Combining Register Allocation and Instruction Scheduling

Rajeev Motwani[1] Stanford University

Krishna V. Palem[2] New York University

Vivek Sarkar[3] IBM Software Solutions Division

Salem Reyen[4] New York University

Abstract

We formulate combined register allocation and instruction scheduling within a basic block as a single optimization problem, with an objective cost function that more directly captures the primary measure of interest in code optimization — the completion time of the last instruction. We show that although a simple instance of the combined problem is *NP-hard*, the combined problem is much easier to solve *approximately* than graph coloring, which is a common formulation used for the register allocation phase in phase-ordered solutions.

Using our framework, we devise a simple and effective heuristic algorithm for the combined problem. This algorithm is called the (α, β)-Combined Heuristic; parameters α and β provide relative weightages for controlling register pressure and instruction parallelism considerations in the combined heuristic. Preliminary experiments indicate that the combined heuristic yields improvements in the range of 16-21% compared to the phase-ordered solutions, when the input graphs contain balanced amount of register pressure and instruction-level parallelism.

[1]Permanent address: Department of Computer Science, Stanford University, Stanford, CA 94305-2140; +1 (415) 723–6045; rajeev@cs.stanford.edu. Supported by an Alfred P. Sloan Research Fellowship, an IBM Faculty Development Award, an OTL grant, and NSF Young Investigator Award CCR-9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

[2]Permanent address: Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012-1185; +1 (212) 998–3512; palem@cs.nyu.edu

[3]Permanent address: Application Development Technology Institute, IBM Software Solutions Division, 555 Bailey Avenue, San Jose, CA 95141; +1 (408) 463–5660; vivek_sarkar@vnet.ibm.com

[4]Permanent address: Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012-1185; m-sr0069@cs.nyu.edu

1 Introduction

Modern optimizing compilers contain several optimization phases, including *register allocation* and *instruction scheduling* which have received widespread attention in past academic and industrial research. It is now generally recognized that the separation between the register allocation and instruction scheduling phases leads to significant problems, such as poor optimization for cases that are ill-suited to the specific phase-ordering selected by the compiler, and additional conceptual and software-engineering complexities in trying to adjust one phase to take into account cost considerations from the other. The goal of our research is to address these problems by building on past work and combining register allocation and instruction scheduling into a single phase. Henceforth, we refer to our model of these problems as CRISP (for Combined Register allocation and Instruction Scheduling Problem).

The main contributions of this paper are as follows:

- A *formulation* of combined register allocation and instruction scheduling within a basic block as a single optimization problem, with an objective cost function that more directly captures the primary measure of interest in code optimization — the completion time of the last instruction.

  This formulation has new underlying *data structure abstractions* which lend themselves to designing the heuristic algorithm mentioned below. This heuristic algorithm could not have been formulated in the context of an existing phase-ordered solution.

- A result showing that a simple instance of the combined problem (single register, no latencies, single functional unit) is *NP-hard*, even though instruction scheduling for a basic block with 0/1 latencies on a single pipelined functional unit is not NP-hard (for a given register allocation) and register allocation for a basic block is also not NP-hard (for a given schedule).

- A result showing that the combined problem is much easier to solve *approximately* than graph coloring, which is a common formulation used for the register allocation phase in phase-ordered solutions[5].

- A simple and effective *heuristic algorithm for the combined problem.* This algorithm is called the (α, β)-Combined Heuristic; parameters α and β provide relative weightages for controlling register pressure and instruction parallelism considerations in the combined heuristic.

- *Experimental results* comparing the heuristic algorithm for the combined problem with two phase-ordered solutions (register allocation followed by instruction scheduling, and instruction scheduling followed by register allocation) on randomly generated graphs.

  Preliminary results show that the combined heuristic yields improvements in the range of 16-21% compared to the phase-ordered solutions, when the input graphs contain balanced amount of register pressure and instruction-level parallelism. If there is an imbalance, then the performance of the combined heuristic approaches that of one of the phase-ordered solutions.

Most of the results in this paper apply to combining register allocation and instruction scheduling within a single basic block. Our ultimate research goal is to extend these results to a more global context. However, it is important to first get a detailed understanding of the combined problem and its characteristics in the context of a single basic block, since such a combined approach has not been studied earlier in the literature. This is what the contributions in our paper provide. In addition, there are cases of practical interest that arise in which large basic blocks are encountered (typically, after code transformations such as inline expansion

[5]Graph coloring is used as a common formulation for *global* register allocation, i.e., register allocation across *multiple* basic blocks. In practice, most compilers that use graph coloring do so uniformly for both local and global register allocation, rather than using an optimal algorithm for register allocation within a single basic block (for a given schedule).

of calls and unrolling of loops) with sufficient register pressure to warrant the use of the combined solution for a single basic block presented in this paper.

The rest of the paper is organized as follows. Section 2 uses an example to discuss phase-ordered solutions to the register allocation and instruction scheduling problems for a single basic block, and to illustrate the problems that arise in phase-ordered solutions. Section 3 presents our basic formulation of the combined register allocation and instruction scheduling optimization problem for a single basic block. Section 4 starts by showing that the combined problem is NP-hard. This section also shows that the combined problem is essentially in low-level approximation complexity class ($\overline{\text{MAX SNP}}$) and thus easier to solve approximately than graph coloring and many other NP-hard problems. Section 5 presents a simple polynomial-time heuristic for our formulation of the combined register allocation and instruction scheduling problem. Section 6 outlines some important extensions to the basic formulation of the combined problem presented in Section 3. Our results apply to this extended formulation as well; these extensions were excluded from the basic formulation so as to simplify the technical discussion in the earlier sections. Section 7 contains experimental results comparing phase-ordered solutions for register allocation and instruction scheduling with our approximate solution to the combined problem. Section 8 discusses related work, and Section 9 wraps up with conclusions and an outline of future work.

