Register Allocation Via Coloring of Chordal Graphs

Fernando Magno Quintão Pereira and Jens Palsberg

UCLA Computer Science Department,
University of California, Los Angeles

Abstract. We present a simple algorithm for register allocation which is competitive with the iterated register coalescing algorithm of George and Appel. We base our algorithm on the observation that 95% of the methods in the Java 1.5 library have chordal interference graphs when compiled with the JoeQ compiler. A greedy algorithm can optimally color a chordal graph in time linear in the number of edges, and we can easily add powerful heuristics for spilling and coalescing. Our experiments show that the new algorithm produces better results than iterated register coalescing for settings with few registers and comparable results for settings with many registers.

1 Introduction

Register allocation is one of the oldest and most studied research topics of computer science. The goal of register allocation is to allocate a finite number of machine registers to an unbounded number of temporary variables such that temporary variables with interfering live ranges are assigned different registers. Most approaches to register allocation have been based on graph coloring. The graph coloring problem can be stated as follows: given a graph G and a positive integer K, assign a color to each vertex of G, using at most K colors, such that no two adjacent vertices receive the same color. We can map a program to a graph in which each node represents a temporary variable and edges connect temporaries whose live ranges interfere. We can then use a coloring algorithm to perform register allocation by representing colors with machine registers.

In 1982 Chaitin [8] reduced graph coloring, a well-known NP-complete problem [18], to register allocation, thereby proving that also register allocation is NP-complete. The core of Chaitin's proof shows that the interference relations between temporary variables can form any possible graph. Some algorithms for register allocation use integer linear programming and may run in worst-case exponential time, such as the algorithm of Appel and George [2]. Other algorithms use polynomial-time heuristics, such as the algorithm of Briggs, Cooper, and Torczon [5], the Iterated Register Coalescing algorithm of George and Appel [12], and the Linear Scan algorithm of Poletto and Sarkar [16]. Among the polynomial-time algorithms, the best in terms of resulting code quality appears to be iterated register coalescing. The high quality comes at the price of handling spilling and coalescing of temporary variables in a complex way. Figure 1

K. Yi (Ed.): APLAS 2005, LNCS 3780, pp. 315–329, 2005.

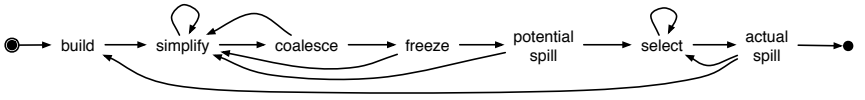


Fig. 1. The iterated register coalescing algorithm

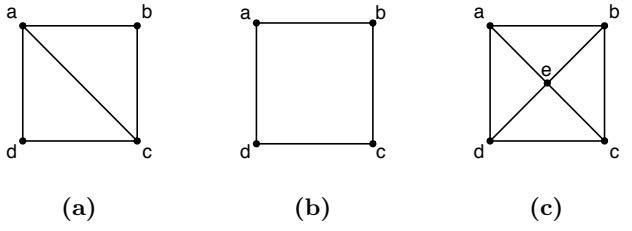


Fig. 2. (a) A chordal graph. (b-c) Two non-chordal graphs.

illustrates the complexity of iterated register coalescing by depicting the main phases and complicated pattern of iterations of the algorithm. In this paper we show how to design algorithms for register allocation that are simple, efficient, and competitive with iterated register coalescing.

We have observed that the interference graphs of real-life programs tend to be *chordal* graphs. For example, 95% of the methods in the Java 1.5 library have chordal interference graphs when compiled with the JoeQ compiler. A graph is chordal if every cycle with four or more edges has a *chord*, that is, an edge which is not part of the cycle but which connects two vertices on the cycle. (Chordal graphs are also known as 'triangulated', 'rigid-circuit', 'monotone transitive', and 'perfect elimination' graphs.) The graph in Figure 2(a) is chordal because the edge *ac* is a chord in the cycle *abcda*. The graph in Figure 2(b) is non-chordal because the cycle *abcda* is chordless. Finally, the graph in Figure 2(c) is non-chordal because the cycle *abcda* is chordless, just like in Figure 2(b).

Chordal graphs have several useful properties. Problems such as *minimum coloring*, *maximum clique*, *maximum independent set* and *minimum covering by cliques*, which are NP-complete in general, can be solved in polynomial time for chordal graphs [11]. In particular, optimal coloring of a chordal graph $G = (V, E)$ can be done in $O(|E| + |V|)$ time.

In this paper we present an algorithm for register allocation, which is based on a coloring algorithm for chordal graphs, and which contains powerful heuristics for spilling and coalescing. Our algorithm is simple, efficient, and modular, and it performs as well, or better, than iterated register coalescing on both chordal graphs and non-chordal graphs.

The remainder of the paper is organized as follows: Section 1 discusses related work, Section 3 summarizes some known properties and algorithms for chordal graphs, Section 4 describes our new algorithm, Section 5 presents our experimental results, and Section 6 concludes the paper.

