Register Allocation via Coloring of Chordal Graphs

Fernando Magno Quintão Pereira
Jens Palsberg

UCLA
University of California, Los Angeles

Abstract. We present a simple, linear-time algorithm for register allocation which is competitive with the iterated register coalescing algorithm of George and Appel. We base the new algorithm on the observation that more than 95% of the methods in the Java 1.5 library have chordal interference graphs. A greedy algorithm can color a chordal graph optimally in linear time, and we can easily add powerful heuristics for spilling and coalescing. Our experimental results 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. Early work in this area includes Ershov's [11] algorithm from 1958 for optimal register allocation on expression trees. After the seminal work of Chaitin [8] in 1982, 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 an 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 each edge connects two temporary variables whose live ranges interfere. We can then use a coloring algorithm, in which colors represent machine registers, to perform register allocation.

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,

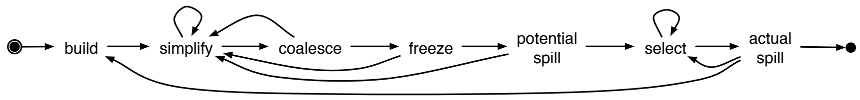


Fig. 1. The iterated register coalescing algorithm.

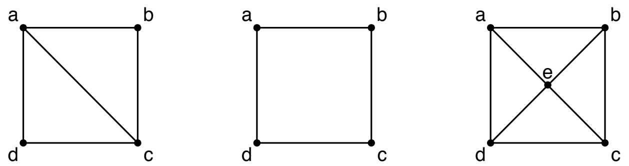
and Torczon [6], the Iterated Register Coalescing algorithm of George and Appel [13], and the Linear Scan algorithm of Poletto and Sarkar [15]. Among the polynomial-time algorithms, the best in terms of resulting code quality appears to be iterated register coalescing. The high quality comes at the price of handling spilling and coalescing of temporary variables in a complex way. Figure 1 illustrates the complexity of iterated register coalescing by depicting the main phases and complex 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. Notably, more than 95% of the methods in the Java Standard Library have chordal interference graphs. 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 [12]. 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 linear-time coloring algorithm for chordal graphs, and which contains powerful heuristics for spilling and coalescing. We show that our algorithm is competitive with the iterated register coalescing algorithm. 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 2 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.



(a) **(b)** **(c)**

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

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. 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 27,921 graphs. Although 1-perfect graphs appear often in practice, Figure 14 shows a Java method whose interference graph is not 1-perfect: its largest clique has two nodes, but three colors are needed to color it.

Brisk et al. [7] has characterized the structure of interference graphs derived from *regular programs*. A regular program is a program in SSA form in which every use of a temporary is preceded by a definition of that temporary in the control flow (see the Appendix for a definition of SSA form). Brisk et al. proved that a regular program yields a perfect interference graph. Although perfect graphs can be colored in polynomial time, the practical consequences of Brisk's proof 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 code generation. After insertion of copy instructions, the target code may not have a perfect interference graph. It remains an open problem to map an optimal coloring of the SSA form to an optimal coloring of the program after insertion of copy instructions.

For 1-perfect graphs, recognition and coloring is NP-complete. In contrast, 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 [14] 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 an induced subgraph in which every pair of vertices is adjacent. A vertex $v, v \in V$ is called *simplicial* if its neighborhood in G is a clique. A *Simplicial Elimination Ordering* of G is a bijection $\sigma : V(G) \to \{1 \ldots |V|\}$, 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 3, is a linear-time 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 $0 \le c \le \Delta(G) + 1$. If we give *greedy coloring* a simplicial elimination ordering of the vertices, then the greedy algorithm yields an optimal coloring [12]. In other words, *greedy coloring* is optimal for chordal graphs.

procedure greedy coloring
    input: $G = (V, E)$, a sequence of vertices ν
    output: a mapping m, $m(v) = c, 0 \le c \le \Delta(G) + 1, v \in V$
    For all $v \in \nu$ **do** $m(v) \leftarrow \perp$
    For $i \leftarrow 1$ **to** $|\nu|$ **do**
        let c be the lowest color not used in $N(\nu(i))$ **in**
            $m(\nu(i)) \leftarrow c$

Fig. 3. The greedy coloring algorithm.

