Register Allocation in Structured Programs

Sampath Kannan*
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

Todd Proebsting
Department of Computer Science
University of Arizona
Tucson, Arizona 85721

*Supported in part by NSF Grant CCR-91-08969

Running Head: Register Allocation

Contact Name: Sampath K. Kannan

Contact Address: Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104.

Abstract

In this paper we look at the register allocation problem. In the literature this problem is frequently reduced to the general graph coloring problem and the solutions to the problem are obtained from graph coloring heuristics. Hence, no algorithm with a good performance guarantee is known. Here we show that when attention is restricted to *structured programs* which we define to be programs whose control-flow graphs are series-parallel, there is an efficient algorithm that produces a solution which is within a factor of 2 of the optimal solution. We note that even with the above restriction the resulting coloring problem is NP-complete.

We also consider how to delete a minimum number of edges from arbitrary control-flow graphs to make them series-parallel and apply our algorithm. We show that this problem is Max SNP hard. However, we define the notion of an *approximate articulation point* and give efficient algorithms to find approximate articulation points. We present a heuristic for the edge deletion problem based on this notion which seems to work well when the given graph is close to series-parallel.

1 Introduction

Register Allocation is a very important task for a compiler. The register allocation problem is the problem of designing a strategy for a compiler to decide which variables to store in what registers at what times. The goal of the allocation strategy is to ensure that as often as possible during execution of a program, the necessary variables have their values stored in registers. Since loading and storing from registers is far faster than reading and writing from RAM a good register allocation strategy greatly speeds up the execution time of the compiled code.

There is a standard graph-theoretic formulation of the register allocation problem [6, 4]. In this formulation, we first consider the *control flow graph* or simply *flow graph* of a program. In this graph the nodes are the instructions or statements of the program and there is an arc from node i to node j if it is possible for the program to execute statement j immediately after executing statement i.

The concept of *liveness* of a variable plays a part in the register allocation problem [2]. Informally, a variable is live in the range of instructions between its definition and its possible uses. More formally, in the graph-theoretic formulation the *live range* of a variable is a connected subgraph of the control-flow graph. The goal is to store certain variables in registers for their entire live range to speed up access to these variables. Intuitively, if two variables are both live at the same instruction it is impossible for both of them to be stored in the same register. This motivates the definition of an *interference graph* where the vertices are the variables and there is an edge between vertex i and j if the live range of i intersects the live range of j. The clean theoretical question here is to find the chromatic number of the interference graph. In practice we have to deal with the bounded number of registers available and it is not sufficient to merely find the chromatic number of the interference graph. Instead, we are given a number k (the number of registers) and we have to properly k color as much of the interference graph as possible. Deciding which variables will not get colored (and hence will have to be stored in RAM) falls under the domain of the design of *spill heuristics*. In this paper, we will focus on the theoretical question and briefly mention how we can use this solution together with spill heuristics to actually implement a register allocation strategy.

It is easy to see that the register allocation graph that arises can be an arbitrary graph and hence the problem of determining the chromatic number of an interference graph is not only NP-complete but it is also impossible to obtain good approximations. (At least, it is clear that an approach based on graph coloring cannot obtain good approximations unless P = NP. Actually since there is also a reduction from graph coloring to register allocation other approaches to solving the register allocation problem also have the same problems.) In fact, in the literature some restrictions have been placed on possible control-flow graphs. In particular it is often assumed that the control flow graph is a *reducible flow graph* [2]. Even with this restriction it turns out that arbitrary register allocation graphs can arise and the problem of register allocation does not in general have a solution with a good theoretical guarantee.

In this paper we focus our attention on *structured programs*. Although it is intuitively clear what this means, for the purpose of this paper we will formally define these programs to be programs where the control flow graph is *series-parallel*. We will argue that this formalization generally coincides with intuition but will point out some areas of divergence in the section on open problems. We will show that for such programs, although the register allocation problem remains NP-hard, there is an efficient algorithm which produces a coloring of the interference graph using at most twice as many colors as optimal. Actually our coloring algorithm can handle control-flow graphs which are subgraphs of series-parallel graphs. This is easy to see because any register-allocation graph that arises from a subgraph of a series-parallel graph can be thought to arise from its series-parallel supergraph. This observation will be useful in the next problem we consider.