2 Related Work

We will discuss two recent efforts to design algorithms for register allocation that take advantage of properties of the underlying interference graphs. Those efforts center around the notions of perfect and 1-perfect graphs. In a 1-perfect graph, the chromatic number, that is, the minimum number of colors necessary to color the graph, equals the size of the largest clique. A perfect graph is a 1-perfect graph with the additional property that every induced subgraph is 1-perfect. Every chordal graph is perfect, and every perfect graph is 1-perfect.

Andersson [1] observed that all the 27,921 interference graphs made publicly available by George and Appel [3] are 1-perfect, and we have further observed that 95.6% of those graphs are chordal when the interferences between precolored registers and temporaries are not considered. Andersson also showed that an optimal, worst-case exponential time algorithm for coloring 1-perfect graphs is faster than iterated register coalescing when run on those graphs.

Recently, Brisk et al. [6] proved that *strict* programs in SSA-form have *perfect* interference graphs; independently, Hack [14] proved the stronger result that strict programs in SSA-form have *chordal* interference graphs. A strict program [7] is one in which every path from the initial block until the use of a variable v passes through a definition of v. Although perfect and chordal graphs can be colored in polynomial time, the practical consequences of Brisk and Hack's proofs must be further studied. SSA form uses a notational abstraction called *phi-function*, which is not implemented directly but rather replaced by copy instructions during an SSA-elimination phase of the compiler. Register allocation after SSA elimination is NP-complete [15].

For example, Figure 3(a) shows a program with a non-chordal interference graph, Figure 3(b) shows the program in SSA form, and Figure 3(c) shows the

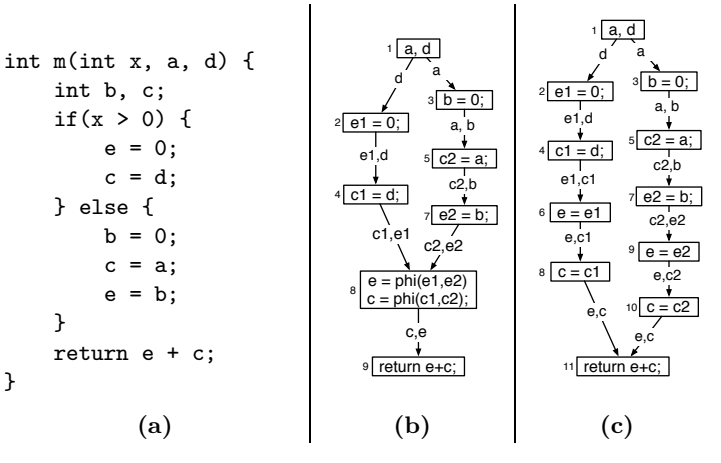


Fig. 3. (a) A program with a non-chordal interference graph, (b) the program in SSA form, (c) the program after SSA elimination

program after SSA elimination. The example program in Figure 3(a) has a cycle of five nodes without chords: a–d–e–c–b–a. In the example in Figure 3(b), $e = phi(e_1, e_2)$ will return e_2 if control reaches block 8 through block 7, and will return e_1 if control reaches block 8 through block 4. The SSA semantics states that all phi-functions at the beginning of a block must be evaluated simultaneously as the first operation upon entering that block; thus, live ranges that reach block 8 do not interfere with live ranges that leave block 8. Hack [14] used this observation to show that phi-functions break chordless cycles so strict programs in SSA-form have chordal interference graphs. The example program after SSA elimination, in Figure 3(c), has an interference graph which is non-chordal, non-perfect, and even non-1-perfect: the largest clique has two nodes but three colors are needed to color the graph. Note that the interference graph has a cycle of seven nodes without chords: a–d–e1–c1–e–c2–b–a.

For 1-perfect graphs, recognition and coloring are NP-complete. Perfect graphs can be recognized and colored in polynomial time, but the algorithms are highly complex. The recognition of perfect graphs is in $O(|V|^9)$ time [9]; the complexity of the published coloring algorithm [13] has not been estimated accurately yet. In contrast, chordal graphs can be recognized and colored in $O(|E|+|V|)$ time, and the algorithms are remarkably simple, as we discuss next.

3 Chordal Graphs

We now summarize some known properties and algorithms for chordal graphs. For a graph G, we will use $\Delta(G)$ to denote the maximum outdegree of any vertex in G, and we will use $N(v)$ to denote the set of neighbors of v, that is, the set of vertices adjacent to v in G. A *clique* in an undirected graph $G = (V, E)$ is a subgraph in which every two vertices are adjacent. A vertex $v \in V$ is called *simplicial* if its neighborhood in G is a clique. A *Simplicial Elimination Ordering* of G is a bijection $\sigma : V(G) \rightarrow \{1 \ldots |V|\}$, such that every vertex v_i is a simplicial vertex in the subgraph induced by $\{v_1, \ldots, v_i\}$. For example, the vertices b, d of the graph shown in Figure 2(a) are simplicial. However, the vertices a and c are not, because b and d are not connected. In this graph, $\langle b, a, c, d\rangle$ is a simplicial elimination ordering. There is no simplicial elimination ordering ending in the nodes a or c. The graphs depicted in Figures 2(b) and 2(c) have no simplicial elimination orderings.

