction Sequence Generation

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

$$A \quad = \quad \{(a, R_1); (b, R_1); (f, R_1); (c, R_2); (d, R_2); (e : R3), (g : R_3)\}$$

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 sequencing algorithm uses the augmented DDG with sequencing edges. The availability of register needs to be checked only if node $v_i$ is the first node in its lineage. Otherwise, the predecessor of $v_i$ would pass the register assigned to it, which would be free

when $v_i$ is ready to be listed. Note that the introduction of sequencing edges in the augmented DDG ensures that $v_i$ is the *last use* of its predecessor. The details of the sequencing algorithm are presented in Appendix A.

Applying the sequencing algorithm to our motivating example, first we list node $a$. The scheduling of node $a$ causes nodes $c$, $d$, and $e$ to be added to the Ready List. Since register $R_2$ is available, node $d$ can be listed next. Now the only available register is $R_3$. Hence we have to list node $e$. Even though node $c$ is ready, it cannot be listed until Register $R_2$ becomes free, which happens after node $g$ is listed. The nodes $c$, $b$, $f$, and $h$ are listed subsequently in that order. The instruction sequence requires only three registers as shown in Figure 1(c).

Unfortunately, the above sequencing algorithm could result in a deadlock due to two reasons. First, the order of listing of two nodes belonging to two different lineages that are assigned the same color may result in a deadlock. For example, in our motivating example a deadlock occurs when node $c$ is listed before $d$ (both share the same register $R2$). Such an ordering between nodes that share the same register, introduces a precedence edge from node $f$, last use of $[c, f)$, to node $d$, which causes a cycle in the augmented DDG. Note, however, such a cycle is not introduced when the list node $d$ before $c$. We refer to these deadlocks caused by wrong ordering of nodes as *avoidable deadlocks*. As the name suggests, these deadlocks can be avoided by checking the different orderings of lineages that share the same color, although the complexity of such an approach is higher.

The second kind of deadlock referred to as *unavoidable deadlock* occurs 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 gradually increases the HRB by 1. We estimate the performance of our heuristic approach in Section 5.

### 3.4 Heuristics Employed

In this section we discuss some additional heuristics in order to make the heuristic method more efficient.

**Chromatic Adjustment:** During the node listing, if a ready node that has the maximum priority cannot be listed due to the non-availability of the register assigned to it in the lineage interference graph, we choose another available register and assign it to the node and its lineage. This chromatic adjustment may imply in further changes of register assignment for the lineages whose first nodes have not yet been listed. We follow the same approach when our scheduling approach is unable to proceed due to a deadlock. However, when we reach a point in which there are no register available, we increase the number of registers allowed as before.

**Height Reduction (HR):** For the sake of computing the height of the nodes, a sequence of nodes in which each node has a single predecessor and a single successor, is abstracted as

a single node. The intuition is that, independent of the sequence length, all these nodes can share the same register without any influence in the rest of the DDG.

**Affinity (A):** This heuristic uses *affinity* to prioritize nodes from the ready list during the generation of the instruction sequence. The motivation for favoring the inclusion of a node that has affinity with nodes recently included in the instruction sequence is to complete the listing of a region of the graph that is likely to share registers, and thus release these registers, before starting listing nodes in unrelated regions of the DDG. The effect of this heuristic is similar to listing all the nodes in one subtree before moving to others in the OCG algorithm [23], but it has the advantage of working well with list scheduling.

The affinity heuristic helps to make an appropriate choice in listing nodes from the ready list that avoids certain deadlocks. For instance, in our motivating example, once nodes $a$ and $e$ are listed, the affinity of node $d$ is higher than that of $c$ and hence it gets listed before $c$. Thus, it avoids the deadlock that would have resulted, had node $c$ got listed before $d$.

The chromatic adjustment heuristic is incorporated in our base method and we refer to it as our base Heuristic Method (HM). When the height reduction heuristic is applied to HM we refer to it as HM+HR method. Lastly, adding affinity to HM+HR results in HM+HR+A. The effect of these heuristics and the performance of the different methods (HM, HM+HR, and HM+HR+A) are discussed in Section 5.