Most programs that are compiled are in fact structured. However a program may deviate slightly from being structured. Also, a compiler performs several optimizations on a program and in the process may alter the control-flow graph of a program. Thus we have to consider the situation where the control-flow graph is not series-parallel. Specifically, we are interested in taking an arbitrary graph as input (possibly with weights on each edge to indicate frequency of control flow along that edge) and deleting a set of edges of minimum total weight in order to make the resulting graph series-parallel. When an edge is deleted, all registers may have to be reassigned values whenever control passes through this edge and hence we want to minimize the total weight of the deleted edges. We show that the problem of finding the set of edges to be deleted is NP-hard even in the unweighted case and in fact is Max-SNP hard. However, we have heuristics that work well in the case that the control-flow graph deviates very little from being series-parallel. In fact our heuristics work to delete as few edges as possible to arrive at a control-flow graph that is a subgraph of a series-parallel graph. In the light of the remarks at the end of the previous paragraph this is sufficient to apply our coloring algorithm.

The rest of the paper is organized as follows. In section 2 we formalize the concepts described above and the problems solved in this paper. In section 3 we describe the algorithm that obtains a factor of two approximation to the coloring in the case where the control-flow graph is series-parallel. In section 4 we describe hardness results for the problems discussed in this paper. We also describe heuristics for finding a smallest set of edges to delete to obtain a series-parallel graph. In section 5 we describe the heuristics we use to decide which variables to spill and also describe our implementation experience. Finally, in section 6 we discuss open problems.

2 Definitions and Formulation of Problems

Definition 1 Series-parallel graphs *are graphs that have an ordered pair of special nodes called the* **source** *and the* **sink** *or collectively as* **terminals** *and they are recursively defined as follows:*

1. *A graph consisting of a source and a sink with a single edge between them is a series-*

parallel graph.

2. *If G_1 and G_2 are series-parallel graphs with terminals (s_1, t_1) and (s_2, t_2) respectively, then the graph G with terminals (s_1, t_2) obtained by identifying nodes t_1 and s_2 is a series-parallel graph. G is said to be a series connection of G_1 and G_2.*

3. *If G_1 and G_2 are series-parallel graphs with terminals (s_1, t_1) and (s_2, t_2) respectively, then the graph G with terminals (s_1, t_1) obtained by identifying s_1 with s_2 and t_1 with t_2 is a series-parallel graph. G is said to be a parallel connection of G_1 and G_2.*

4. *Any graph that cannot be obtained as above is not a series-parallel graph.*

Definition 2 *The* **control-flow graph** *(or simply* **flow graph***) of a program is a directed graph whose nodes are the instructions in the program with an edge from node i to node j if it is possible for instruction j to be executed right after instruction i in some execution of the program. Control-flow graphs have two special nodes which are the entry and exit points of the program.*

Definition 3 *We will say that a program is* **structured** *if its control-flow graph (ignoring the directions on the edges) is a series-parallel graph with terminals that are the entry and exit nodes of the program.*

Note that most of the common constructs in modern programming languages such as **for** and **while** loops and **if** ... **then** ... **else** statements either lead to control-flow graphs that are series-parallel or may easily be modified to produce such control-flow graphs.

To see that all loops give rise to series-parallel control-flow graphs, we define the control-flow graph associated with a loop in a slightly modified manner described below. In general for any loop we create two special nodes called entry and exit nodes so that control flows into the loop through the entry node and flows out of the loop through the exit node. A loop where the loop condition is tested at the top will have an edge from the entry node to the exit node, an edge from the exit node to the body of the loop and an edge from the body of the loop to the entry node. A loop whose condition is tested at the bottom will have an edge from the entry node to the body of the loop, an edge from the body of the loop to the exit node and an edge from the exit node to the entry node.

Among common constructs only **go to** statements present the possibility that the resulting flow graph is not series-parallel and these statements are generally considered not to be part of structured programming.

As stated in the introduction, we will assume that the *live range* of a variable is a connected subgraph of the control-flow graph of a program. In what follows we will asume that the control-flow graph is series-parallel and make definitions accordingly.