Theorem 1. (Dirac [10]) *An undirected graph without self-loops is chordal if and only if it has a simplicial elimination ordering.*

The algorithm *greedy coloring*, outlined in Figure 4, is a $O(E)$ heuristic for graph coloring. Given a graph G and a sequence of vertices ν, *greedy coloring* assigns to each vertex of ν the next available color. Each color is a number c where $0 \leq c \leq \Delta(G) + 1$. If we give *greedy coloring* a simplicial elimination ordering of the vertices, then the greedy algorithm yields an optimal coloring [11]. In other words, *greedy coloring* is optimal for chordal graphs.

```
procedure greedy coloring
1    input: G = (V, E), a sequence of vertices ν
2    output: a mapping m, m(v) = c, 0 ≤ c ≤ Δ(G) + 1, v ∈ V
3    For all v ∈ ν do m(v) ←⊥
4    For i ← 1 to |ν| do
5        let c be the lowest color not used in N(ν(i)) in
6            m(ν(i)) ← c
```

Fig. 4. The greedy coloring algorithm

```
procedure MCS
1    input: G = (V, E)
2    output: a simplicial elimination ordering σ = v_1, ..., v_n
3    For all v ∈ V do λ(v) ← 0
4    For i ← 1 to |V| do
5        let v ∈ V be a vertex such that ∀u ∈ V, λ(v) ≥ λ(u) in
6            σ(i) ← v
7            For all u ∈ V ∩ N(v) do λ(u) ← λ(u) + 1
8            V ← V − {v}
```

Fig. 5. The maximum cardinality search algorithm

The algorithm known as *Maximum Cardinality Search* (MCS)[17] recognizes and determines a simplicial elimination ordering σ of a chordal graph in $O(|E|+|V|)$ time. MCS associates with each vertex v of G a weight $\lambda(v)$, which initially is 0. At each stage MCS adds to σ the vertex v of greatest weight not yet visited. Subsequently MCS increases by one the weight of the neighbors of v, and starts a new phase. Figure 5 shows a version of MCS due to Berry et al. [4].

The procedure MCS can be implemented to run in $O(|V|+|E|)$ time. To see that, notice that the first loop executes $|V|$ iterations. In the second loop, for each vertex of G, all its neighbors are visited. After a vertex is evaluated, it is removed from the remaining graph. Therefore, the weight λ is increased exactly $|E|$ times. By keeping vertices in an array of buckets indexed by λ, the vertex of highest weight can be found in $O(1)$ time.

4 Our Algorithm

Our algorithm has several independent phases, as illustrated in Figure 6, namely coloring, spilling, and coalescing, plus an optional phase called *pre-spilling*. Coalescing must be the last stage in order to preserve the optimality of the coloring algorithm, because, after merging nodes, the resulting interference graph can be non-chordal. Our algorithm uses the *MCS* procedure (Figure 5) to produce an ordering of the nodes, for use by the pre-spilling and coloring phases. Our approach yields optimal colorings for chordal graphs, and, as we show in Section 5, it produces competitive results even for non-chordal graphs. We have

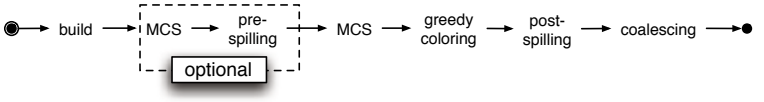
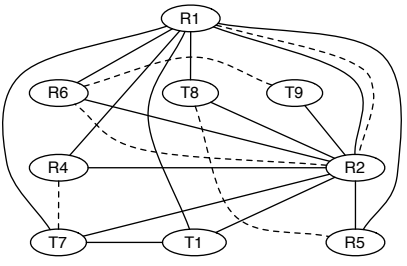


Fig. 6. The main phases of our algorithm

```
int gcd (int R1, int R2)
1.  IFCMP_I_EQ    ..  R2   0   (12);
2.  ZERO_CHECK_I  ..  T1  R2;
3.  DIV_I         T7  R1   R2;
4.  CHECK_EX      T1
5.  MOVE_I        R4  T7;
6.  MUL_I         T8  R2   R4;
7.  MOVE_I        R5  T8;
8.  SUB_I         T9  R1   R5;
9.  MOVE_I        R6  T9;
10. MOVE_I        R1  R2;
11. MOVE_I        R2  R6;
12. GOTO          ..  ..   ..  (1);
13. RETURN_I      ..  R1;
```

(a)



(b)

Fig. 7. (a) Euclid's algorithm. (b) Interference graph generated for gcd().

implemented heuristics, rather than optimal algorithms, for spilling and coalescing. Our experimental results show that our heuristics perform better than those used in the iterated register coalescing algorithm.