2 Phase-ordered Solutions

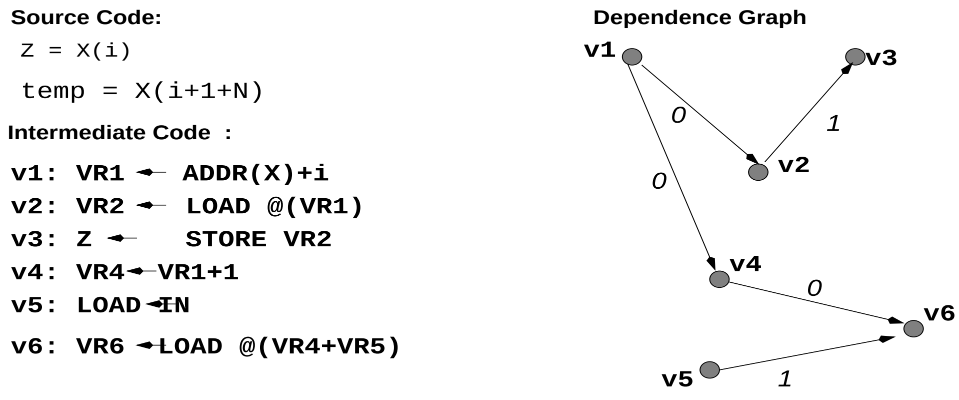

Figure 1: A Basic Block and Dependency Graph

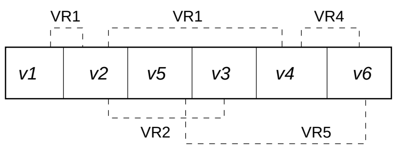

Figure 2: Instruction Scheduling followed by Register Allocation

In this section, we use an example to discuss phase-ordered solutions to the register allocation and instruction scheduling problems for a single basic block, and to illustrate the problems that arise in phase-ordered solutions. In our discussion, we informally use terms such as “value range”, “spills”, “schedule”, and “completion time” and postpone their formal definitions to Section 3.

The example that we will use is shown in Figure 1. To simplify the discussion, we chose an example with a small basic block (6 instructions) in the context of a single two-stage pipeline and two available

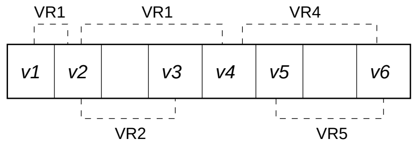

Figure 3: Register Allocation followed by Instruction Scheduling

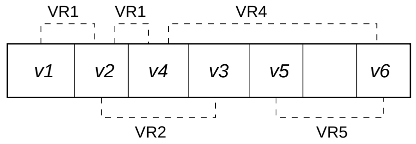

Figure 4: Combined Register Allocation and Instruction Scheduling

physical registers. The trends discussed for this example can also be observed for larger problem sizes, as reported in Section 7.

Instruction Scheduling followed by Register Allocation A common phase ordering used in industry compilers is to perform instruction scheduling before register allocation. This phase ordering gives priority to exploiting instruction-level parallelism over register utilization, which, if one had to choose a specific phase ordering, is quite likely the right choice for modern RISC processors.

It is well known that any phase-ordered solution can generate poor code for cases that are ill-suited to that phase-ordering. Figure 2 shows the schedule generated by this approach. It is an optimal schedule with no idle slots and completion time = 6 cycles. The instruction scheduler cleverly hides the unit latency in each memory instruction by pushing v_3 further away from v_2 and by pulling v_5 further away from v_6. However, the problem occurs later during register allocation where we see that the instruction schedule has stretched out value ranges $(\text{vr2}, \sigma_1(v_2), \sigma_1(v_3))$ and $(\text{vr5}, \sigma_1(v_5), \sigma_1(v_6))$ thus increasing register pressure. As a result, the bandwidth at time 3 is three, and this phase-ordered solution is forced to spill one value range.

Register Allocation followed by Instruction Scheduling The converse approach is to perform register allocation before instruction scheduling. This phase ordering gives priority to utilizing registers over exploiting instruction-level parallelism. It was a common approach used in early compilers when the target machine had only a small number of available registers. Figure 3 shows the schedule and register allocation generated by this approach. The register allocation requires no spills. The register allocator cleverly avoids spilling a value range by allocating virtual registers vr2, vr5 to one physical register and vr1, vr4 to the second physical register. However, the problem occurs during instruction scheduling when we see that an idle slot is created in the schedule between v_2 and v_3 and between v_5 and v_6 due to extra register dependences. As a result, the completion time of the schedule is now 8 cycles, with no spills.

Combined Register Allocation and Instruction Scheduling This approach pays attention to issues relating to both spilling and loss of instruction-level parallelism.

Figure 4 shows the schedule and register allocation generated by this approach. This solution chose to move v_5 closer to v_6, but did not move v_3 closer to v_2. As a result, there is one idle slot in the schedule and no value ranges need to be spilled.

3 CRISP Model for a Single Basic Block

Let $V = \{v_1, \ldots, v_n\}$ be the set of *instructions* in the basic block. $DG \subseteq V \times V$ is the *data-dependence graph*, in which an edge (v_i, v_j) enforces the constraint that instruction v_j must *start* execution after instruction v_i *completes* in any *feasible schedule*. Each instruction is labeled with an *execution time*, $t(v_i)$. Each dependence edge $(v_i, v_j) \in DG$ is labeled with an inter-instruction *latency*, $l(v_i, v_j) \geq 0$; this means that instruction v_j must start at least $l(v_i, v_j)$ cycles after the completion time of instruction v_i. In this basic formulation, we assume that all instructions have the execution time = 1 cycle, and all edge latencies are either 0 or 1. These assumptions are primarily for ease of exposition; extensions to this basic formulation are discussed in Section 6.

A *schedule*, σ, specifies the start time $\sigma(v_i) \geq 0$ for each instruction v_i. The *completion time of schedule* σ (also referred to as the *makespan*) is simply $T(\sigma) = \max_i\{\sigma(v_i) + t(v_i)\}$, the completion time of the last instruction to complete in σ. All schedules must obey the following condition, which enforces the data-dependence and the inter-instruction latency constraints:

$$\sigma(v_j) \leq \sigma(v_i) + t(v_i) + l(v_i, v_j) \quad \text{for each data-dependence edge } (v_i, v_j) \in DG$$

To model register usage, we define $use(v_i)$ and $def(v_i)$ to be the set of *virtual registers* read and written by instruction v_i. We assume that each virtual register r satisfies the static-single-assignment rule: r must belong to exactly one $def(v_i)$ set. Thus, a single program variable is separated into multiple virtual registers, one for each *def*. We define $prod(r)$ to be the *producer instruction* for virtual register r so that $prod(r) = v_i \;\Rightarrow\; r \in def(v_i)$, and we define $cons(r)$ to be the set of *consumer instructions* for virtual register r so that $v_i \in cons(r) \;\Rightarrow\; r \in use(v_i)$. We assume that the dependence graph DG includes all data flow dependences on virtual registers i.e. for each virtual register r, $v_i = prod(r) \wedge v_j \in cons(r) \Rightarrow (v_i, v_j) \in DG$. DG may include other dependences as well, e.g., a dependence between two memory operations that may potentially interfere.