Definition 4 *A* **register allocation graph (RAG)** *is an intersection graph of connected subgraphs of a series-parallel graph.*

Thus RAGs are special kinds of interference graphs that arise when the control-flow graph is series-parallel. Note that for example RAGs include all chordal graphs since trees can be extended to series-parallel graphs by adjoining a single node (the sink) to a tree and connecting all the leaves of the tree to this sink. Since chordal graphs are intersection graphs of subtrees of a tree, it follows that they are RAGs. Also, RAGs include all circular-arc graphs since a simple cycle (with appropriate terminals) is a series-parallel graph. Thus the class of RAGs does not seem to be the same as any class of graphs studied in the literature.

The register-allocation problem for structured programs can now be stated as follows:

Problem: Register Allocation for Structured Programs (RASP).

Input: A series-parallel control-flow graph C and connected subgraphs of C representing the live range of the variables. (The implied RAG G is also available as part of the input although it could be efficiently inferred from the other information.)

Output: A proper coloring of G that uses the minimum number of colors possible.

In the next section we will describe an approximation algorithm to produce a coloring which is within twice the optimal in the next section.

3 The Approximation Algorithm

Note that the RASP problem is NP-hard since it includes the problem of coloring circular-arc graphs optimally which is known to be NP-hard.

The approximation algorithm relies on structural properties of RAGs.

Lemma 5 *If G is a RAG then it has two cliques whose removal separates it. Moreover each of the components resulting from this operation are themselves RAGs.*

Proof: The proof follows immediately from the fact that series-parallel graphs have complete decompositions in which at each stage the separators consist of one or two vertices. Each vertex in the (series-parallel) control-flow graph represents a clique in the RAG consisting of those variables that are live at that vertex. Thus the (at most) two vertices that need to be removed to separate the series-parallel graph correspond to the two cliques that need to be removed to separate the RAG. ∎

The above lemma can be used to obtain an approximation algorithm which comes within a factor of 3 of the optimal coloring. We briefly describe this algorithm.

1. Construct the decomposition tree of the series-parallel control flow graph.
2. Maintain a set of forbidden colors which is initially empty.
3. Color the variables passing through the terminals A and B of the series-parallel graph each with its own color and add these colors to the list of forbidden colors.

4. If the top-level decomposition is a parallel decomposition, recurse to coloring (the variables in) each of the components passing on the forbidden set of colors constructed in the previous step.

5. If the top-level decomposition is a series decomposition, at node C, color each uncolored variable passing through C with a new color from the complement of the forbidden set using the smallest indexed color at each point. Recurse with the following forbidden sets — for the graph with terminals A and C, the forbidden set is the set of colors used to color variables passing through A or C. For the graph with terminals B and C the forbidden set is the set of colors used to color variables passing through C or B.

Lemma 6 *The above algorithm achieves an approximation ratio of 3.*

Proof: Let k be the size of the largest clique in the RAG G. Clearly, the number of colors contained in the forbidden set at any point in the algorithm is at most $2k$ and at most k colors from outside of the forbidden set can be used by the algorithm at any point. Thus the total number of colors used is at most $3k$ and since k is a lower bound on the number of colors needed in an optimal coloring, the lemma follows. ∎

This algorithm can be implemented in linear time in the size of the input. It is well known that decomposition trees of series-parallel graphs can be found in linear time in their size. Since the input consists of the subgraphs corresponding to each variable, it is also easy to see that the set of variables passing through any given node of C can be identified in time proportional to the number of these variables and hence forbidden sets of colors can be maintained in linear time.

In order to improve the approximation factor from 3 to 2 we make two further observations. First of all, note that RAGs have a little more structure than indicated by the lemma above. In fact, if a RAG G has only a two clique separator (and no one clique separator) then every component that results from the removal of this separator must have a one clique separator. This follows from the observation that successive parallel compositions in building up a series-parallel graph can be combined into one parallel composition and hence in the decomposition tree of the series-parallel graph, any node that represents a parallel composition can be assumed to have all of its children representing series compositions. The above observations can be summarized in the following recursive characterization of RAGs.