In order to illustrate the basic principles underlying our algorithm, we will as a running example show how our algorithm allocates registers for the program in Figure 7 (a). This program calculates the greatest common divisor between two integer numbers using Euclid's algorithm. In the intermediate representation adopted, instructions have the form op, t, p_1, p_2. Such an instruction defines the variable t, and adds the temporaries p_1 and p_2 to the chain of used values. The interference graph yielded by the example program is shown in Figure 7 (b). Solid lines connecting two temporaries indicate that they are simultaneously alive at some point in the program, and must be allocated to different registers. Dashed lines connect move related registers.

Greedy Coloring. In order to assign machine registers to variables, the *greedy coloring* procedure of Figure 4 is fed with an ordering of the vertices of the interference graph, as produced by the *MCS* procedure. From the graph shown in Figure 7 (b), MCS produces the ordering: ⟨ `T7, R1, R2, T1, R5, R4, T8, R6, T9` ⟩, and *greedy coloring* then produces the mapping between temporaries and colors that is outlined in Figure 8 (a). If the interference graph is chordal, then the combination of *MCS* and *Greedy Coloring* produces a minimal coloring. The coloring phase uses an unbounded number of colors so that the interference graph can always be colored. The excess of colors will be removed in the post-spilling stage.

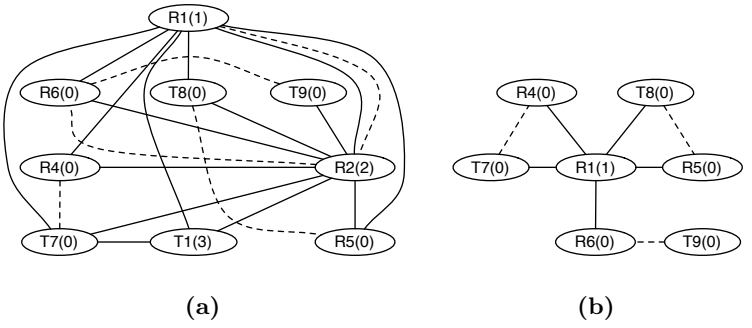


Fig. 8. (a) Colored interference graph. (b) Interference graph after spilling the highest colors.

Post-Spilling. Given an instance of a register allocation problem, it may be possible that the number of available registers is not sufficient to accommodate all the temporary variables. In this case, temporaries must be removed until the remaining variables can be assigned to registers. The process of removing temporaries is called spilling. A natural question concerning spilling when the interference graph is chordal is if there is a polynomial algorithm to determine the minimum number of spills. The problem of determining the maximum K-colorable subgraph of a chordal graph is NP-complete [20], but has polynomial solution when the number of colors (K) is fixed. We do not adopt the polynomial algorithm because its complexity seems prohibitive, namely $O(|V|^K)$ time.

Iterated register coalescing performs spilling as an iterative process. After an unsuccessful attempt to color the interference graph, some vertices are removed, and a new coloring phase is executed. We propose to spill nodes in a single iteration, by removing in each step all nodes of a chosen color from the colored interference graph. The idea is that given a K-colored graph, if all the vertices sharing a certain color are removed, the resulting subgraph can be colored with $K-1$ colors. We propose two different heuristics for choosing the next color to be removed: (i) remove the least-used color, and (ii) remove the highest color assigned by the greedy algorithm.

The spilling of the highest color has a simpler and more efficient implementation. The heuristic is based on the observation that the greedy coloring tends to use the lower colors first. For a chordal graph, the number of times the highest color is used is bounded by the number of maximal cliques in the interference graph. A maximal clique is a clique that cannot be augmented. In other words, given a graph $G = (V, E)$, a clique Q is maximal if there is no vertex $v, v \in V - Q$, such that v is adjacent to all the vertices of Q. For our running example, Figure 8 (b) shows the colored interference graph after the highest colors have been removed, assuming that only two registers are available in the target machine. Coincidentally, the highest colors are also the least-used ones.

```
procedure coalescing
1     input: list l of copy instructions, G = (V, E), K
2     output: G', the coalesced graph G
3     let G' = G in
4     for all x := y ∈ l do
5         let S_x be the set of colors in N(x)
6         let S_y be the set of colors in N(y)
7         if there exists c, c < K, c ∉ S_x ∪ S_y then
8             let xy, xy ∉ V be a new node
9             add xy to G' with color c
10            make xy adjacent to every v, v ∈ N(x) ∪ N(y)
11            replace occurrences of x or y in l by xy
12            remove x from G'
13            remove y from G'
```

Fig. 9. The greedy coalescing algorithm

Coalescing. The last phase of the algorithm is the coalescing of move related instructions. Coalescing helps a compiler to avoid generating redundant copy instructions. Our coalescing phase is executed in a greedy fashion. For each instruction $a := b$, the algorithm looks for a color c not used in $N(a) \cup N(b)$, where $N(v)$ is the set of neighbors of v. If such a color exists, then the temporaries a and b are coalesced into a single register with the color c. This algorithm is described in Figure 9. Our current coalescing algorithm does not use properties of chordal graphs; however, as future work, we plan to study how coalescing can take benefit from chordality.