For a single basic block, we do not need to worry about the case of multiple *def*s reaching the same *use* (as identified by ϕ-functions in static-single-assignment form). Since the basic-block contains straight-line code with no intervening control flow, there must be a unique *def* value that reaches a specific *use*[6]. We also assume that all virtual register-register copy operations are coalesced within the basic block before invoking the instruction scheduling and register allocation phases of the compiler.

It is also convenient to define $k = \max_{v_i \in V}\{|use(v_i)| + |def(v_i)|\}$ to be the maximum number of virtual registers accessed by a single instruction. Typically, $k \leq 4$.

For a given schedule σ, we define a *value range of virtual register* r to be the triple $(r, \sigma(v_i), \sigma(v_j))$, such that

- $\sigma(v_i) < \sigma(v_j)$, instruction v_i is scheduled before instruction v_j in σ
- v_i is either a producer or a consumer of virtual register r
- v_j is a consumer of virtual register r
- there is no intervening instruction v_k such that $\sigma(v_i) < \sigma(v_k) < \sigma(v_j)$ and v_k is a consumer of virtual register r

Note that a virtual register can have multiple value ranges: the first being from the def to the first use, the next being from the first use to the second use, and so on. This is different from a *live range*, which traditionally extends from the first def to the last use. Value ranges represent the finest granularity for splitting live ranges.

Let *VR*(σ) be the set of value ranges (triples) over all virtual registers in schedule σ. The *bandwidth* of *VR*(σ) at any time τ, $BW(\sigma, VR(\sigma), \tau)$ is the number of value-ranges in *VR*(σ) that start at some time before τ and end at some time after τ in schedule σ. Note that there is no overlap between a value range that

[6] A virtual register that is *live on entry* to a basic block is modeled by adding a def to a dummy source node for the basic block, and a virtual register that is *live on exit* to a basic block is modeled by adding a use to a dummy sink node for the basic block.

ends at instruction v_i and a value range that starts at instruction v_i. This correctly models the hardware's ability to execute an instruction that uses the same physical register as an input and an output, without there being any interference between the two.

Let R be the number of *physical registers* available. Register spills are required if $BW(\sigma, \mathit{VR}(\sigma), \tau) > R$ at any time τ. Let $\mathit{SVR}(\sigma) \subseteq \mathit{VR}(\sigma)$ be the set of *spilled* value ranges and $\mathit{AVR}(\sigma) = \mathit{VR}(\sigma) - \mathit{SVR}(\sigma)$ be the set of *active* (non-spilled) value ranges.

Spilling a value range, (r, τ_1, τ_2), has the effect of forcing the value contained in virtual register r to reside in memory during the time interval, (τ_1, τ_2), thereby avoiding the need for a physical register to hold that value during that time interval. The overhead of spilling a single value range consists of a *store* instruction inserted at time τ_1 and a *load* instruction inserted at time τ_2. For architectures that charge a single cycle overhead for each instruction, the total cost of all spilled value ranges will equal twice the number of spilled value ranges[7].

Schedule σ and spilled-value-range set $\mathit{SVR}(\sigma)$ are said to be *resource-feasible* if and only if they satisfy the following two resource constraints:

1. At any given time τ, the bandwidth for the active value ranges must be no larger than the number of available registers, $BW(\sigma, \mathit{AVR}(\sigma), \tau) \leq R$.

   In this basic formulation, we ignore the (at most k) extra registers required for storing and loading spilled register values. This is not a serious limitation because typically, at any instant of time in the schedule, the number of registers to be spilled is much smaller than the number of available registers. For processors with hardware register renaming, the extra registers are a non-issue.

2. At any given time τ, the number of executing instructions (i.e. instructions with $\sigma(v_i) \leq \tau < \sigma(v_i) + t(v_i)$) must not exceed the number of available functional units M.

   In this basic formulation, we assume that $M = 1$, deferring the discussion of the general case to Section 6. Together with the restriction of 0/1 edge latencies, our basic formulation models a CPU with a single two-stage pipeline as in [25, 26, 4].

Given the above definitions, we formulate the combined register allocation and instruction scheduling problem as follows:

Combined Register Allocation and Instruction Scheduling Problem (CRISP) $\Pi_k(R)$:
Find a resource-feasible schedule σ and spilled-value-range set $\mathit{SVR}(\sigma)$ so as to minimize the overall completion time $C(\sigma) = T(\sigma) + 2|\mathit{SVR}(\sigma)|$, the sum of the completion time of the schedule and twice the number of spilled value ranges.

4 Theoretical Underpinnings of CRISP

In this section, we present a rigorous theoretical study of the CRISP problem $\Pi_k(R)$ as formulated above; the proofs of some our results are omitted. First, we show that the problem (in fact, an extremely restricted version thereof) is NP-hard. We observe that instruction scheduling for a basic block with 0/1 latencies on a single pipelined functional unit is not NP-hard (for a given register allocation) and register allocation for a basic block is also not NP-hard (for a given schedule).

Given that the combined cost measure has not been studied earlier in the literature, it is important to settle the hardness issue first. The rest of this section is devoted to showing that a significant distinction

[7] In some cases, this spill cost can be an overestimate because it may be possible to accommodate a load/store spill instruction in an idle slot of the schedule. It's also possible to avoid executing intermediate spill load/store instructions for a set of contiguous value ranges for the same virtual register.

can be made among NP-hard problems if we change the goal from that of obtaining an *optimal* solution to that of obtaining an *approximate* or *near-optimal* solution. We provide a series of results related to the approximability of CRISP.

4.1 Establishing NP-hardness of CRISP

We show that a restricted version of the CRISP problem $\Pi_k(R)$ is equivalent to an NP-hard problem called *Feedback Vertex Set (FVS)* [14]. This will imply that the generic CRISP problem $\Pi_k(R)$ is also NP-hard. We start by giving a formal definition of the *decision* version of the two problems.

Restricted CRISP (RCRISP): The restriction of $\Pi_1(1)$ to the input instances where the edge latencies are all 0. The objective is to determine whether there is a permutation σ such that $C(\sigma) \leq t$, for a parameter t.

