Improvements to Conservative and Optimistic Register Coalescing ⋆

Florent Bouchez, Alain Darte, and Fabrice Rastello

Compsys Project
Université de Lyon, LIP, ENS Lyon, CNRS, INRIA, France.
Research Report n^o RR2007-41
Firstname.Lastname@ens-lyon.fr

Abstract. Register coalescing is used, as part of register allocation, to reduce the number of register copies. Developing efficient register coalescing heuristics is particularly important to get rid of the numerous move instructions introduced by code transformations such as static single assignment, among others. The challenge is to find a good trade-off between a too aggressive strategy that could make the interference graph uncolorable, possibly increasing the spill (transfer to memory), and a too conservative strategy that preserves colorability but leaves too many moves. The two main approaches are "iterated register coalescing" by George and Appel and "optimistic coalescing" by Park and Moon. The first one coalesces moves, one by one, in a conservative way. In the second one, moves are first coalesced regardless of the colorability, then some coalescings are undone to reduce spilling. Previous experiments show that optimistic coalescing outperforms conservative coalescing. We show that, with a more involved conservative strategy, incremental conservative coalescing can be as efficient as optimistic coalescing. We also develop a more aggressive optimistic approach with a different de-coalescing phase. The combination of the two approaches leads to about 10% improvements compared to traditional optimistic coalescing.

1 Introduction

If high-level source codes contain few copy (move) instructions, this is not the case after the many optimization phases that occur in a compiler. For example, static single assignment (SSA) [9] introduces virtual "ϕ-functions"at joint points of the control-flow graph, which semantically correspond to parallel copy instructions placed in blocks on the incoming control-flow edges. When going out of SSA, these copies must be carefully removed [18, 17] or one can rely on a later register allocation phase to eliminate them. Another example is live-range splitting, which is mandatory to improve spilling (transfers between registers and memory). The extreme situation is, as in [2], when live-ranges are split at each program point so as to formulate the spilling problem as an integer linear program. Loads and stores are then nicely optimized but many copies are created that need to be removed later. Finally, register moves can also be added to cope, in a simple way, with register constraints or calling conventions. A later phase is supposed to be able to remove them. In other words, there are many reasons to try to get rid of copy instructions at assembly-code level. Register coalescing does this during register allocation, thus tightly connected to spill optimization and register assignment.

⋆ This work is supported by a contract with STMicroelectronics, Grenoble

Chaitin et al. introduced graph coloring register allocation in [8]. Each node of the interference graph G corresponds to a program variable and an edge between two nodes means that they interfere, i.e., cannot share the same register. If k registers are available, a k-coloring of G, if there is one, gives a correct assignment of variables to registers. A move can be deleted if the two variables involved in the copy are assigned the same register. *Coalescing* enforces this common assignment by merging the two corresponding nodes in the graph. For coloring, all Chaitin-like register allocators rely on the following "simplify" rule to assign colors to variables: **A node x with $< k$ neighbors is always colorable no matter how $G \setminus \{x\}$ is colored**, it can thus be removed (simplified) from the graph and pushed on a stack. If this *simplify phase* removes all nodes, G is *greedy-k-colorable*. It is also k-colorable as each node can be popped from the stack and colored with one of the colors not used by its $< k$ neighbors previously popped. This second phase is the *select phase*. Spilling and coalescing are always optimized with this simplify rule in mind, because it is the only simple rule to color general graphs.

Spilling is usually done with Briggs et al. mechanism [5, 6]. If, in the simplify phase, all nodes have at least k neighbors, one is removed from the interference graph, pushed on the stack, and marked as a *potential spill*. If, during the select phase, no color is available for a potential spill, it is an *actual spill*, and loads/stores are inserted. In the initial algorithm of Chaitin et al., spilling was decided in the simplify phase, missing the opportunity of "luck" for coloring in the select phase. For coalescing, the initial proposal of Chaitin et al. was to coalesce moves, before the simplify phase, in an *aggressive* fashion, i.e., regardless of the colorability of the resulting graph. The consequence is that many moves are indeed deleted but the number of potential, and even actual, spills increases. To address this issue, in addition to aggressive coalescing, two main approaches were explored, *conservative* coalescing and *optimistic* coalescing.

Conservative coalescing consists in coalescing a move only when one can ensure that the resulting graph is k-colorable if the initial graph was. In Briggs' test [5, 6], a move is coalesced only if the resulting node has at most $(k-1)$ neighbors with degree at least k. This test was used by Briggs et al., after a first phase of live-range splitting followed by aggressive coalescing limited to original moves, to make sure that moves introduced by live-range splitting are not coalesced back too much. George and Appel [10] then proposed *iterated register coalescing*, a fully-conservative approach, where conservative coalescing is inter-mixed with the simplify phase and no aggressive coalescing is performed. A second complementary test, George's test, is proposed for coalescing with pre-colored nodes: a node can be merged with a pre-colored node if all neighbors of the first are neighbors of the second. As these two tests are fast but not exact (when the test fails, it does not mean that the move cannot be coalesced conservatively), better results are obtained if moves are tested several times while simplifying nodes. For that, worklists of possibly-coalescable moves are created and updated during the simplify phase. This increases the running time of the complete register allocator, even with the smart implementation strategies defined by Leung and George in [14].