Lemma 7 *A graph G is a RAG iff whenever G has only a two clique separator S, (and no one clique separator) each component of $G - S$ is a RAG with a one clique separator and whenever G has a one clique separator S, each component of $G - S$ is a RAG.*

Proof: The observations in the paragraph preceding the lemma establish one direction of the lemma. The other direction is fairly routine and involves noting that if G and its subgraphs satisfy the conditions of the lemma, one can construct a series-parallel graph and connected subgraphs of it such that G is the intersection graph of these subgraphs. ∎

Since a separator in a RAG consists of a set of vertices that can be partitioned into (at most) two cliques, we need to look more closely at such graphs. Such graphs have been considered in the literature and have been named *bicliques*[11]. The algorithm which achieves a factor of 3 approximation colors each vertex of a biclique with its own color. This may be far from optimal. In fact bicliques can be optimally colored efficiently.

Lemma 8 *Let $G = (\{V_1 \cup V_2\}, E)$ be a biclique on n vertices with the induced subgraphs on V_1 and V_2 being cliques. Then there is an $O(n^{2.5})$ algorithm to optimally color G.*

Proof:

Note that the complement of G is a bipartite graph. Note also that at most two vertices of G, one from V_1 and one from V_2 can use any one color. A pair of vertices using the same color must be adjacent in the complement. Thus, the optimal coloring is obtained by constructing the complement $\bar{G}$, finding a maximum matching in $\bar{G}$, and using the same color only on the end-points of each matched edge. This algorithm has time complexity no worse than $O(n^{2.5})$[9].

∎

We will also need a variant of the biclique coloring algorithm where one of the cliques has already been colored. Clearly, this algorithm does not significantly differ from the regular algorithm and we will freely use either variant in what follows. The above facts can be used to design a modified algorithm that achieves a factor of 2 approximation. This algorithm is described below.

1. Construct the decomposition tree of the series-parallel control flow graph.

2. Maintain a set of forbidden colors which is initially empty.

3. Color the variables passing through the terminals A and B of the series-parallel graph by using the algorithm that optimally colors a biclique.

4. If the top-level decomposition is a parallel decomposition, recurse to coloring (the variables in) each of the components passing on the forbidden set of colors constructed in the previous step.

5. If the top-level decomposition is a series decomposition, at node C, then optimally color the biclique formed by variables passing through A and C using previously assigned colors to the variables passing through A. For every variable v passing through C that was not previously colored, if the color assigned to v by the biclique coloring algorithm conflicts with a color assigned to a variable passing through B, then reassign v the smallest indexed color outside of the forbidden set. Update the smallest indexed color available. Recurse with the following forbidden sets — for the graph with terminals A and C the forbidden set is the set of colors used to color variables passing through A and C and similarly for the graph with terminals C and B.

Lemma 9 *The above algorithm achieves an approximation ratio of 2.*

Proof:

If we can show that the set of colors used at any point is the union of a set of colors to color a biclique and a set of colors to color a clique, then we will be done, since the number of colors in the optimal coloring is lower bounded by the cardinalities of each of these sets.

Assume inductively that this is true at a particular stage with terminals A and B. If the next stage involves a parallel decomposition, then obviously no new colors are used. Suppose that the next stage is a series decomposition at C. Then the algorithm works by finding the optimal coloring of the biclique formed by variables through A and C and then changing some of the colors assigned to variables passing through C if they conflict with colors assigned to variables passing through B. Imagine instead that the reassignment of colors was done to the conflicting variables passing through B. Then, the variables passing through A and C are optimally colored (using a biclique coloring) and the variables passing through B are colored using at most a number of new colors equal to the size of the clique at B. Thus the total number of colors used still satisfies the inductive hypothesis proving the claimed approximation bound.

∎

4 Dealing with General Control-Flow Graphs

Given an arbitrary control-flow graph we would like to transform it into a (subgraph of a) series-parallel graph and then apply our approximate coloring algorithm to do register allocation. Even if control-flow graphs are not series-parallel, we expect that they have only a few offending edges whose removal transforms them into series-parallel graphs. In the context of register allocation, removing an edge e from a control-flow graph means that whenever program execution goes along e, all register values may have to be reassigned. As long as the execution frequency along deleted edges is fairly small, this will not have a significant impact on the efficiency of the compiled code. For simplicity, we will assume here that all edges have equal execution frequencies. Thus our goal is to take an arbitrary control-flow graph and find the series-parallel subgraph with the maximum number of edges. We will refer to this problem as the KSP problem.