Feedback Vertex Set (FVS) Problem: Given a *directed* graph $G(W, E)$ and a positive integer s, decide whether there is a feedback vertex set $S \subseteq W$ of size at most s, where a feedback vertex set is a set of vertices whose removal (along with incident edges) from G will result in an acyclic graph

The Reduction: Given an instance $G(W, E)$ and s of FVS, we reduce it to an instance of RCRISP as follows. For each vertex $w \in W$, we introduce into the basic block V the following: instructions w_1, w_2, and w_3, and a distinct virtual register r_w, where $def(w_1) = use(w_2) = use(w_3) = \{r_w\}$ and $use(w_1) = def(w_2) = def(w_3) = \emptyset$. In the dependence graph DG, the *only* dependence edges are as follows: for an edge from v to w in G, we add an edge from v_3 to w_2 in DG; further, for each vertex $w \in W$, we add edges from w_1 to both w_2 and w_2. Finally, we define the parameter $t = 3n + 2s$, where n is the number of vertices in G and s is the parameter for the FVS problem. Note that the total number of instructions in the RCRISP instance is $3n$ and so s is a bound on the number of spills allowed for the RCRISP instance.

The following lemma relates the two problems. Recall that a DAG is an directed acyclic graph.

Lemma 1 *The RCRISP instance has a solution σ with 0 spills if and only if the FVS instance is a DAG.*

Proof: Suppose that the graph G is a DAG. Then, we can associate with it an ordering isomorphism $f : W \to \{1, \ldots, n\}$ such that there for any pair of vertices $v, w \in W$, there is an edge in G from v to w *only if* $f(v) < f(w)$. From this, we construct a schedule σ for the RCRISP instance by defining an ordering isomorphism $g : V \to \{1, \ldots, 3n\}$ and then scheduling instruction $v_i \in V$ at time $g(v_i)$. We define g as follows: for each vertex $w \in W$, $g(w_1) = 3f(w) - 2$, $g(w_2) = 3f(w) - 1$, and $g(w_3) = f(w)$. Intuitively, the linear ordering of V is obtained from the linear ordering of W by replacing each vertex $w \in W$ by the sequence of vertices $\langle w_1, w_2, w_3 \rangle$.

We claim that schedule σ respects the dependence graph DG, and this can be verified by considering the two types edges in DG. Suppose there is an edge from v_3 to w_2. Then, there is an edge in G from v to w, implying that $f(v) < f(w)$; hence, $g(v_3) < g(w_2)$ and this verifies that v_3 is scheduled before w_2 by σ. The edges from w_1 to w_2 and w_3 are respected because of the ordering of these three vertices. It can also be verified that the schedule σ meets the register bound $R = 1$ without any spilling, as follows: first, observe that a value range for a virtual register r_w is either $(r_w, 3f(w) - 2, 3f(w) - 1)$ or $(r_w, 3f(w) - 1, 3f(w))$, implying that each value range has an extent of 1; therefore, at most one value range is active at any point in the schedule.

Conversely, suppose we have a schedule σ that does not require any spilling. We claim that it must be the case that for any $w \in W$, the instructions w_1, w_2, and w_3 are scheduled contiguously. Suppose not, then there must be another instruction v_i scheduled between w_1 and either w_2 or w_3. But then, at the time when v_i gets executed, there must be at least two active value ranges (one for r_v and another for r_w), leading to the violation of the register bound $R = 1$ in the absence of any spills. Note that we can assume without loss of generality that the ordering for the w-instructions is $\langle w_1, w_2, w_3 \rangle$ since w_1 must precede the other two, and there is no dependence between w_2 and w_3.

Thus, the schedule σ defines an ordering isomorphism $g : V \to \{1, \ldots, 3n\}$ such that for each vertex $w \in W$, $g(w_1) = g(w_3) - 2$, and $g(w_2) = g(w_3) - 1$. We can now define an ordering isomorphism $f : W \to \{1, \ldots, n\}$ such that for each vertex $w \in W$, $f(w) = g(w_3)/3$. We claim that for any pair of vertices $v, w \in W$, there is an edge in G from v to w only if $f(v) < f(w)$. This is so because then there must be an edge from v_3 to w_2, implying that the v-instructions all get scheduled before the w-instructions and that $3f(v) = g(v_3) < g(w_3) = 3f(w)$. Thus, since G has an ordering isomorphism that respects the edge directions, it must be the case that G is a DAG. □

Using Lemma 1, we are able to prove the desired theorem.

Theorem 1 *There is a polynomial-time reduction from FVS to RCRISP, and hence RCRISP is NP-hard.*

Proof: It is clear that the reduction from FVS to RCRISP can be computed in polynomial time (in fact, time quadratic in the size of G), and so we focus on the verification of the correctness of the reduction. For the latter, we need to show that the FVS instance has a feedback vertex set S of size at most s if and only if the RCRISP instance has a schedule σ of total cost $C(\sigma) \leq t = 3n + 2s$. Note that a schedule σ has cost $C(\sigma) = 3n + 2N$ where N is the number of spills. Thus, we are required to show that G has a feedback vertex set of size at most s if and only if the RCRISP instance has a schedule with at most s spills.

Suppose that the FVS instance has a feedback vertex set S of size s. Consider the subgraph $H = G[W \setminus S]$ induced by the set of vertices in G *not* belonging to S. The graph H is a DAG by definition of a feedback vertex set S and so, by Lemma 1, the instructions corresponding to the vertices in H can be scheduled without any spilling. Let π be a no-spill schedule (respecting the register bound $R = 1$) for these instructions obtained from vertices in $W \setminus S$. Observe that this schedule must respect all the precedence relations between those instructions, including those obtained by transitivity from the instructions corresponding to the vertices contained in S; however, by the form of our reduction, each instruction either has predecessors or successors, but not both, and so there are no transitively induced precedences to be handled.

We now extend π to a schedule for all instructions in the RCRISP instance, introducing exactly s spills in the process. Consider any vertex $w \in S$ and the corresponding instructions w_1, w_2, and w_3. In σ, we schedule w_1 and w_2 contiguously *before* all the instructions in π, and we schedule w_3 *after* all the instructions in π; further, the value range for r_w extending from w_2 to w_3 is spilled. The exact ordering between the instructions scheduled before or after the schedule π is immaterial from the point of view of precedence edges. Finally, note that putting each pair of instructions w_1 and w_2 next to each other means that the value range between them does not need to be spilled. Clearly, we have obtained a valid schedule where the number of spills is exactly the number of vertices in S, i.e., s.