Given a chordal graph G, algorithms such as *Maximum Cardinality Search* (MCS) [17] determine a simplicial elimination ordering σ of its vertices 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 unit the weight of the neighbors of v, and

procedure MCS

input: $G = (V, E)$

output: a simplicial elimination ordering $\sigma = v_1, \ldots, v_n$

For all $v \in V$ **do** $\lambda(v) \leftarrow 0$

For $i \leftarrow 1$ **to** $|V|$ **do**

  let $v \in V$ **be** a vertex such that $\forall u \in V, \lambda(v) \geq \lambda(u)$ **in**

    $\sigma(i) \leftarrow v$

    For all $u \in V \cap N(v)$ **do** $\lambda(u) \leftarrow \lambda(u) + 1$

    $V \leftarrow V - \{v\}$

Fig. 4. The maximum cardinality search algorithm.

starts a new phase. A version of MCS, as published by Berry et al. [4], is given in Figure 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 5, namely coloring, spilling, and coalescing, plus an optional phase called *pre-spilling* that precedes the coloring phase. Coalescing must be the last stage because, after merging nodes, the resulting interference graph can be non-chordal. Our algorithm uses the *MCS* procedure (Figure 4) 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, our approach produces competitive results even for non-chordal graphs. We have 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.

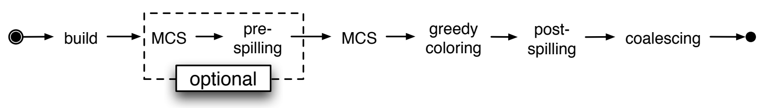
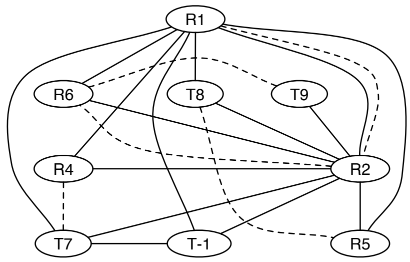


Fig. 5. The main phases of our algorithm.