Iterated register coalescing is now very popular due to its clean and reproducible design although, in terms of moves optimization, optimistic coalescing [15], proposed by Park and Moon, was proved to be more efficient. Optimistic coalescing relies on the fact that, although coalescing can increase the degree of the merged nodes, it can also

decrease the degree of common neighbors and thus have positive effects. Thus, it seems better to first perform aggressive coalescing and to decide, during the select phase, to undo some coalescing to avoid spilling. In Park and Moon algorithm, when a coalesced node is to be spilled, it is split back (de-coalesced) into separate nodes, some of them are colored with a common color, the other are either colorable at the very end of the select phase or spilled. Optimistic coalescing takes its benefit from the aggressive phase and, despite its naive de-coalescing phase, outperforms iterated register coalescing.

Recent work [2, 7, 3, 13] has shown that register allocation can be performed in two phases: a first spilling phase, with live-range splitting, that inserts loads, stores, and moves so that the resulting interference graph is greedy-k-colorable, and a second phase that coalesces moves and assigns colors to variables. In this context, register coalescing is crucial to reduce the cost of moves added blindly by live-range splitting. However, this is a pure coalescing problem, with no spill: starting from a greedy-k-colorable graph, how to coalesce moves so that it remains greedy-k-colorable? The same problem arises in the last iteration of graph coloring register allocators, i.e., when no spill is needed. In [2], Appel and George have adapted the de-coalescing phase of Park and Moon so that split nodes can always be colored and are never spilled. To ensure this, live-range splitting is done so that all nodes in the interference graph have degree $< k$, so this approach does not work for general greedy-k-colorable graphs.

In this paper, our goal is to address the following questions:

- Can we take advantage of the fact that the initial graph is greedy-k-colorable (no spill needed) to improve register coalescing?
- Can we derive more involved conservative tests?
- Can we avoid testing each move several times as in iterated register coalescing?
- Can we adapt Park and Moon optimistic coalescing to greedy-k-colorable graphs and improve the de-coalescing phase?
- Is incremental conservative coalescing (i.e., coalescing each move, one after the other, in a conservative way) really worse than optimistic coalescing (i.e., aggressive coalescing followed by de-coalescing)?

Section 2 recalls some necessary definitions and elementary properties linked to register coalescing. Section 3 presents a more involved conservative test to try to decide if the graph remains k-colorable after a given move is coalesced. This test is used in an incremental approach whose results are comparable traditional optimistic algorithm, although it is conservative. Section 4 shows how Park and Moon de-coalescing approach can be improved, leading to roughly 10% improvements on the collection of interference graphs provided by Appel (coalescing challenge [1]). Section 5 is an analysis of our results on these graphs, for different variants and trade-offs between running times and quality of results, comparing also with the optimal solutions provided in [12]. Section 6 concludes, discussing further possible improvements and open problems.

2 Background material

Before going further, we need to define more precisely some of the terms used in Section 1 and recall some important properties linked to greedy-k-colorable graphs. We refer to our previous theoretical paper [4] for the details and proofs not provided here.

The *interference graph* $G = (V, E)$ is an undirected graph where each node $v \in V$ corresponds to a live-range of the program. There is an *interference* $e = (u, v) \in E$ iff u and v cannot share the same register. In addition to interferences, each copy instruction (also called move) is represented by an *affinity* $a = (u, v)$. In general, an affinity has a weight $w(a)$, measure of a dynamic execution count of the corresponding copy instruction. A *coloring* of G is a function f of V such that $f(u) \neq f(v)$ whenever $(u, v) \in E$. When f takes at most k different values, it is a k-coloring. Given a set of affinities A, a coalescing is defined by a coloring f, with no constraint on the number of colors. An affinity $a = (u, v)$ is coalesced if $f(u) = f(v)$. The *coalesced graph* G_f is the graph obtained from G by merging all pairs of nodes linked by a coalesced affinity.