Pre-Spilling. To color a graph, we need a number of colors which is at least the size of the largest clique. We now present an approach to removing nodes that will bring the size of the largest clique down to the number of available colors and guarantee that the resulting graph will be colorable with the number of available colors (Theorem 2). Gavril [11] has presented an algorithm *maximalCl*, shown in Figure 10, which lists all the maximal cliques of a chordal graph in $O(|E|)$ time. Our pre-spilling phase first runs *maximalCl* and then the procedure *pre-spilling* shown in Figure 11. Pre-spilling uses a map ω which maps each vertex to an approximation of the number of maximal cliques that contain that vertex. The objective of pre-spilling is to minimize the number of spills. When an interference graph is non-chordal, the *maximalCl* algorithm may return graphs that are not all cliques and so pre-spilling may produce unnecessary spills. Nevertheless, our experimental results in Section 5 show that the number of spills is competitive even for non-chordal graphs.

The main loop of pre-spilling performs two actions: (i) compute the vertex v that appears in most of the cliques of ξ and (ii) remove v from the cliques in which it appears. In order to build an efficient implementation of the pre-spilling algorithm, it is helpful to define a bidirectional mapping between vertices and the cliques in which they appear. Because the number of maximal cliques is

```
procedure maximalCl
1    input: G = (V, E)
2    output: a list of cliques ξ = ⟨Q_1, Q_2, ..., Q_n⟩
3    σ ← MCS(G)
4    For i ← 1 to n do
5        Let v ← σ[i] in
6            Q_i ← {v} ∪ {u | (u, v) ∈ E, u ∈ {σ[1], ..., σ[i − 1]}}
```

Fig. 10. Listing maximal cliques in chordal graphs

```
procedure pre-spilling
1    input: G = (V, E), a list of subgraphs of G: ξ = ⟨Q_1, Q_2, ..., Q_n⟩,
         a number of available colors K, a mapping ω
2    output: a K-colorable subgraph of G
3    R_1 = Q_1; R_2 = Q_2; ... R_n = Q_n
4    while there is R_i with more than K nodes do
5        let v ∈ R_i be a vertex such that ∀u ∈ R_i, ω(v) ≥ ω(u) in
6            remove v from all the graphs R_1, R_2, ..., R_n
7    return R_1 ∪ R_2 ∪ ... ∪ R_n
```

Fig. 11. Spilling intersections between maximal cliques

bounded by $|V|$ for a chordal graph, it is possible to use a bucket list to compute $\omega(v), v \in V$ in $O(1)$ time. After a temporary is deleted, a number of cliques may become K-colorable, and must be removed from ξ. Again, due to the bidirectional mapping between cliques and temporaries, this operation can be performed in $O(|N(v)|)$, where $N(v)$ is the set of vertices adjacent to v. Overall, the spilling algorithm can be implemented in $O(|E|)$.

Theorem 2. *The graph pre-spilling(G,maximalCl(G),K,ω) is K-colorable.*

Proof. Let $\langle Q_1, Q_2, \ldots, Q_n \rangle$ be the output of *maximalCl*(G). Let $R_1 \cup R_2 \cup \ldots \cup R_n$ be the output of *pre-spilling*$(G$,*maximalCl*(G),K,$\omega)$. Let $R_i^\bullet = R_1 \cup R_2 \cup \ldots \cup R_i$ for $i \in 1..n$.

We will show that for all $i \in 1..n$, $R_i^\bullet$ is K-colorable. We proceed by induction on i.

In the base case of $i = 1$, we have $R_1^\bullet = R_1 \subseteq Q_1$ and Q_1 has exactly one node. We conclude that $R_1^\bullet$ is K-colorable.

In the induction step we have from the induction hypothesis that $R_i^\bullet$ is K-colorable so let c be a K-coloring of $R_i^\bullet$. Let v be the node $\sigma[i+1]$ chosen in line 5 of *maximalCl*. Notice that v is the only vertex of Q_{i+1} that does not appear in $Q_1, Q_2, \ldots, Q_i$ so c does not assign a color to v. Now there are two cases. First, if v has been removed by *pre-spilling*, then $R_{i+1}^\bullet = R_i^\bullet$ so c is a K-coloring of $R_{i+1}^\bullet$. Second, if v has not been removed by *pre-spilling*, then we use that R_{i+1} has at most K nodes to conclude that the degree of v in R_{i+1} is at most $K-1$. We have that c assigns a color to all neighbors for v in R_{i+1} so we have a color left to assign to v and can extend c to a K-coloring of $R_{i+1}^\bullet$.

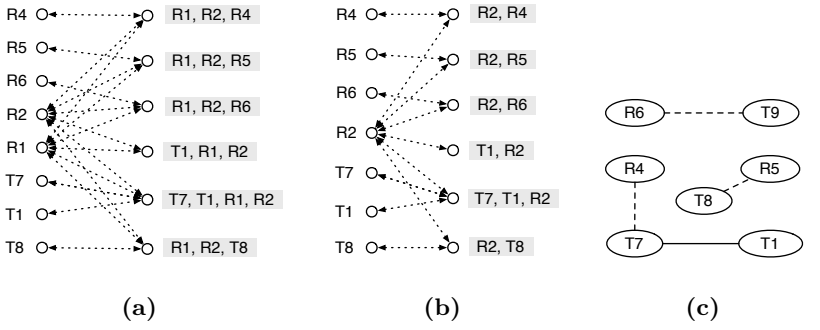


Fig. 12. (a) Mapping between nodes and maximal cliques. (b) Mapping after pruning node `R1`. (c) Interference graph after spilling `R1` and `R2`.