# 4 Exact Approach to MRS 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,

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

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 $u \neq v$. This relation can be also expressed as $|f_u - f_v| > 0$. We represent this relation as an integer linear constraint using the technique proposed by Hu in [13]. 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. Note that for any two nodes $u$ and $v$, such

that there exists a directed path from $u$ to $v$, we know that $f_v$ is always greater than $f_u$, and hence we can replace the above inequalities simply by $f_v - f_u > 0$.

Next we derive conditions that ensure that the instruction sequence obeys true data dependences. If there exists an arc $(u, v)$ in the DDG[3], then clearly node $u$ must be listed ahead of $v$. That is

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

Lastly, we need to formulate the condition that if the live ranges of two variables overlap, they cannot share the same register. In order to do this, we associate an integer $g_u$ with each node $u$ that has at least one descendent. Thus, if $g_u$ and $g_v$ are the values associated with two nodes $u$ and $v$, then $g_u \neq g_v$ if the live ranges of $u$ and $v$ overlap. To simplify the discussion, let us assume that both $u$ and $v$ have only one successor, $u_s$ and $v_s$ respectively. Then the live range of $u$ is the semi-open interval $[f_u, f_{u_s})$ and that of $v$ is $[f_v, f_{v_s})$. If these live ranges are overlapping, then clearly $g_u \neq g_v$. This condition is equivalent to saying that one or more of the following sets of conditions must be true.

$$g_u - g_v > 0 \tag{5}$$
$$g_v - g_u > 0 \tag{6}$$
$$f_{u_s} \leq f_v \tag{7}$$
$$f_{v_s} \leq f_u \tag{8}$$

If either Inequality 5 or Inequality 6 is true, then the nodes are assigned different registers and hence it does not matter whether their live ranges overlap or not. However, if both Inequalities 5 and 6 are false, *i.e.*, $g_u = g_v$, then we need to ensure that the live ranges of $u$ and $v$ do not overlap. Inequality 7 states that the only use of $u$ happens before $v$ and hence $v$ can reuse the register associated to $u$. The equality represents the case in which $u_s$ is same as $v$. Similarly for Inequality 8.

To state that one of the Inequalities 5 – 8 hold in a linear programming formulation, we once again use the technique suggested by Hu [13]. We use four 0-1 variables, one for each inequality ( 5 – 8), namely $d^0_{u,v}$, $d^1_{u,v}$, $d^2_{u,v}$, and $d^3_{u,v}$, These variables are related to Inequalities 5 – 8 in the following way:

$$g_u - g_v > -n * d^0_{u,v} \tag{9}$$
$$g_v - g_u > -n * d^1_{u,v} \tag{10}$$
$$f_v - f_{u_s} >= -n * d^2_{u,v} \tag{11}$$
$$f_u - f_{v_s} >= -n * d^3_{u,v} \tag{12}$$

In order for one of the four conditions to be true, it is required that the sum $d^0_{u,v}+d^1_{u,v}+d^2_{u,v}+d^3_{u,v}$ be equal to the number of constraints minus how many of them need be true. That is,

$$d^0_{u,v} + d^1_{u,v} + d^2_{u,v} + d^3_{u,v} = 4 - 1 = 3 \tag{13}$$

[3]In the ILP formulation, we do not consider lineages, and hence the notion of augmented DDG is also not used. Thus, throughout this section, by DDG we refer only to the original DDG.