The KSP Problem: Given a graph G with two special nodes source and sink and an integer K, pick K or more edges such that the resulting graph G' is (a subgraph of a) series-parallel with terminals at the given source and sink.

Theorem 10 KSP *is NP-Complete.*

Clearly, KSP is in NP. We shall now reduce 3-SAT to KSP. We are given a 3-SAT Boolean formula F, with m clauses $C_1, \ldots, C_m$ and n variables $\alpha_1, \ldots, \alpha_n$. We shall construct a graph $G(F)$ and an integer K which is a YES-instance of KSP if and only if F is satisfiable. $G(F)$ has $8n + m + 2$ vertices: eight vertices for each variable α_i, one vertex for each clause C_i, and source and sink vertices.

The graph $G(F)$ can be broken up into the 'truth-setting' part and the 'clause satisfaction' part. In the truth-setting part for each variable α_i in F, there will be two paths from source to sink each consisting of 4 internal vertices, accounting for the 8 vertices per variable. In addition there will be two 'cross-edges' from the second internal vertex from each of the two paths to the third internal vertex in the other path. It will turn out that there will always be a spanning series-parallel subgraph and that in any such subgraph of $G(F)$ we can choose the two non-crossing paths corresponding to a variable by deleting the cross-edges and this corresponds to choosing the variable to be true. We can instead choose the two crossing paths by deleting the edges between the second and third internal nodes in each path and this corresponds to setting the variable to false.

In the clause satisfaction part there is a vertex for each clause C. if variable x occurs uncomplemented in C then c is connected to the first and fourth internal vertices of one of the paths corresponding to x. (Thus, if we pick the two non-crossing paths for x, we can also pick these two edges to c corresponding to setting x to true and thereby satisfying C.) If x occurs complemented in C then c is connected to the first internal vertex in one path and the fourth internal vertex in the other path corresponding to x. (Thus, if we pick the two crossing paths for x, we can also pick these two edges to c corresponding to setting x to false and thereby satisfying C.)

Thus, we can always choose 10 edges per variable. For each clause we can choose at most 2 edges, and we can choose 2 edges only when one of the variables in the clause has been 'made true' by the paths chosen for the variables. If we specify K to be $10n + 2m$, then $G(F)$ has a series-parallel subgraph with K edges if and only if F is satisfiable.

We give a formal description of the construction of $G(F)$ below.

$$\begin{aligned} V &= \{u_{ij} : i = 1, \ldots, n; j = 1, \ldots, 4\} \\ &\cup \{v_{ij} : i = 1, \ldots, n; j = 1, \ldots, 4\} \\ &\cup \{c_k : k = 1, \ldots, m\} \cup \{s, t\} \end{aligned}$$

$G(F)$ has $12n + 6m$ edges,

$$\begin{aligned} E = \; & \{[s, u_{i1}] : i = 1, \ldots, n\} \\ & \cup \{[s, v_{i1}] : i = 1, \ldots, n\} \\ & \cup \{[u_{i4}, t] : i = 1, \ldots, n\} \\ & \cup \{[v_{i4}, t] : i = 1, \ldots, n\} \\ & \cup \{[c_k, u_{i1}], [c_k, u_{i4}] : k = 1, \ldots, m; \alpha_i \in C_k\} \\ & \cup \{[c_k, u_{i1}], [c_k, v_{i4}] : k = 1, \ldots, m; \bar{\alpha}_i \in C_k\} \\ & \cup \{[u_{ij}, u_{ij+1}] : i = 1, \ldots, n; j = 1, 2, 3\} \\ & \cup \{[v_{ij}, v_{ij+1}] : i = 1, \ldots, n; j = 1, 2, 3\} \\ & \cup \{[u_{i2}, v_{i3}], [v_{i2}, u_{i3}] : i = 1, \ldots, n\}. \end{aligned}$$

The above reduction shows the NP-completeness of KSP. In fact, by using more than one vertex per clause and using the same types of edges, we can make this reduction an L-reduction[12] thereby showing that KSP is Max SNP-hard.