In order to illustrate the basic principles underling our algorithm, we will show how it allocates registers for the program in Figure 6 (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 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 6 (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.

```
int gcd (int R1, int R2)
1.  IFCMP_I_EQ    ..  R2   0    (12);
2.  ZERO_CHECK_I  ..  T-1  R2;
3.  DIV_I         T7  R1   R2;
4.  CHECK_EX      T-1
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. 6. (a) Euclid's algorithm. (b) Interference graph generated for gcd().

4.1 Greedy Coloring

In order to assign machine registers to variables, the *greedy coloring* procedure of Figure 3 is fed with an ordering of the vertices of the interference graph, as produced by the *MCS* procedure. From the graph shown in Figure 6 (b), MCS produces the ordering: ⟨ `T7, R1, R2, T-1, R5, R4, T8, R6, T9` ⟩, and *greedy coloring* then produces the mapping between temporaries and colors that is outlined in Figure 7 (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 step of the algorithm.

4.2 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. Despite of all the properties of chordal graphs, optimal spilling for chordal graphs is NP-complete, as shown by Yannakakis and Gavril in 1987 [20].

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 chosing 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 in bounded by the number of maximum 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. As we will show in Section 5, it is possible to spill less nodes by removing the least-used color.

For our running example, Figure 7 (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.

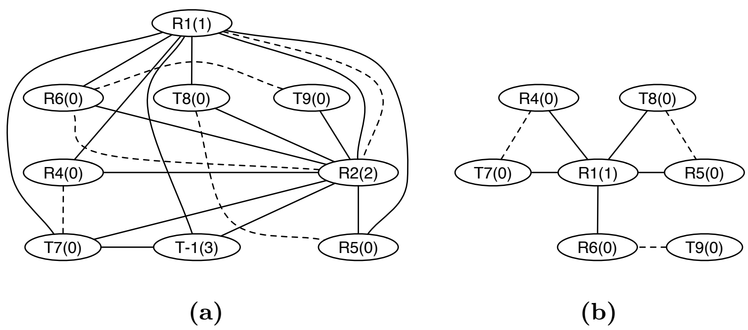


Fig. 7. (a) Colored interference graph. (b) Interference graph after spilling the highest colors.

4.3 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, based on a colored graph. 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 color exists, then the temporaries a and b are coalesced on a single register with the color c. This algorithm is described in Figure 8.

procedure coalescing
    input: list l of copy instructions, $G = (V, E)$, K
    output: G', the coalesced graph G
    let $G' = G$ **in**
    for all $x := y \in l$ **do**
        let S_x **be** the set of colors in $N(x)$
        let S_y **be** the set of colors in $N(y)$
        if there exists $c, c < K, c \notin S_x \cup S_y$ **then**
            let $xy, xy \notin V$ **be** a new node
            add xy to G' with color c
            make xy adjacent to every $v, v \in N(x) \cup N(y)$
            replace occurrences of x or y in l by xy
            remove x from G'
            remove y from G'

Fig. 8. The greedy coalescing algorithm.

4.4 Pre-spilling

To color a graph, we need a number of colors which is at least the size of the largest clique. We will 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 [12] has presented an algorithm *maximalCl*, shown in Figure 9, 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 10. 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.

The main loop of pre-spilling performs two main 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 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 $O(|E|)$.

procedure maximalCl
    input: $G = (V, E)$
    output: a set of cliques $\xi = \{Q_1, Q_2, \ldots, Q_n\}$
    $\sigma \leftarrow$ **MCS(G)**
    For $i \leftarrow 1$ **to** n **do**
        Let $v \leftarrow \sigma[i]$ **in**
            $Q_i \leftarrow \{v\} \cup \{u \mid (u, v) \in E, u \in \{\sigma[1], \ldots, \sigma[i-1]\}\}$

Fig. 9. Listing maximal cliques in chordal graphs.

procedure pre-spilling
    input: a list of cliques $\xi = \langle Q_1, Q_2, \ldots, Q_n \rangle$,
        $G = (V, E)$, number of available colors K, a mapping ω
    output: a K colorable graph G'
    while there is Q_i with at least K nodes **do**
        remove Q_i from ξ
        let $v \in Q_i$ **be** a vertex such that $\forall u \in Q_i, \omega(v) \geq \omega(u)$ **in**
            remove v from all cliques in ξ

Fig. 10. Spilling intersections between maximal cliques.

Theorem 2. *For any simple graph G, after procedures maximalCl and then pre-spilling(G, K, ω) are executed, the resulting graph will be K-colorable, where* $K > 0$.

Proof. The proof is by induction on the ordering ξ produced by the procedure *maximalCl* shown in Figure 9.

Basis: Let Q_1 be the first clique in ξ. Q_1 has just one vertex, and is K colorable.

Induction Hypothesis: If all $\langle Q_1, Q_2, \ldots, Q_i \rangle$ have size at most K, then the graph $G_i = Q_1 \cup Q_2 \cup \ldots Q_i$ is K colorable.