Conversely, suppose now that the RCRISP instance has a valid schedule σ with exactly s spills. Let $S \subseteq W$ be the set of vertices in G corresponding to all virtual registers which are spilled in σ; note, it doesn't matter whether only one or both of the value ranges of a virtual register corresponding to a vertex were spilled and therefore $|S| \leq s$. Consider now the schedule π obtained from σ by removing all instructions corresponding to the vertices in S; clearly, π is a no-spill precedence-respecting schedule for the remaining instructions. In other words, the instructions corresponding to the vertices in $W \setminus S$ have a no-spill schedule (π) respecting the precedences in the induced subgraph $H = G[W \setminus S]$. From Lemma 1, we obtain that H is a DAG and hence S is a feedback vertex set for G of size at most s. □

4.2 Approximating NP-hard Optimization Problems

A standard technique for dealing with an NP-hard optimization problem is to relax the requirement of finding an optimal solution, and instead settle for a near-optimal solution. This is the basic motivation behind the development of the area of approximation algorithms [24]. Formally, let Π be an NP-hard optimization problem and I denote an input instance. We define $OPT(I)$ as the *value* of the optimal solution for the instance I. A suboptimal solution $S(I)$ is said to be an r-approximation if $S(I)/OPT(I) \leq r$. If the goal is to *minimize* the value of the solution, then $r \geq 1$ and it is desirable to find a solution with a *small* ratio r.

Definition 1 *Let* Π *be a* minimization *problem. An* approximation algorithm A *for* Π *is a* polynomial-time *algorithm that produces a valid solution* $A(I)$ *for any input* I. *The* performance ratio $\rho(A)$ *is defined as:*

$$\rho(A) = \max_I \frac{A(I)}{OPT(I)}.$$

In practice one would like a guarantee that $\rho \to 1$ rather than a constant ρ, but it is often the case that an approximation algorithm with a constant (worst-case) performance ratio is also a good algorithm for the average-case. We believe that a performance ratio bounded by constant is a precondition for good performance in practice.

There has been a recent characterization [28, 2, 20] of the class of NP problems that permit polynomial-time approximation with a constant ratio ρ. Informally, this class $\overline{\text{MAX SNP}}$ is the closure (under approximation-preserving reductions) of the syntactic class MAX SNP that consists of all problems expressible in terms of quantified first-order boolean formulas. We have used this machinery from the theory of approximation algorithms to guide our modeling and solution of the CRISP problem. Consequently, we are able to show that our CRISP formulation is not only a more accurate reflection of the desired goals in code optimization, but it also leads to optimization problems that are "easier to approximate" than the problems formulated using more conventional approaches.

4.3 Approximating Graph Coloring and Spill-Code Minimization

A standard technique for *global* register allocation is to view it as a problem involving the R-coloring of an interference graph for the live ranges [6, 7, 8, 9, 10, 11]. The vertices correspond to live ranges or virtual registers, and an edge between two vertices represents the information that the two live ranges will be active simultaneously, requiring the assignment of distinct physical registers. Thus, it is feasible to create a no-spill schedule if and only if the interference graph can be colored with R colors such that no two adjacent vertices receive the same color (i.e., physical register). Since graph coloring is NP-hard, heuristics are employed to determine a legal R-coloring, if one exists. When the graph is not R-colorable, spill code is introduced heuristically.

In this section, we provide strong *negative* results on the approximability of the spill-code minimization problem in this context, giving some explanation for the inherent difficulty of this problem as encountered in practice. While this sheds light on difficulty of register allocation and spill-code minimization problems, our interest is more towards establishing a contrast with CRISP which will be shown to bypass these negative results while providing a more accurate model of the optimization problem of reducing the run-time.

Proposition 1 *Given an interference graph* $G(V, E)$ *and a register bound* R*, there is an approximation algorithm with ratio* $\rho = \frac{R}{R-1}$ *for the problem of finding the maximum* R*-colorable subgraph* $H \subseteq G$.

For example, when $R = 32$, the approximation ratio is $32/31$ which is extremely close to 1.

However, there are two inaccuracies in using this problem as a model for spill-code minimization. First, the subgraph H is obtained from G by removing edges from it, and this corresponds to removing interferences between live-ranges. Spilling live-ranges corresponds more closely to removing *vertices* from the graph G, and the resulting problem becomes that of finding a maximum set $W \subseteq V$ such that the vertex-induced subgraph $H = G[W]$ is R-colorable. We can prove the following strong negative result for this new problem based on the results in [2].

Theorem 2 *Given an interference graph $G(V, E)$ and a register bound R, it is NP-hard to approximate the problem of finding the maximum R-colorable vertex-induced subgraph of G to within a ratio of n^ϵ.*

The second inaccuracy in modeling spill-code minimization is that the objective is not to *maximize* the number of live ranges *not spilled*, but rather to *minimize* the number of live ranges that are *spilled*. This is because that the running time of the generated code is increased by an amount proportional to the number of spills, and the number of non-spills is completely irrelevant. While the two problems are equivalent in the case of finding *optimal* solutions, there is no reason why their *approximability* should be related. In fact, for $R = 1$, the problem of minimizing the amount of spill is the minimum vertex cover problem which can be approximated within ratio $\rho = 2$ [24]. However, we can show that this happy situation does not carry over to the case of arbitrary register bounds.

Theorem 3 *Given an interference graph $G(V, E)$ and a register bound R, it is NP-hard to approximate within a ratio $\rho \leq n^\epsilon$ the problem of finding the minimum number of vertices to delete from G so as to leave an R-colorable subgraph.*

4.4 Approximability of CRISP

We now consider the approximability of the CRISP formulation. We saw earlier that FVS is a special case of CRISP, and so it becomes interesting to first analyze the approximability of this problem. The current best-known approximation algorithms for FVS instances of size n guarantee only a performance ratio $\rho = \Omega(\log n)$ (for example, see Leighton and Rao [22]). In fact, we conjecture that it is NP-hard to obtain an approximation ratio $\rho = O(\log n)$ for FVS.

Given the difficulty of approximating FVS, it may appear impossible to devise a "good" approximation algorithm (with constant ratio ρ) for even the RCRISP problem. However, it is important to keep in mind that the equivalence is only between the optimal solutions to these problems, and not their approximate solutions. In particular, a careful examination of the reduction from FVS to RCRISP and the proof of Theorem 1 shows that the FVS instance has a solution of value s if and only if the RCRISP instance has a solution of value $t = 3n + 2s$, where n is the size of the FVS instance. Since a feedback vertex set is a subset of the set of vertices, its size must satisfy $0 \leq s \leq n$. Consequently, the range of values for the RCRISP instance will be $3n \leq t \leq 5n$. It follows that any greedy solution for that RCRISP instance must be a ρ-approximation for $\rho \leq 5/3$. Of course, this only applies to the instances of RCRISP obtained from FVS instances via the reduction, but the basic idea can be generalized to all RCRISP instances, and in fact to all CRISP instances. We obtain the following theorem whose proof is omitted for the sake of brevity:

Theorem 4 *For an instance I of CRISP, let $u(I)$ be the average value of $|use(v_i)|$ over all instructions $v_i \in V$. There exists a polynomial-time algorithm for CRISP that has approximation ratio $1+c$ for instances I with $u(I) \leq c$.*

In practice, the value of $u(I)$ is significantly smaller than k, the average read/write degree of the basic block which should lie between 1 and 2.