Lemma 11 *KSP is Max SNP-hard.*

4.1 A Heuristic to Solve KSP:

Since KSP is NP-hard we will explore a reasonable heuristic to solve it. Our heuristic is based on the idea of *approximate articulation points.*

Definition 12 *A vertex in a graph is a k-articulation point if it becomes an articulation point after the removal of (at most) k edges.*

If we can find the smallest k for which there is a k-articulation point v, in the control-flow graph, we can use this vertex to produce a *series* decomposition and recurse to smaller problems.

We will also find the smallest number of edges that need to be removed to produce a *parallel* decomposition by computing a min-cut in $G - \{s, t\}$. Sometimes we will have subgraphs H where we know only one terminal s_h. In such cases we want to choose the other terminal in a way that minimizes the number of edges deleted. We cannot apply the min-cut procedure in such situations since we do not know t_h. Instead, we have to find the smallest l for which there is an l-articulation point in $H - s_h$. If this l is small enough, the articulation point found will be the other terminal of H.

Once we have computed these two quantities we can greedily choose a smallest set of edges to produce a series or parallel decomposition. We describe the details of this choice below. Let k be the smallest integer such that there is a vertex v which is a k articulation point. Let l be the size of the min-cut in $G - \{s, t\}$.

If $l \leq k$ then perform a parallel decomposition by removing the l edges in the min-cut. Otherwise we will do a series decomposition. There are several cases.

Case 1: The vertex discovered, v, is neither s nor t and removing the k edges and v separates s from t.

In this case we remove the k edges and do a series decomposition at v and recurse on the components.

Case 2: The vertex discovered is s or t.

This case does not arise if $k < l$.

Case 3: The vertex discovered is v which is neither s nor t, but the removal of v does not separate s from t.

For every component other than the one containing s and t, we recursively find a series-parallel decomposition with a neighbor of v as one of the terminals and the

other terminal unspecified. We also recursively process the component containing s and t. (We are decomposing the original graph into a subgraph of a series-parallel graph after removing a few edgesin this case.)

In recursive applications of this algorithm it is possible that only one of the terminals, s, is known. In this case whenever an articulation point v other than s is discovered, we will let t be chosen later such that v separates t from s.

The above greedy heuristic seems especially good for graphs that require only a few edge deletions to make them series-parallel. The main task is to find approximate articulation points efficiently.

Finding Approximate Articulation Points

It is possible to find approximate articulation points using a brute-force technique based on network flows. To do this, for each vertex v in the graph G, find the minimum cut in $G - v$. (Here we are dealing with all possible cuts without a specific source and sink.)

Lemma 13 *The vertex v for which $G-v$ has the smallest min-cut is the desired approximate articulation point.*

Proof:

Suppose $G - v$ has a min-cut of size k. Then v is a k-articulation point. Conversely if v is a k-articulation point then the removal of k edges from $G - v$ disconnects $G - v$ and hence thhe min-cut in $G - v$ is of size no greater than k.

∎

The problem with the above algorithm is that as stated it requires $O(n)$ min-cut computations. Unless it can be improved to solve the problem in a single min-cut computation this approach will probably be prohibitively expensive in practice.

For finding k-articulation points for small k (such as $k = 1$ or 2) techniques based on *ear decompositions* of graphs (See for example [10]) yield linear time algorithms. If there are no 1 or 2-articulation points, we use various heuristics based on depth-first search. The goal of our heuristics is to make sure that all edges that need to be removed to make a vertex an articulation point occur as back edges in the depth-first search. In order to try to achieve this we reorder the neighbor list of each vertex attempting to move "local" neighbors to the front of the list. A more formal description of this heuristic and its properties is currently open.

5 Spill Heuristics and Implementation Experience

Currently, we are experimenting with these heuristics in order to compare their efficiency and effectiveness against the general graph coloring techniques used in other register allocation systems [6, 4]. Our system proceeds by compiling a FORTRAN program into an arbitrary control-flow graph that is processed into a series-parallel decomposition. Heuristics are

employed to create series-parallel graphs from arbitrary flow graphs as needed. Once the decomposition is created, the graph is colored using the heuristic modeled after the first approximation algorithm described above. (We have yet to experiment with the theoretically better heuristic that achieves an approximation ratio of 2.) The implementation differs by using actual interferences — rather than *forbidden sets* — to more greedily assign colors to register candidates. When the heuristic exhausts available registers, uncolored candidates are simply kept in memory. To optimize performance, variables are colored in priority order based on static estimates of execution frequencies.