The condition (Inequality 13) is derived for (exactly) one of the four Inequalities (9 – 12) to be true. Does Inequality 13 hold, when more than one of the four inequalities are true? This amounts to asking, if the live ranges of $u$ and $v$ do not overlap, do Inequalities 9 – 13 force $u$ and $v$ to share the same register? The answer is no, as explained in the following example. For instance, if $f_u > f_{v_s}$, then $f_{u_s} > f_v$, therefore clearly $d^2_{u,v}$ must be 1 and $d^3_{u,v}$ can be 0 or 1. Further, if $g_u \neq g_v$ and $g_u > g_v$, then $d^1_{u,v}$ must be 1 and $d^0_{u,v}$ can be 0 or 1. In this case, to satisfy Inequality 13, we can choose either $d^3_{u,v}$ or $d^0_{u,v}$ (but not both) to be 1. Thus, even when the live ranges of $u$ and $v$ are non-overlapping, the above Inequalities do not necessarily restrict that they be assigned the same register[4].

There is one complication: if $u$ or $v$ have more than one successor then we need to ensure that Inequalities 11 and 12 hold for all successors of $u$ and $v$. We can achieve this by including one condition (Inequality 11) for each successor of $u$ , but with the same $d^2_{uv}$. Similarly Inequality 12 is written for each successor of $v$. Inequalities 9 – 12 must hold for all pairs of nodes $u$ and $v$ that require a register to be associated with them.

Finally, the objective function of the ILP formulation is to minimize the maximum of $g_u$ assigned to any node $u$. This min-max objective function can be specified as

$$\textbf{minimize} \quad z \quad \text{such that}$$

$$g_u \leq z \quad \text{for all} \quad u \text{ in } V' \tag{14}$$

where $V'$ is the set of nodes which have a successor and hence need to be assigned registers. Thus the ILP problem is to minimize $z$ subject to Inequalities 1 – 4, 9 – 12, 13, and 14.

# 5 Experimental Results

In this section, we outline our experimental framework, and report the experimental results.

## 5.1 Experimental Framework

We have implemented our heuristic methods (HM, HM+HR, and HM+HR+A) and the optimal method based on integer programming formulation (ILP method) in the MOST framework [3]. To solve the integer linear program, we used the commercial solver *CPLEX*. For comparison reasons, we have also implemented the Sethi-Ullman algorithm for the OCG problem as proposed in [23]. These implementations were tested on 852 DDGs extracted from `SPEC` (`integer` and `floating point`), `linpack`, `livermore`, and the `NAS kernels` benchmarks. These DDGs were extracted with a highly-optimizing research compiler. The DDGs varied widely in size with a median of 10 nodes, a geometric mean of 12 nodes, and an arithmetic mean of 19 nodes per DDG.

[4]However, to achieve minimum register sequence, we want the nodes $u$ and $v$ to share the same register when their live ranges are non-overlapping. But this is achieved through the objective function of the ILP formulation.

### 5.2 Performance of Heuristic Methods

In this section we report the performance of the various heuristic methods.

#### 5.2.1 Optimality of Heuristic Solutions

First we compare the three heuristic methods with the ILP approach. Out of the 852 DDGs, in 671 benchmarks both the ILP and the heuristic methods found a solution. In the remaining DDGs the ILP method failed to obtain a solution in a reasonable computation time. HM found an optimal solution for the MRIS problem in 665 cases. In the remaining 6 cases, HM produced instruction sequences that, on the average used 1.3 extra registers. The maximum difference between the heuristic and the exact solutions was 2 registers. The HM+HR method obtained optimal solution in 665 cases, and the HM+HR+A method in 667 cases. Thus, our best heuristic method missed an optimal solution only in 4 cases out of 671. In those cases where the heuristic methods found a sub-optimal solution, the average additional registers used is 1.33 (HM), 1.17 (HM+HR), and 1.25 (HM+HR+A).