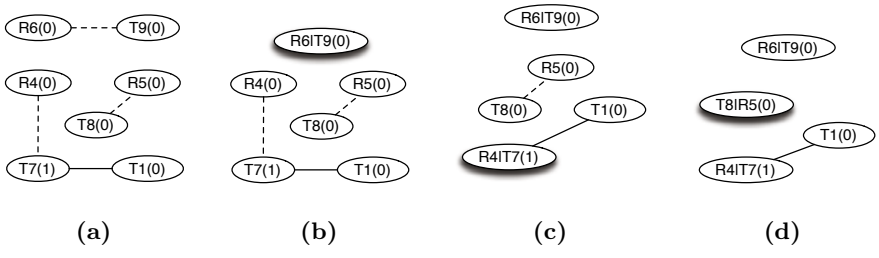
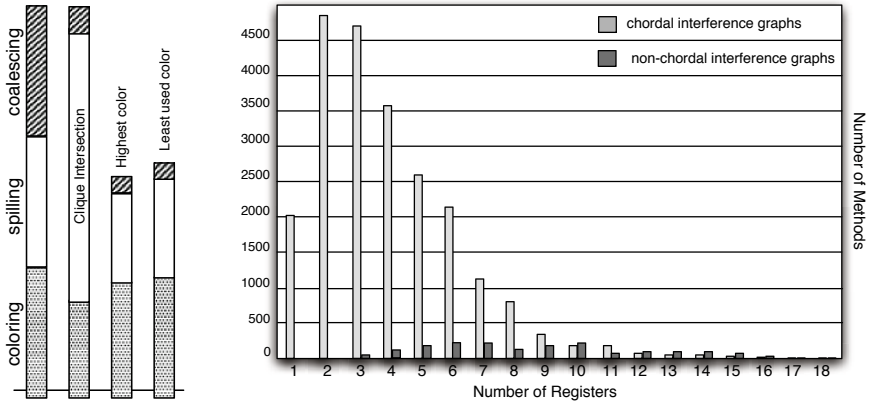


Fig. 13. (a) Coloring produced by the greedy algorithm. (b) Coalescing `R6` and `T9`. (c) Coalescing `R4` and `T7`. (d) Coalescing `R5` and `T8`.

Figure 12 (a) shows the mapping between temporaries and maximal cliques that is obtained from the `gcd(x, y)` method, described in Figure 7 (a). Assuming that the target architecture has two registers, the cliques must be pruned until only cliques of size less than two remain. The registers `R1` and `R2` are the most common in the maximal cliques, and, therefore, should be deleted. The configuration after removing register `R1` is outlined in Figure 12 (b). After the pruning step, all the cliques are removed from ξ. Figure 12 (c) shows the interference graph after the spilling phase.

Figure 13 outlines the three possible coalescings in this example. Coincidentally, two of the move related registers were assigned the same color in the greedy coloring phase. Because of this, their colors do not had to be changed during the coalescing stage. The only exception is the pair (`R4, T7`). In the coalescing phase, the original color of `R4` is changed to the same color of `T7`. Afterwards, the registers are merged.

Complexity Analysis. The coloring phase, as a direct application of maximum cardinality search and greedy coloring, can be implemented to run in $O(|V|+|E|)$ time.



(a) **(b)**

Fig. 14. (a) Time spent on coloring, spilling and coalescing in the different heuristics. (b) Number of registers assigned to methods of the Java 1.5 Standard Library.

Our heuristics for spilling can all can be implemented to run in $O(|E|)$ time. In order to implement spilling of the least-used color, it is possible to order the colors with bucket sort, because the maximum color is bounded by the highest degree of the interference graph plus one. The same technique can be used to order the weight function for the pre-spilling algorithm because the size of the list ξ, produced by the procedure *maximalCl*, is bounded by $|V|$.

Coalescing is the phase with the highest complexity, namely $O(t^3)$, where t is the number of temporaries in the source code. Our coalescing algorithm inspects, for each pair of move related instructions, all their neighbors. It is theoretically possible to have up to t^2 pairs of move related instructions in the target code. However, the number of these instructions is normally small, and our experimental results show that the coalescing step accounts for less than 10% of the total running time (see Figure 14 (a)).