Induction Step: Consider the graph G_{i+1}, and let v_{i+1} be the vertex in $\sigma(i+1)$. According to the procedure *maximalCl*, all the neighbors of v_{i+1} that are in Q_{i+1} come before v_{i+1} in σ. Therefore, they have been assigned colors, by the induction hypothesis. These vertices can be adjacent, or can be non-adjacent. They will be pairwise adjacent if G is a chordal graph. Thus, v_{i+1} has the highest degree in Q_{i+1}, independent on G being chordal or not. By hypothesis, Q_{i+1} has at most K nodes, and is K colorable because $(\Delta(Q_{i+1})+1) \leq K$. All the vertices in Q_{i+1} but v_{i+1} have been colored, with at most $K-1$ colors. The *K-th* color, if necessary, will be assigned to v_{i+1}.

Figure 11 (a) shows the mapping between temporaries and maximal cliques that is obtained from the `gcd(x, y)` method, described in Figure 6 (a). Assuming that the target architecture has two registers, the cliques must be pruned until only cliques of size less than two remains. 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 11 (b). After the pruning step, all the cliques are removed from ξ. Figure 11 (c) shows the interference graph after the spilling phase.

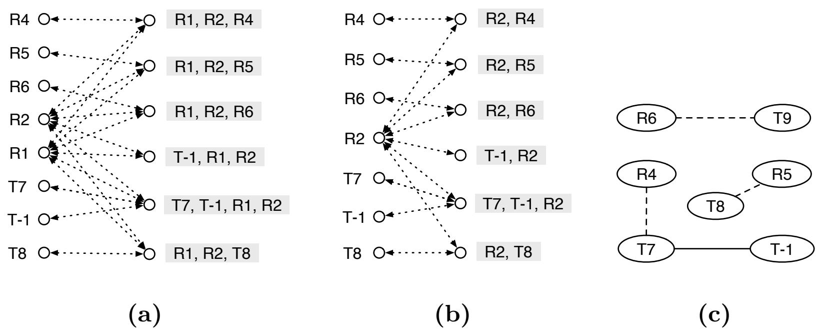


Fig. 11. (a) Mapping between nodes and maximal cliques. (b) Mapping after pruning node `R1`. (c) Interference graph after spilling `R1` and `R2`.

Figure 12 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 it, 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.

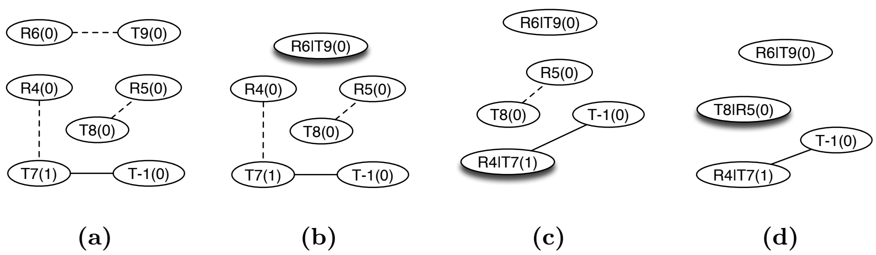


Fig. 12. (a) Coloring produced by the greedy algorithm. (b) Coalescing `R6` and `T9`. (c) Coalescing `R4` and `T7`. (d) Coalescing `R5` and `T8`.

4.5 Complexity Analysis

The coalescing is the most costly phase of our algorithm, because it needs a number of operations that can be as high as $O(t^3)$, where t is the number of temporaries in the source code. The greedy coalescing algorithm described in Section 4.3 inspects, for each pair of move related instructions, all the neighbors of the two involved nodes. It is theoretically possible that there be up to t^2 pairs of move related instructions in the target code. In this case, the cubic complexity will arise if every temporary variable interfere with every other temporary variable. However, our experimental results show that the coalescing step accounts for less than 10% of the total running time.

Regarding the complexity of the spilling algorithms, all our heuristics can be implemented to execute $O(|E|)$ number of operations. 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 in the interference graph. 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 $|V|$.

Finally, the coloring phase, as a direct application of maximum cardinality search and greedy coloring, can be implemented to run in $O(|V|+|E|)$. We would like to emphasize that the algorithms are the same for chordal and non-chordal graphs; therefore, the fact that the interference graph can be non-chordal does not increase the complexity of any of the steps of our algorithm.

5 Experimental Results

In order to validate the algorithm, we have built an evaluation framework using the JoeQ compiler [19], and we have compared our algorithm to iterated register coalescing. We have implemented the different heuristics discussed in Section 4.2 to perform spilling. When pre-spilling is used, post-spilling is not necessary, as shown in Theorem 2. All the implementations perform register allocation for programs written in Java. Our benchmark suite is the complete library of the standard Java 1.5 distribution, i.e. the set of classes in `rt.jar`. In total, 23,681 methods have been analyzed, and approximately 91% of the interference graphs produced are chordal graphs. Two different versions of the target code have been used. 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 substituting the *phi* functions by copy instructions. In this second mode the percentage of chordal graphs is bigger: 95,5%. The execution of the spilling step before the coloring phase increases even more the number of chordal graphs. The removal of a vertex cannot change a chordal graph into non-chordal graph; on the other hand, the removal of a vertex can change a non-chordal graph into chordal graph by eliminating chordless cycles.

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`, our algorithm caused the spilling of 41 temporaries, whereas Iterated Register Coalescing spilled 86. In this case, the algorithms can use 6 registers for the allocation. Also, because our algorithm tends to spill less temporaries, and to use less registers in the allocation, it is able to find more opportunities for coalescing.

Alg	SSA	number of registers	register/ method	spill/ method	Total spills	maximum # spills	coalescing/ moves	running time
NIA	no	18	4.20	0.0044	102	15	0.38	44m5.128s
Spilling	yes	18	4.13	0.0034	81	14	0.72	46m95.892s
least-used	no	6	3.79	0.43	10,218	30	0.37	44m40.95s