Our compiler optimizes the FORTRAN code before submitting it to the register allocator. We have found that the majority of our programs contain non-series-parallel constructs, but that they are relatively easy to transform into series-parallel form. The system emits C code instrumented to gather statistics on how well the register allocator worked by counting loads and stores. Despite our simple spilling heuristics, preliminary results indicate that our system performs roughly as well as previously known register allocations techniques on a variety of numeric benchmark programs. Many more tests on a wider variety of programs are needed to verify these results. We plan to use more sophisticated spilling heuristics in the future. Because we have a decomposition graph, it should be straightforward to spill candidates not across the entire program as we do now, but rather only over structured portions of the program (eg., spilling a candidate that is not used in a loop over the entire loop so that busy candidates can utilize that register). Spilling in a structured way will improve program performance, and should be simpler than previously proposed techniques for spilling live ranges [3, 5, 7, 8].

6 Conclusions and Open Problems

Series-parallel graphs seem like a good model for control-flow graphs in structured programs. However, one common program segment causes the resulting graph to be not series-parallel. This is the **if** ... **then** ... **else** statement with short circuit evaluation. Specifically the following code fragment illustrates the problem.

if A **or** B **then** C; D;

In order to handle this case we could do one of two things — transform the control-flow graph with some simple transformation to make it series-parallel or consider a broader class of graphs than series-parallel graphs to model structured programs.

We have no performance guarantees on the algorithms to extract series-parallel subgraphs of a given graph. It may be possible to obtain guarantees especially with restrictions on the types of input graphs such as a bound on the degree of every node or assuming that there is a small constant k such that there are k edges that can be removed to make the graph series-parallel. (An $O(n^k)$ algorithm in this case clearly works but is too expensive to implement.)

When an edge from the control-flow graph needs to be removed in order to make the graph series-parallel it may be possible to do something cleverer than simply reassigning values to all the registers. A more careful look at the coloring produced by our algorithm may tell us exactly which registers need to be reassigned.

Compilers perform instruction scheduling as well as register allocation and these tasks are sometimes in conflict. In instruction scheduling the instructions in a program are reordered to exploit the parallel capabilities of the processor. If instruction scheduling is done before register allocation this may extend the live range of variables and cause more variables to be spilled. On the other hand if register allocation is performed before instruction scheduling this may constrain what instructions can be rescheduled. An integrated approach based on heuristics performing both tasks simultaneously has been proposed by Pinter[13]. Developing an algorithm for this task that provides performance guarantees is an important open problem.

7 Acknowledgements

We would like to acknowledge several helpful discussions with Mudita Jain. Thanks are also due to Brady Montz for his implementation efforts. Finally we would like to thank the referees for SODA for many helpful suggestions.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Algorithms*. Addison-Wesley, Reading, Massachussetts, 1974.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[3] P. Briggs, K. Cooper, and L. Torczon. Aggressive live range splitting. Technical report, Rice University, 1991.

[4] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 275–284, 1989.

[5] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 192–203, June 1991.

[6] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 98–101, June 1982.

[7] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[8] Rajiv Gupta, Mary Lou Soffa, and Tim Steele. Register allocation via clique separators. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, 264–274, 1989.

[9] J. Hopcroft and R.M. Karp. $O(n^{5/2})$ Algorithm for Maximum Matching in Bipartite Graphs. *SIAM J. Comput.*, vol. 2, 225–231, 1973.

[10] P. Kelsen and V. Ramachandran. On Finding Minimal Two-Connected Subgraphs. *Journal of Algorithms*, vol. 18, 1995.

[11] J. Naor, M. Naor, and A. Schäffer. Fast Parallel Algorithms for Chordal Graphs. *19th ACM STOC*, New York, 355–364, May 25–27, 1987.

[12] C.H. Papadimitriou and M. Yannakakis. Optimization, Approximation, and Complexity Classes. *20th ACM STOC*, Chicago 229–234, May 2–4, 1988.

[13] Shlomit Pinter. *Register Allocation with Instruction Scheduling: a New Approach.* Proc. ACM-SIGPLAN-PLDI*, Albuquerque, Albuquerque, 248–257, June 1993.*