5 Experimental Results

We have built an evaluation framework in Java, using the JoeQ compiler [19], in order to compare our algorithm against the iterated register coalescing. When pre-spilling is used, post-spilling is not necessary (Theorem 2). Our benchmark suite is the entire run-time library of the standard Java 1.5 distribution, i.e. the set of classes in `rt.jar`. In total, we analyzed 23,681 methods. We analyzed two different versions of the target code. One of them is constituted by the intermediate representation generated by JoeQ without any optimization. In the other version, the programs are first converted to single static assignment form (SSA), and them converted back to the JoeQ intermediate representation, by

Table 1. Comparison between our algorithm (NIA) and Iterated Register Coalescing (IRC), including results for the three different spilling heuristics in Section 4

Algorithm	SSA	number of registers	register/ method	spill/ method	Total spills	maximum # spills	coalescing/ moves	running time (s)
NIA	no	18	4.20	0.0044	102	15	0.38	2645.1
Post-spilling	yes	18	4.13	0.0034	81	14	0.72	2769.9
least-used	no	6	3.79	0.43	10,218	30	0.37	2645.0
color	yes	6	3.75	0.51	12,108	91	0.73	2781.7
NIA	no	18	4.20	0.0048	115	15	0.34	2641.5
Post-spilling	yes	18	4.13	0.010	246	63	0.72	2767.0
highest	no	6	3.80	0.50	11,923	33	0.35	2674.3
used color	yes	6	3.75	0.80	19,018	143	0.69	2764.2
NIA	no	18	4.20	0.0044	105	15	0.34	2640.5
Pre-spilling	yes	18	4.13	0.0039	94	17	0.72	2763.2
	no	6	3.78	0.45	10,749	34	0.35	2645.8
	yes	6	3.75	0.49	11,838	43	0.70	2765.1
IRC	no	18	4.25	0.0050	115	16	0.31	2644.1
	yes	18	4.17	0.0048	118	27	0.70	2823.2
	no	6	3.81	0.50	11,869	32	0.31	2641.5
	yes	6	3.77	0.57	13,651	86	0.66	2883.7

substituting the *phi* functions by copy instructions. In the former case, approximately 91% of the interference graphs produced are chordal. In the latter, the percentage of chordal graphs is 95.5%.

Table 1 shows results obtained by the iterative algorithm (IRC), and our non-iterative register allocator (NIA). The implementation of both algorithms attempts to spill the minimum number of registers. As it can be seen in the table, our technique gives better results than the traditional register allocator. It tends to use less registers per method, because it can find an optimum assignment whenever the interference graph is chordal. Also, it tends to spill less temporaries, because, by removing intersections among cliques, it decreases the chromatic number of several clusters of interfering variables at the same time. Notably, for the method `coerceData`, of the class `java.awt.image.ComponentColorModel`, with 6 registers available for allocation, the pre-spilling caused the eviction of 41 temporaries, whereas Iterated Register Coalescing spilled 86. Also, because our algorithm tends to spill fewer temporaries and to use fewer registers in the allocation, it is able to find more opportunities for coalescing. The Iterated register coalescing and our algorithm have similar running times. The complexity of a single iteration of the IRC is $O(|E|)$, and the maximum number of iterations observed in the tests was 4; thus, its running time can be characterized as linear. Furthermore, both algorithms can execute a cubic number of coalescings, but, in the average, the quantity of copy instructions per program is small when compared to the total number of instructions.

Table 2 compares the two algorithms when the interference graphs are chordal and non-chordal. This data refers only to target programs after SSA elimination.

Table 2. Comparative performance of our spilling heuristics for chordal and non-chordal interference graphs

Algorithm	chordal graph	number of registers	register/ method	spill/ method	Total spills	maximum # spills	coalescing/ moves
NIA	no	18	8.17	0.054	61	17	0.75
Pre-spilling	no	6	5.77	4.55	5173	43	0.79
	yes	18	3.92	0.0015	33	6	0.69
	yes	6	3.65	0.29	6665	31	0.68
IRC	no	18	8.39	0.062	71	27	0.74
	no	6	5.79	4.89	5562	86	0.66
	yes	18	3.97	0.0015	34	6	0.67
	yes	6	3.68	0.39	8089	45	0.67

Table 3. Results obtained from the allocation of registers to 27,921 interference graphs generated from ML code

Algorithm	chordal graph	Total of spills	maximum number of spills	coalescing/ moves	allocation time (s)
Post-spilling least used color	yes	1,217	84	0.97	223.8
	no	63	14	0.94	
Post-spilling highest used color	yes	1,778	208	0.97	222.9
	no	80	20	0.94	
Pre-spilling	yes	1,127	86	0.97	482.3
	no	1,491	23	0.93	

In general, non-chordal interference graphs are produced by complex methods. For instance, methods whose interference graphs are non-chordal use, on average, 80.45 temporaries, whereas the average for chordal interference graphs is 13.94 temporaries.

The analysis of methods whose interference graphs are chordal gives some insight about the structure of Java programs. When an interference graph is chordal, the mapping between temporaries and registers is optimal, i.e. it uses the smallest possible number of registers. Figure 14 (b) shows the relation between number of methods of the Java Library and the minimum number of registers necessary to handle them. Only methods that could be colored with less than 18 colors (99.6%) are shown. Allocation results for methods whose interference graph are non-chordal are also presented, even though these may not be optimal.

Figure 14 (a) compares the amount of time spent on each phase of the algorithm when different spilling heuristics are adopted. The time used in the allocation process is a small percentage of the total running time presented in Table 1 because the latter includes the loading of class files, the parsing of bytecodes, the liveness analysis and the construction of the interference graph. When pre-spilling is used, it accounts for more than half the allocation time.