5 A General Heuristic for CRISP

While we have established that CRISP has an approximation algorithm with worst-case ratio ρ that is a small constant, this does not suffice for practical purposes where we would like an average ratio of something like $\rho = 1 + \epsilon$ with $\epsilon \to 0$ on typical input instances. We present a generic heuristic in this section which we believe will give the desired performance in practice, as evidenced by our experimental results in Section 7.

The heuristic draws insight from the following *optimal* algorithm for spill-code generation when the instruction schedule is fixed. Suppose that for an instance of CRISP, we have a specific instruction schedule

σ that respects the dependences in DG. The problem is to spill the minimum number of value ranges so as to satisfy the register resource constraint. We can view value range *VR* $= (r, \sigma(v_s), \sigma(v_t))$ as defining an interval I_j that stretches between points $\sigma(v_s)$ and $\sigma(v_t)$ on the real line. The spill-code minimization problem can now be viewed as the problem of removing the smallest number of intervals so as to ensure that no more than R value ranges contain or straddle any point on the line. This is equivalent to the problem of deleting the smallest number of vertices in an interval graph $\mathcal{I}$ to render it R-colorable. Recall that an interval graph is obtained by placing a vertex for each interval, and connecting two intervals if they overlap non-trivially.

The following greedy algorithm runs in *linear* time and can be shown to give an optimal solution: scan the points from left-to-right; at each point where the register bound is violated, delete that interval containing the current point which goes furthest to the right. We omit the straightforward proof of the following proposition.

Proposition 2 *The greedy algorithm gives an optimal solution to the CRISP spill-code minimization.*

In the case of multiprocessor schedules, we can employ the same algorithm after associating with a point all instructions that are scheduled on any processor at that point in time. Variants of this greedy algorithm have been considered earlier in the compiler literature [12, 13, 17, 18].

We are now ready to describe our proposed heuristic. The heuristic depends on the following two rank functions that will be used in a greedy list scheduling algorithm.

Register Rank γ_R: For each instruction v, let $S_r(v)$ be the set of all successors of v in DG that read a virtual register r associated with v. For any $w \in S_r(v)$, let $p(v, w)$ be the length of the longest directed path in DG going from v to w, where the length is defined by the adding up the instruction execution times and the edge latencies over the path. Also, let $b(v, w)$ be the total execution time of the instructions that lie on all paths from v to w in DG. We define the rank as:

$$\gamma_R(v) = \min_{w \in S_r(v)} \max \left\{ p(v, w), \frac{b(v, w)}{M} \right\}.$$

This rank is 0 if $S_r(v) = 0$. If there are multiple virtual registers associated with v, then the rank is summed over the distinct choices of the virtual registers at v.

Scheduling Rank γ_S: We use the rank function of any good greedy list scheduling algorithm; for example, the rank function defined in [25, 26] could be a good choice.

Now, for any choice of the parameters $\alpha, \beta \in [0, 1]$ such that $\alpha + \beta = 1$, we define the following heuristic.

(α, β)**-Combined Heuristic:** This heuristic creates a combined rank function $\gamma = \alpha\gamma_S + \beta\gamma_R$, and orders the instructions into a list in increasing order of rank. Then, it runs the greedy list scheduling algorithm using this list to obtain a schedule without worrying about the register bound. The spill-code minimization algorithm described above is then used on this schedule to decide which value ranges to spill.

It is easy to verify that the computation of the rank functions and the greedy list schedule can be implemented in worst-case polynomial (quadratic) time; however, it is expected to be run in linear-time on typical inputs or in the average-case. This heuristic easily generalizes to CRISP, as well as its extensions described in Section 6.

6 Extensions to the Basic CRISP Model

In this section, we outline some important extensions to the basic formulation of the CRISP problem presented in Section 3; these extensions were excluded from the basic formulation so as to simplify the technical discussion in the earlier sections. While it is clear that the hardness results apply to the extended formulation, it is also not very hard to see that our heuristics and algorithms can also be suitably extended.

Multiple Register Classes. The basic formulation in Section 3 can be directly extended to model multiple register classes. For instance, consider the case when we need to model fixed-point registers, floating-point registers, and condition registers. We assume that each virtual register and each physical register belongs to one of these classes. Therefore, we can define $VR_c(\sigma)$ to be the set of value ranges (triples) over all virtual registers from register class c in schedule σ. The main extension required is to define the *bandwidth for register class* c at time τ, $BW_c(\sigma, VR_c(\sigma), \tau)$ to be the number of value-ranges from register class c that start at some time before τ and end at some time after τ.

Let R_c be the number of *physical registers* available for class c. Register spills are required for class c if $BW_c(\sigma, VR(\sigma), \tau) > R_c$ at any time τ. Let $SVR_c(\sigma) \subseteq VR_c(\sigma)$ be the set of *spilled* value ranges for register class c, and $AVR_c(\sigma) = VR_c(\sigma) - SVR_c(\sigma)$ be the set of *active* (non-spilled) value ranges for register class c. Let O_c be the overhead of a spill for register class c (in Section 3, we assumed $O_c = 2$ for a single class of registers). The objective function to be minimized is now $T(\sigma) + \sum_c O_c \times |SVR_c(\sigma)|$.

k-Stage Pipeline. In the basic formulation, we restricted all instruction execution times to = 1, and all edge latencies to = 0 or 1. This formulation can be used to model a single two-stage pipeline as in [25, 26, 4]. The generalization to a single k-stage pipeline is simple. We still restrict all execution times to = 1, but now restrict edge latencies to be in the range $0, \ldots, k-1$.