color	yes	6	3.75	0.51	12,108	91	0.73	46m21.744s
NIA	no	18	4.20	0.0048	115	15	0.34	44m09.49s
Spilling	yes	18	4.13	0.010	246	63	0.72	46m67.98s
highest	no	6	3.80	0.50	11,923	33	0.35	44m34.293s
used color	yes	6	3.75	0.80	19,018	143	0.70	46m44.18s
NIA	no	18	4.20	0.0044	105	15	0.34	44m0.450s
Spilling	yes	18	4.13	0.0039	94	17	0.72	46m39.151s
clique	no	6	3.78	0.45	10,749	34	0.35	44m5.834s
intersection	yes	6	3.75	0.49	11,838	43	0.69	46m58.099s
IRC	no	18	4.25	0.0050	115	16	0.21	44m4.121s
	yes	18	4.17	0.0048	118	27	0.70	47m31.211s
	no	6	3.81	0.50	11,869	32	0.21	44m13.479s
	yes	6	3.77	0.57	13,651	86	0.66	48m31.661s

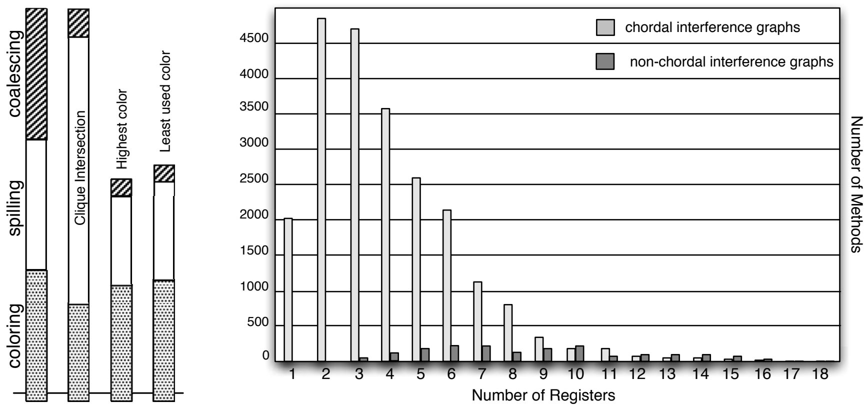
Table 1. Comparison between our algorithm (NIA) and Iterated Register Coalescing (IRC). Results for the three different heuristics outlined in Figure 5 are presented.

Table 2 compares the two algorithms when the interference graphs are chordal, and when they are not. In this case the source code has been converted to SSA form. 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. Methods with chordal interference graphs use 13.94 temporaries.

The analysis of methods whose interference graphs are chordal gives some insight about the structure of Java programs. When the interference graph is chordal, the mapping between temporaries and registers is optimal, i.e. it uses the smallest possible number of registers. Figure 13 (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, event though these are not optimal.

Alg	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
spilling	no	6	5.77	4.55	5173	43	0.79
clique	yes	18	3.92	0.0015	33	6	0.69
intersections	yes	6	3.65	0.29	6665	31	0.68
	no	18	8.39	0.062	71	27	0.74
IRC	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 2. Comparative performance of register allocation algorithms when the interference graphs are chordal, and when they are not.



(a) **(b)**

Fig. 13. (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.

Iterated register coalescing and our algorithm have similar running times. The complexity of a single iteration of the iterated register coalescing is $O(|E|)$. As discussed in Section 4.5, $O(|E|)$ is also the running complexity of maximum cardinality search. The iterated register coalescing tends to resolve the great majority of methods in a single iteration. The highest number of iterations observed in the Java tests was 4; thus, its time can be characterized as linear in the number of interferences. Both algorithms can execute a cubic number of coalescings, although, in the average, the quantity of copy instructions per program is small, when compared to the total number of instructions. Figure 13 (a) compares the amount of time spent on coloring, spilling and coalescing by the different heuristics implemented.

We have also tested our register allocation algorithm on the 27,921 interference graphs published by George and Appel. Such graphs have been 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. The compilation results are outlined in Table 3. The graphs contain 21 pre-colored temporaries, which represent the machine registers available for the allocation. Each of them interfere with each other. Because of these cliques, all the graphs, after spilling, demanded exactly 21 colors. When the graphs are chordal, the spilling by removal of clique intersections yields the best results; however, this heuristic suffers a penalty when dealing with non-chordal ML graphs. They present a 21-clique, and must be colored with 21 registers. In such circumstances, the procedure *maximalCl* from Figure 9 have listed some false maximal cliques, and unnecessary spills have been caused. Overall, the spilling of the least-used colors present the best results. Concerning the executing time, these tests are faster than those performed in the Java Library, because in that case it was necessary to load class files, to compile them to a specific intermediate representation, and to perform liveness analyses in order to generate the interference graphs.

Alg	chordal graph	Total of spills	maximum number of spills	coalescing/ moves	allocation time (s)
Least used color	yes	1,217	84	0.97	223.750
	no	63	14	0.94	
Highest used color	yes	1,778	208	0.97	222.890
	no	80	20	0.94	
clique intersections	yes	1,127	86	0.97	482.340
	no	1,491	23	0.93	

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

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 linear time. Nevertheless, it is important to emphasize that, although our algorithm makes use of properties of chordal graphs, it is competitive, even when performing register allocation on non-chordal inputs.

In order to validate the algorithm, we compared it to the iterated register coalescing. Our algorithm has been able to allocate less registers per method, and to spill less temporaries. In addition, it can coalesce about the same proportion of copy instructions as the iterative algorithm does.