We have also tested our register allocation algorithm on the 27,921 interference graphs published by George and Appel. Those graphs were generated by the standard ML compiler of New Jersey compiling itself [3]. Our tests have shown that 95.7% of the interference graphs are chordal when the interferences between pre-colored registers and temporaries are not taken into consideration. The compilation results are outlined in Table 3. The graphs contain 21 pairwise interfering pre-colored registers, which represent the machine registers available for the allocation. Because of these cliques, all the graphs, after spilling, demanded exactly 21 colors. When the graphs are chordal, pre-spilling gives the best results; however, this heuristic suffers a penalty when dealing with the non-chordal graphs, because they present a 21-clique, and must be colored with 21 registers. In such circumstances, the procedure *maximalCl* from Figure 10 have listed some false maximal cliques, and unnecessary spills have been caused. Overall, the spilling of the least-used colors gives the best results. The execution times for analyzing the ML-compiler-based benchmarks are faster than those for analyzing the Java Library because the latter set of timings includes the times to construct the interference graphs.

6 Conclusion

This paper has presented a non-iterative algorithm for register allocation based on the coloring of chordal graphs. Chordal graphs present an elegant structure and can be optimally colored in $O(|V| + |E|)$ time. For the register allocation problem, we can find an optimal allocation in time linear in the number of interferences between live ranges, whenever the interference graph is chordal. Additionally, our algorithm is competitive even when performing register allocation on non-chordal inputs.

In order to validate the algorithm, we compared it to iterated register coalescing. Our algorithm allocates fewer registers per method and spills fewer temporaries. In addition, our algorithm can coalesce about the same proportion of copy instructions as iterated register coalescing.

In addition to being efficient, our algorithm is modular and flexible. Because it is non-iterative, it presents a simpler design than traditional algorithms based on graph coloring. The spill of temporaries can happen before or after the coloring phase. By performing spilling before coloring, it is possible to assign different weights to temporaries in order to generate better code. Our implementation and a set of interference graphs generated from the Java methods tested can be found at `http://compilers.cs.ucla.edu/fernando/projects/`.

Acknowledgments. We thank Ben Titzer and the reviewers for helpful comments on a draft of the paper. Fernando Pereira is sponsored by the Brazilian Ministry of Education under grant number 218603-9. We were supported by the National Science Foundation award number 0401691.

References

1. Christian Andersson. Register allocation by optimal graph coloring. In *12th Conference on Compiler Construction*, pages 34–45. Springer, 2003.
2. Andrew W Appel and Lal George. Optimal spilling for cisc machines with few registers. In *International Conference on Programming Languages Design and Implementation*, pages 243–253. ACM Press, 2001.
3. Andrew W Appel and Lal George. 27,921 actual register-interference graphs generated by standard ML of New Jersey, version 1.09–`http://www.cs.princeton.edu/~appel/graphdata/`, 2005.
4. Anne Berry, Jean Blair, Pinar Heggernes, and Barry Peyton. Maximum cardinality search for computing minimal triangulations of graphs. *Algorithmica*, 39(4):287–298, 2004.
5. Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, 1994.
6. Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial-time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*. ACM Press, 2005.
7. Zoran Budimlic, Keith D Cooper, Timothy J Harvey, Ken Kennedy, Timothy S Oberg, and Steven W Reeves. Fast copy coalescing and live-range identification. In *International Conference on Programming Languages Design and Implementation*, pages 25–32. ACM Press, 2002.
8. G J Chaitin. Register allocation and spilling via graph coloring. *Symposium on Compiler Construction*, 17(6):98–105, 1982.
9. Maria Chudnovsky, Gerard Cornuejols, Xinming Liu, Paul Seymour, and Kristina Vuskovic. Recognizing berge graphs. *Combinatorica*, 25:143–186, 2005.
10. G A Dirac. On rigid circuit graphs. In *Abhandlungen aus dem Mathematischen Seminar der Universiat Hamburg*, volume 25, pages 71–75. University of Hamburg, 1961.
11. Fanica Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SICOMP*, 1(2):180–187, 1972.
12. Lal George and Andrew W Appel. Iterated register coalescing. *Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):300–324, 1996.
13. M Grotschel, L Lovasz, and A Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.
14. Sebastian Hack. Interference graphs of programs in SSA-form. Technical report, Universitat Karlsruhe, 2005.
15. Fernando M Q Pereira and Jens Palsberg. Register allocation after SSA elimination is NP-complete. Manuscript, 2005.
16. Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.
17. Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.
18. Douglas B West. *Introduction to Graph Theory*. Prentice Hall, 2nd edition, 2001.
19. John Whaley. Joeq:a virtual machine and compiler infrastructure. In *Workshop on Interpreters, virtual machines and emulators*, pages 58–66. ACM Press, 2003.
20. Mihalis Yannakakis and Fanica Gavril. The maximum k-colorable subgraph problem for chordal graphs. *Information Processing Letters*, 24(2):133–137, 1987.