Spill Cost of Contiguous Value Ranges. We briefly mentioned in Section 3 that it is possible to avoid executing intermediate spill load/store instructions for a set of *contiguous* value ranges for the same virtual register. The savings come from not requiring an intermediate load/store pair at the point between two contiguous value ranges. This can be modeled by allowing contiguous value ranges to be merged in the SVR_c sets, e.g., if $(r, \tau_1, \tau_2) \in SVR_c$ and $(r, \tau_2, \tau_3) \in SVR_c$, then we can replace them by a single merged value range, (r, τ_1, τ_3). Let SVR'_c be the reduced spilled-value-range set obtained after this merging. The cost function now becomes $T(\sigma) + \sum_c O_c \times |SVR'_c(\sigma)|$, which can be smaller than before since $|SVR'_c(\sigma)| \leq |SVR_c(\sigma)|$.

Multiple Functional Units. For M load/store units, we extend the definition of the combined cost function to now be, $C(\sigma) = T(\sigma) + 2\lceil (|SVR(\sigma)|)/M \rceil$, so as to account for spill costs more accurately.

7 Experimental Results

In this section, we present some experimental results comparing phase-ordered solutions for register allocation and instruction scheduling with our approximate solution for CRISP using the (α, β)-combined heuristic. Subsection 7.1 summarizes the results observed. Subsection 7.2 outlines the instruction scheduling and register allocation algorithms that were implemented to obtain these experimental results. Subsection 7.3 describes the methodology used to automatically generate random problem instances for CRISP.

7.1 Summary of Results

Table 1 summarizes the average cost ratios that we observed for the combined heuristic, when compared to the phase-ordered solution in which instruction scheduling is followed by register allocation, for a single two-stage pipeline with 4, 8, and 16 available registers. Cost ratio values < 1 identify cases when the combined heuristic has a lower cost than the phase-ordered solution. These measurements were averaged over 100 randomly generated 100-instruction DAG's. The results show that the combined heuristic yields combined costs that are 16-21% better those obtained by this phase-ordered solution, despite the fact that the phase-ordered solution found schedules with makespans that are 13-14% smaller than those obtained by

Processor Description	# Available Registers	Average Ratio of Makespan	Average Ratio of # Spilled Value Ranges	Average Ratio of Combined cost
Two-stage pipeline	4	1.14	0.73	0.84
	8	1.14	0.57	0.79
	16	1.13	0.33	0.84

Table 1: Average cost ratio of Combined Heuristic compared to Scheduling→Allocation phase ordering, for 100 randomly generated 100-instruction DAG's

Processor Description	# Available Registers	Average Ratio of Makespan	Average Ratio of # Spilled Value Ranges	Average Ratio of Combined cost
Two-stage pipeline	4	0.78	0.81	0.79
	8	0.93	0.72	0.82
	16	1.04	0.65	0.96

Table 2: Average cost ratio of Combined Heuristic compared to Allocation→Scheduling phase ordering, for 100 randomly generated 100-instruction DAG's

the combined solution. These results underscore the importance of focusing the optimization on improving the combined cost function right from the start rather than on first improving the makespan.

Table 2 summarizes the improvements that we observed for the combined heuristic, when compared to the phase-ordered solution in which register allocation is followed by instruction scheduling, for a single two-stage pipeline with 4, 8, and 16 available registers. These measurements were averaged over 100 randomly generated 100-instruction DAG's. The results show that the combined heuristic performs 4-21% than this phase-ordered solution.

The above results were obtained using a simple $(0, 1)$-combined heuristic, where we ignore the $b(v, w)$ component of the rank function $\gamma_R(v)$ defined in Section 5.

In separate measurements, we've observed that for DAG's with little register pressure the phase-ordered solution in which instruction scheduling is followed by register allocation performs a little better (2-5%) than the $(0, 1)$-combined heuristic. However, the difference goes away if the (α, β) values are adjusted to give more priority to instruction scheduling in this case e.g. by setting $\alpha = 0.75$ and $\beta = 0.25$. The ability to adapt the behavior of the combined heuristic by adjusting the α and β values is an important practical strength of our algorithm.

7.2 Algorithms used in Experiments

For the Scheduling⟶Allocation phase-ordered solution, we implemented the Rank Algorithm [25, 26] for the instruction scheduling step, and the greedy algorithm (Section 5) for the register allocation step.

For the Allocation⟶Scheduling phase-ordered solution, we first tried using a standard coloring-based heuristic algorithm (e.g. [11]) and quickly discovered that it led to significantly larger number of spills than the greedy algorithm outlined in Section 5. To avoid severely penalizing this phase-ordered solution, note that all our measurements in Table 2 are based on using the greedy algorithm from Section 5 for the first register allocation step. For the instruction scheduling step, we use the Rank Algorithm as before.

For the combined solution, we use the (α, β)-Combined Heuristic presented in Section 5.

7.3 Generation of Random DAG's

The random DAG generator works with the following set of input parameters: number of nodes, number of node types, average inputs and outputs for each type, max latency, percentage of loads and stores. It forms a sequence of the nodes and randomly assign types to them. Now, the the connectivity and latency information is added as follows:

1. Repeat the following steps for each node i in the input graph.
2. We now select random predecessors for node i as follows: Let l = number of inputs for instruction in node i. Select l nodes randomly from the current def set. If node i is not a store (sink), addd it to the current def set.
3. Toss a coin biased by age to remove nodes that have been in the def set for a long time.
4. Increase the ages of all the nodes in the def. set.
5. Once the graph is defined, randomly assign latenices to all the nodes where the outgoing edges from node i get latencies $1 \ldots$ maxlatency$\{type(i)\}$; note that given a node i, its type defines its maximum latency.

8 Related Work

Recent research has now started to examine the interaction between register allocation and scheduling. The closest approach to a formulation which deals with the combined optimization problem is due to Probsting and Fischer [30]. They present an algorithm based on Sethi-Ullman numbering [31] which solves the combined allocation and scheduling problem for data-dependences that can be represented as *trees*. They restrict themselves to the case where the inter-instructional latencies (modeling pipeline delays) are required to be either 0 or 1 and latency-1 instructions (delayed loads) are only allowed to occur as leaves of the trees. Their approach does not address the question of data dependence structures more general than trees; note that even within basic blocks, the elimination of common subexpressions gives rise to directed acyclic graph (DAG) structures rather than trees.

In contrast, our proposed research aims at the general case where program fragments that represent basic blocks are modeled as DAGs, as also pieces that represent traces of entire procedures, well beyond single basic blocks. We will also consider more aggressive pipeline depths — essentially arbitrary integer depths specified as part of the input, rather than merely 0/1 pipeline latencies.