The coloring of the lineage interference graph provides an heuristic register bound (HRB). The value of the HRB obtained is affected by the height reduction technique. Table 1(a) shows the number of DDGs in which the HRB underestimates, overestimates, or is equal to the minimum register requirement (MRR) computed by the ILP solution. In all 16 DDGs in which the MRR is overestimated, it is overestimated by one register by both HM and HM+HR. HM overestimates MRR by 1 register in one case and by 2 registers in the other, while HM+HR overestimates the MRR by 1 register in all three DDGs. Notice that underestimations are often fixed by the sequencing algorithm. All the results presented in Table 1(a) are for small DDGs with a low MRR for which the ILP method can find a solution.

| Estimation of HRB | Heuristic Methods | |
|---|---|---|
| | HM | HM+HR |
| HRB = MRR (tight) | 653 | 652 |
| HRB < MRR (underestimation) | 16 | 16 |
| HRB > MRR (overestimation) | 2 | 3 |

(a) Comparison of HRB with MRR for Smaller DDGs

| Method | No. of DDGs where HRB = Actual Reg. Required | HRB < Actual Reg. Required | | |
|---|---|---|---|---|
| | | No. of DDGs | Avg. Register Increase | Max. Register Increase |
| HM | 88 | 93 | 1.9 | 9 |
| HM+HR | 122 | 59 | 2.0 | 9 |
| HM+HR+A | 121 | 60 | 1.8 | 7 |

(b) Comparison of HRB with Actual Register Requirement for Larger DDGs

Table 1: Tightness of Heuristic Register Bound

### 5.2.2 Effectiveness of Heuristic Methods in Larger DDGs

How good is our heuristic solution for the 181 DDGs for which the ILP approach failed to find an optimal solution? The first column of Table 1(b) reports the number of DDGs for which an instruction sequence was found that used HRB registers. The table also shows the number of DDGs for which the instruction sequence requires more registers than HRB, the average register increase and the maximum register increase for each method.

Our heuristic method (HM) constructed an instruction sequence that used no more than HRB registers for 88 DDGs. In the remaining 93 DDGs, the modified list scheduling method resulted in a deadlock (avoidable or unavoidable) and resolved the deadlock by using additional registers. The average increase in the number of registers in these cases is 1.9, with the maximum increase as high as 9 registers. Although the MRR is not available for these DDGs, the small number of cases in which HRB overestimates MRR when the ILP approach terminates serve as an indication of the quality of the solution for these larger DDGs.

It can be seen that the HM+HR and HM+HR+A methods performed significantly better than HM for larger DDGs. This table shows that the use of affinity in the sequencing algorithm has resulted in a smaller number of extra registers being used.

### 5.2.3 Comparison with OCG Method

Next we compare the solutions produced by our heuristic methods with those generated by Sethi-Ullman's OCG algorithm (S-U). Once again we only consider the larger DDGs where the ILP method failed to find a solution (181 cases). The reason for this is that in 667 DDGs out of the remaining 671 cases, we already got the optimal solution.

Table 2 summarizes the comparison of the three heuristic methods with S-U in terms of the number of registers used. For instance, when the height reduction and the affinity heuristics are used (HM+HR+A) are employed, our heuristic method uses the same number of registers as S-U for 75 DDGs. For 6 DDGs our method uses on the average 1.8 more registers, but never uses more than 4 extra registers. On the other hand, for 100 DDGs, our heuristic method uses on average 2.6 less registers than S-U, and in some cases S-U uses as many as 8 extra registers. The numbers in the table show that all three heuristic methods perform better than S-U.

## 5.3 Summary of Main Results

The main results can be summarized as follows.

- In terms of optimality, the heuristic methods performed extremely well. Out of the 671 DDGs where both the heuristic and the ILP methods have reached a solution, the heuristic methods found an optimal solution in a overwhelming majority (99.4% or 667 out of 671) of the DDGs,

| Description | HM | | HM+HR | | HM+HR+A | |
|---|---|---|---|---|---|---|
| | S-U Better | HM Better | S-U Better | HM+HR Better | S-U Better | HM+HR+A Better |
| No. of DDGs | 25 | 100 | 18 | 104 | 6 | 100 |
| Avg. Incr. in Registers | 1.4 | 2.3 | 1.2 | 2.6 | 1.8 | 2.6 |
| Min. Incr. in Registers | 1 | 1 | 1 | 1 | 1 | 1 |
| Freq. of Min. Increase | 16 | 25 | 14 | 31 | 4 | 28 |
| Max. Incr. in Registers | 2 | 7 | 2 | 7 | 4 | 8 |