Besides 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.

Finally, 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/`.

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. Bela Bollobas. The chromatic number of random graphs. *Combinatorica*, 8:49 – 55, 1988.
6. 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.
7. 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.
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. A P Ershov. On programming of arithmetic operations. *Communications of the ACM*, 1(8):3 – 6, 1958.
12. 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.
13. Lal George and Andrew W Appel. Iterated register coalescing. *Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):300 – 324, 1996.
14. M Grotschel, L Lovasz, and A Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169 – 197, 1981.
15. Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895 – 913, 1999.
16. B K Rosen, F K Zadeck, and M N Wegman. Global value numbers and redundant computations. In *5th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12 – 27. ACM Press, 1988.
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.

A The case of non-chordal graphs

It is possible to give examples of programs whose interference graphs are non-chordal. Chordless cycles are due to the existence of *intermittent variables*, that is, variables whose live ranges are non-continuous. Discontinuities exist because of the redefinition of variables, and because of branches in the target code. One way to reduce the number of discontinuities in the live ranges of variables is to convert the target code to *Single Static Assignment Form*, or *SSA* [16]. In this format, every variable is defined only once. However, even programs in SSA form can present non-chordal interference graphs.

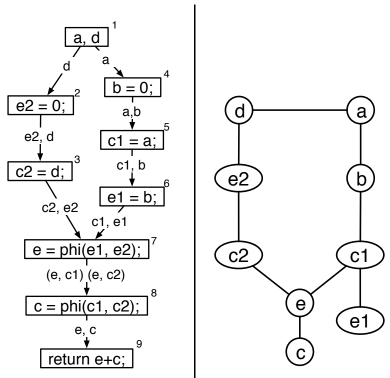
The code exhibited in Figure 14 (a) is such an example. The corresponding control flow graph, in SSA form, is exhibited in Figure 14 (b). The *phi* function is an abstraction used in programs in SSA form, and it allows to merge different temporaries into a single definition. In the given example, $e = phi(e_1, e_2)$ will return e_2 if control reaches block 7 throw block 3, and will return e_1 if control reaches block 7 throw block 6. Notice that *phi* functions are abstract constructions. In order to generate actual code from a program in SSA form, it is necessary to substitute the *phi* abstractions by copy instructions. Also, *phi* constructs differ from ordinary functions in the sense that the live range of their parameters do not interfere with each other. The parameters of a *phi* function can be assigned the same register, or different registers. The interference graph for this example is outlined in Figure 14 (c).

In our algorithm, the implementations of spilling and coloring use properties of chordal graphs in order to produce good results, but they do not have to be changed in order to handle non-chordal graphs. In face of a non-chordal graph, the final coloring is not necessarily optimal. Our algorithm can be thought as the result of applying the greedy coloring on a random graph. Given a graph G, the maximum number of colors necessary to color it is bounded by $\Delta(G) + 1$, where $\Delta(G)$ is the largest degree of a vertex in G. This upper bound can be arbitrarily greater than the optimum; however, results from theory of probabilities assure that, if the vertices are randomly given to the greedy algorithm, then the final coloring will be between the optimum and approximately twice the optimum for almost every possible graph [5].

When the interference graph is non-chordal, it is not possible to list its maximal cliques by means of the strategy described in Section 4.2. If spilling before coloring is used, the algorithm will deal with clusters of nodes that do not constitute complete subgraphs, and unnecessary spills can happen. Nevertheless, our experimental results presented in Section 5 show that, when the weight function denotes the number of maximal cliques in which a node is present, the number of spills is competitive even if the underlying interference graph is non-chordal.

```
int m(int x, a, d) {
    int b, c;
    if(x > 0) {
        e = 0;
        c = d;
    } else {
        b = 0;
        c = a;
        e = b;
    }
    return c + e;
}
```

(a)



(b) **(c)**

Fig. 14. (a) Example of non-chordal program, (b) Control-flow graph in SSA form, (c) The interference graph.

Our tests have shown that removing intersections between cliques, or spilling the least-used colors cause the spill of less temporaries when dealing with non-chordal graphs than the iterated register coalescing algorithm does.