Another interesting attempt at reducing the interaction instruction scheduling and register allocation involves constructing a special "parallelizable interference graph." [29]. This graph has the property that if an optimal coloring is found for it, the result does not create any false dependences between instructions (due to register sharing). However, this approach is still dependent on allocation via graph coloring. If the performance of the coloring heuristic is poor and hence the number of available colors (registers) is less than the number required by the coloring heuristic, it once again becomes necessary to introduce spill-code. In this case, Pinter's non-interference property breaks down and heuristics are suggested to prioritize allocation over scheduling, or vice-versa, based on "estimates" of relative benefits. Other approaches have also been proposed for handling this interaction, e.g., by iterating repeatedly over the two steps *separately* [5, 15].

It is also interesting to note that *rematerialization* [7], a recent enhancement to register allocation, is actually a limited form of combined register allocation and instruction scheduling. The idea behind rematerialization is to shorten a live range by moving the instruction that computes the live range value closer to its first use. Rematerialization only considers limited code motion of this kind, and does not address any of the code motion that is required for instruction scheduling.

The drawback with previously known approaches is that they do not model combined register allocation and instruction scheduling within a single cost function. As a result, they do not have a sharp quantitative understanding and ability to predict the performance of algorithms and heuristics that are designed for the two optimizations, as parameters of the target processors (e.g. number of functional units, pipeline depths, number of registers) change over time.

9 Conclusions and Future Work

In this paper, we studied the problem of combining register allocation and instruction scheduling into a single phase. We established hardness and approximability results for this problem. We developed a new heuristic algorithm for the combined problem that is very promising, as evidenced by our early experimental results.

For future work, we will do a more comprehensive comparison of the combined and phase-ordered solutions, including a prototype implementation of these solutions within the back-end of the IBM XL compilers.

Acknowledgements

We thank Allen Leung of the Courant Institute for providing his scheduler and random graph generator for our experiments.

References

[1] F. Allen, B. Rosen, and K. Zadeck (editors). *Optimization in Compilers,* ACM Press and Addison-Wesley (to appear).

[2] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof Verification and Intractability of Approximation Problems. In *Proceedings of 33rd Annual IEEE Symposium on Foundations of Computer Science,* pp. 14–23, 1992.

[3] D. Bernstein, D. Goldin, M. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Pinter. Spill-code Minimization Techniques for Optimizing Compilers. In *Proceedings of SIGPLAN'89 Conference on Programming Language Design and Implementation,* pp. 258–263, 1989.

[4] D. Bernstein and M. Rodeh. Global Instruction Scheduling for Superscalar Machines. In *Proceedings of SIGPLAN'91 Conference on Programming Language Design and Implementation,* pp. 241–255, 1991.

[5] D. Bradlee, S. Eggers, and R. Henry. Integrated Register Allocation and Instruction Scheduling for RISCs. In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 122–131, 1991.

[6] P. Briggs, K.D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the SIGPLAN 89 Conference on Programming Language Design and Implementation*, pp. 275–274, 1989.

[7] P. Briggs, K.D. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems,* 16(3):428–455, 1994.

[8] G.J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN 89 Conference on Compiler Construction*, pp. 98–101, 1982.

[9] 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, 1981.

[10] F. Chow and J. Hennessy. The Priority based Graph Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems,* 12(4):501–536, 1990.

[11] F. Chow, K. Knobe, A. Meltzer, R. Morgan, and K. Zadeck. *Register Allocation*. In *Optimization in Compilers,* (eds: F. Allen, B. Rosen, and K. Zadeck), ACM Press and Addison-Wesley (to appear).

[12] C.W. Fraser and D.R. Hanson. Simple Register Spilling in a Retargetable Compiler. *SPE*, 22:85–99, 1992.

[13] R.A. Freiburghouse. Register Allocation Via Usage Counts. *CACM*, 17, 1974.

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

[15] J. Goodman and W. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. In *Proceedings of ACM Conference on Supercomputing,* pp. 442–452, 1988.

[16] R.L. Graham. Bounds for certain multiprocessing anomalies. *Bell Systems Technical Journal*, 45:1563–1581, 1966.

[17] L.J. Hendren, G.R. Gao, E.R. Altman, and C. Mukerji. Register Allocation using Cyclic Interval Graphs. ACAPS Technical Memo 33, McGill University, 1992.

[18] W-C. Hsu, C.N. Fischer, and J.R. Goodman. On the Minimization of Loads/Stores in Local Register Allocation. IEEE SE, 15:1252–1260, 1989.

[19] D. Karger, R. Motwani, and M. Sudan. Approximate Graph Coloring by Semidefinite Programming. In *Proceedings of 35th Annual IEEE Symposium on Foundations of Computer Science,* pp. 2–13, 1994.

[20] S. Khanna, R. Motwani, M. Sudan, and U.V. Vazirani. On Syntactic versus Computational Views of Approximability. In *Proceedings of 35th Annual IEEE Symposium on Foundations of Computer Science,* pp. 819–830, 1994.

[21] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys, Sequencing and Scheduling: Algorithms and Complexity, In: *Handbooks in Operations Research and Management Science, Vol. 4: Logistics of Production and Inventory*, 1990.

[22] F.T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pp. 256–269, 1988.

[23] C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. In *Proceedings of of the 25th ACM Symposium on Theory of Computing,* pp. 286–293, 1993.

[24] R. Motwani. *Lecture Notes on Approximation Algorithms.* Report No. STAN-CS-92-1435, Department of Computer Science, Stanford University, 1992.

[25] K. Palem and B. Simons. Scheduling Time-critical Instructions on RISC Machines. *ACM TOPLAS*, 5(3), 1993.

[26] K. Palem and B. Simons. Instruction Scheduling. In *Optimization in Compilers,* (eds: F. Allen, B. Rosen and K. Zadeck). ACM Press and Addison-Wesley (to appear).

[27] K. Palem and V. Sarkar Code Optimization in Modern Compilers. *Lecture Notes, Western Institute of Computer Science*, Stanford University, August 1995.

[28] C.H. Papadimitriou and M. Yannakakis. Optimization, Approximation, and Complexity Classes. *Journal of Computer and System Sciences*, 43:425–440, 1991.

[29] S. Pinter. Register Allocation with Instruction Scheduling. In *Proceedings of SIGPLAN'93 Conference on Programming Language Design and Implementation,* pp. 248–257, 1993.

[30] T. Probsting and C. Fischer. Linear-time Optimal Code Scheduling for Delayed-load Architectures. In *Proceedings of SIGPLAN'91 Conference on Programming Language Design and Implementation,* pp. 256–267, 1991.

[31] R. Sethi and J. Ullman. The Generation of Optimal Code for Arithmetic Expressions. *Journal of the ACM,* 17(4), 1970.