Table 2: Performance of Heuristic Methods vs. Sethi-Ullmann on 181 Large Loops

- In 181 DDGs where the ILP method failed to obtain an optimal solution the heuristic methods quickly reached a good solution. For the 60 DDGs in which our heuristic method (HM+HR+A) required more registers, on average, it used only 1.8 registers above HRB, which itself is found to be a good bound.
- Lastly, a comparison with the Sethi-Ullman's algorithm reveals that our approach performed better (in terms of number of registers) in 100 out of the 181 DDGs, equal in 75 DDGs, and worse in only 6 DDGs. Moreover, the register requirements of our heuristic solution was less by 2.6 registers on the average per DDG.

# 6 Related Work

Instruction scheduling [10, 17] and register allocation [2, 7, 8, 9, 17, 18, 20, 28] are important phases in a high performance compiler. The ordering of these phases and its implications on the performance of the code generated have been studied extensively for in-order issue superscalar processors and Very Long Instruction Word (VLIW) processors. In such processors it is often necessary to expose enough instruction-level parallelism even at the expense of increasing the register pressure and, to some extent, the amount of spill code generated. Integrated techniques that try to minimize register spills while focusing on exposing parallelism were found to perform well [6, 5, 16, 19].

The Minimum Register Instruction Sequence (MRIS) problem studied in this paper is different from the traditional register allocation problem [2, 7, 8, 9, 17, 18]. 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* [2, 10, 11, 17, 29]. In the traditional instruction scheduling problem, the main objective is to minimize the total time (length) of the schedule, taking into account the execution latencies of each operation (instruction) in the DDG and the availability of function unit resources. This is in contrast to the MRIS problem, where only the true dependence constraints are observed.

The MRIS problem is closely related to the *optimal code generation (OCG)* problem [2, 22, 21]. For the case in which the dependence graph is a tree, an optimal algorithm to solve the OCG problem exists. For a general DAG, the problem is known to be NP-Complete since 1975 [22]. An important difference between traditional code generation methods and our MRIS problem is that the former emphasizes on reducing the code length (or schedule length) while the latter minimizes the number of registers. Further the code generation problem is specifically for generating a minimum schedule length code for a given 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, they (Berson, Gupta and Soffa) have used *register reuse dags* for measuring the register pressure [5]. A register reuse dag is similar to a lineage discussed in this paper. They have evaluated register spilling and register splitting methods for reducing the register requirements in the URSA method.

Lastly, the lineage formation and the heuristic list scheduling methods proposed in this paper are major improvements over, respectively, the chain formation and the modified list scheduling method discussed in [12]. The chain formation method allocates, at least conceptually, one register with 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. The application of the chain formation algorithm on our motivating example would result in chains $c1 = \{a, b, f, h\}$, $c2 = \{a, c, f\}$, $c3 = \{a, d, g\}$, $c4 = \{a, e, g, h\}$, The chain interference graph for this is a fully connected graph with 4 vertices and hence the HRB is 4. Whereas the minimum register requirement and the HRB in our lineage interference method is 3. Thus, 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. In particular, the affinity heuristics facilitates the listing the nodes in a subtree, before other subtrees are explored. As discussed in Section 3.4, this heuristic not only avoids certain deadlocks but also reduces the extra registers used to avoid deadlocks. Such simple, but effective heuristics are not present in the original method [12].

# 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 the requirements in modern out-of-order issue processors and low-power embedded 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. Register spills lead to memory accesses which are expensive in terms of power dissipation in low-power embedded processors.

We proposed two solution methods for the MRIS problem. The first solution uses a heuristic approach, and is based on the notion of lineage formation, lineage interference graph and a modified and efficient list scheduling method. The second approach is based on an elegant integer linear programming formulation. The proposed methods have been implemented in the MOST framework. Our experimental results demonstrate that the combination of the notion of instruction lineages with the concepts of height reduction and node affinity forms a powerful framework that obtained the optimal solution for all the small graphs in which our ILP formulation could find a solution. Our comparison with the Sethi-Ullman algorithm for the OCG problem showed that our heuristic solution performed better than their algorithm in 55% (75 out of 181) of the graphs, and worse only in 3% of the graphs.

## References


[1] G. Araujo and S. Malik. Optimal code generation for embedded memory non-homogeneous register architectures. In *Proc. of the 1995 Intl. Symp. on System Synthesis*, pages 36–41, 1995.

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

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

[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] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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`)

[13] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, Nov. 1961.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] M. Poletto and V. Sarkar Linear scan register allocation *ACM Trans. of Programming Langauges and Systems.* (to appear).

[21] 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.

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

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

[24] 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.

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

[26] V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy: An overview. In *IEEE Symposium on Low-Power Electronics*, pages 38–39, 1994.

[27] H. Tomiyama, T. Ishihara, A. Inoue, and H. Yasuura. Instruction scheduling for power reduction in processor-based system design. In *Proceeding of Design Automation and Test in Europe (DATE'98)*, February 1998.

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

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

# A Details of Heuristic Algorithms

In this section we present the details of the lineage formation algorithm and the sequencing algorithm discussed in Section 3.1 and Section 3.3.

The lineage formation algorithm is presented in Figure 3. The instruction sequence generation method is shown in Figure 4. The sequencing algorithm takes as inputs $G'$, the DDG augmented with sequencing edges; $L$, a list of lineages obtained from the lineage formation algorithm; $A$, the register assignment for the nodes from the coloring of the lineage interference graph; and $N$, the HRB. In the algorithm presented in Figure 4, FirstNode($v_i$) is true if the node $v_i$ is the first node of a lineage, and LastNode($v_i$) is true if $v_i$ is the last node in a lineage. 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 $v_i$ would pass the register assigned to it, which would be free when $v_i$ is ready to be listed because the algorithm is using the augmented DDG with the sequencing edges. Note that the introduction of sequencing edges in the augmented DDG ensures that $v_i$ is the *last use* of its predecessor.

```
LineageFormation(V, E)
1.   mark all nodes in the DDG as not in any lineage
2.   compute the height of every node in the DDG
3.   while there is a node not in any lineage do
4.        recompute height ← false
5.        v_i ← highest node not in lineage
6.        start a new lineage containing v_i
7.        mark v_i as in a lineage
8.        while v_i has a descendent do
9.             v_j ← lowest descendent of v_i that is not in any lineage
10.            for each descendent v_k ≠ v_j of v_i do
11.                 add sequencing edge from v_k to v_j
12.            endfor
13.            if v_i has multiple descendents
14.                 recompute height ← true
15.            endif
16.            add v_j to lineage
17.            if v_j is already marked as in a lineage
18.              break;
19.            mark v_j as in a lineage
20.            v_i ← v_j
21.       end while
22.       if recompute height = true
23.            recompute the height of every node in the DDG
24.       endif
25.  end while
```

Figure 3: Lineage Formation Algorithm

```
Scheduling(G', L, A, N)
1.    ReadyList ← {(v_i, R_j) such that v_i has no predecessors }
2.    RegAvailable ← {R_1, R_2, ..., R_N}
3.    while ReadyList ≠ ∅ do
4.         for each node v_i in the ReadyList in decreasing height order do
5.              if (FirstNode(v_i) = False) or (((v_i, R_j) ∈ A) and (R_j ∈ RegAvailable))
6.                   Remove R_j from RegAvailable
7.                   Remove (vi, R_j) from ReadyList
8.                   Schedule(v_i, R_j)
9.                   Add to the ReadyList all successors of v_i
                       that have all its predecessors listed
10.                  if LastNode(v_i) and (v_i, R_j) ∈ A
11.                    Return R_j to RegAvailable
12.                  endif
13.             endif
14.        endfor
15.   end while
```

Figure 4: Scheduling Algorithm