A graph G is k-colorable if it has a k-coloring. It is *chordal* [11] iff it has no chordless cycle of length ≥ 4. Chordal graphs are of particular interest for register allocation as the interference graph associated to variables of a strict SSA program is chordal [7, 3, 16, 13]. This is because live-ranges of SSA variables can be viewed as subtrees of the dominance tree. As mentioned in Section 1, a graph is greedy-k-colorable iff removing successively each node of degree $< k$ leads to the empty graph, i.e., if Procedure `Is_kGreedy` returns TRUE. Nodes can then be popped from the stack and colored if needed, picking for each node a color not used by its at most $(k-1)$ already-colored neighbors. An important property is that a k-colorable chordal graph is greedy-k-colorable [4, Property 1]. In other words, the simplify phase of graph coloring register allocators always succeeds to color a chordal graph with k colors if it is k-colorable.

```
Procedure Is_kGreedy(G)                    /* test for greedy-k-colorable graphs */
   Data: Undirected graph G = (V, E); ∀v ∈ V, degree[v] is set to degree of v in G
 1 stack = ∅ ; worklist = {v ∈ V | degree[v] < k} ;
 2 while worklist ≠ ∅ do
 3     Let v ∈ worklist ;
 4     foreach w neighbor of v do
 5         degree[w] ← degree[w]-1 ;
 6         if degree[w] = k − 1 then worklist ← worklist ∪ {w}
 7     push(v, stack) ; worklist ← worklist \ {v}        /* Remove v from G */
 8 if V = ∅ then return TRUE else return FALSE
```

In the next sections, we consider the following coalescing optimization problems as sub-problems of our coalescing strategies. They are stated here as unweighted but, in practice and in the next sections, affinities have weights that specify how often the corresponding moves are executed. One then seeks a coalescing of largest weight.

Problem: AGGRESSIVE COALESCING
Instance Graph $G = (V, E)$, affinities $A \subseteq V^2$, integer K.
Question Is there a coalescing of G, i.e., f with $f(u) \neq f(v)$ whenever $(u, v) \in E$, such that at most K affinities $(u, v) \in A$ are not coalesced, i.e., satisfy $f(u) \neq f(v)$?

Aggressive coalescing is NP-complete. A simple heuristic is to sort affinities by decreasing weights and to coalesce each affinity, one after the other, if no interference prevents it. However, picking the right set of affinities to coalesce is hard. Some heuristics for out-of-SSA conversion [18, 17] try to exploit the structure of ϕ-instructions

and consider several moves simultaneously, but out-of-SSA heuristics have never been integrated into a unified "coloring-coalescing" scheme. Also, no other sophisticated aggressive coalescing strategy exists, although there is space for improvements.

Problem: CONSERVATIVE COALESCING
Instance Graph $G = (V, E)$, affinities A, integers K and k.
Question Is there a coalescing f of G such that the coalesced graph G_f is k-colorable and at most K affinities are not coalesced?

Conservative coalescing is NP-complete. The same is true if one asks the graph G_f to be not only k-colorable, but also greedy-k-colorable. A traditional heuristic for conservative coalescing is to consider moves one after the other, in an incremental fashion. No heuristic exists that can consider several moves simultaneously. In an incremental approach, for each move, one has to consider the following problem.

Problem: INCREMENTAL CONSERVATIVE COALESCING
Instance Graph $G = (V, E)$, one given affinity $a = (x, y)$, an integer k.
Question Can we coalesce a to get a k-colorable graph, i.e., is there a k-coloring f of G such that $f(x) = f(y)$?

Incremental conservative coalescing is NP-complete for a general graph. However, it is polynomially-solvable if G is chordal. The proof of this result in [4] is conceptual. In Section 3, we give a linear-time algorithm for it, which we extend as a heuristic for a greedy-k-colorable graph. The complexity of incremental conservative coalescing for a greedy-k-colorable graph remains open. Note that asking G_f to be not only k-colorable but also greedy-k-colorable is easy as one just has to coalesce a then to run Procedure 1 to check that the resulting graph is greedy-k-colorable. However, this approach may still be too conservative. It may indeed happen that, to be able to coalesce the affinity a while keeping the graph greedy-k-colorable, other nodes need to be merged.

The last sub-problem to consider is the strategy used in optimistic coalescing: how to de-coalesce moves after a phase of aggressive coalescing so as to get a k-colorable or, more practical, a greedy-k-colorable graph. The problem is NP-complete even if the first phase of aggressive coalescing is trivial (i.e., when all moves can be coalesced).

Problem: DE-COALESCING FOR OPTIMISTIC COALESCING
Instance Graph $G = (V, E)$ greedy-k-colorable, affinities A all coalesceable (i.e., there is a coalescing f of G such that for all $(u, v) \in A$, $f(u) = f(v)$), integers k and K.
Question Is there a de-coalescing of G_f (i.e., a coalescing g of G such that $g(u) = g(v)$ implies $f(u) = f(v)$), such that at most K affinities (u, v) are not coalesced (i.e., satisfy $g(u) \neq g(v)$) and such that G_g is greedy-k-colorable?

In the next sections, we propose heuristics for these sub-problems and evaluate them using the collection of interference graphs obtained after the spill mechanism of [2]. Significant improvements can be obtained over previously-proposed heuristics.

3 Conservative coalescing, chordal graphs, and greedy-k graphs

So far, two existing conservative tests exist, the tests of Briggs [6] and of George [10]. Let us examine why they are conservative when applied on a greedy-k-colorable graph.
Briggs Merge u and v if the resulting node has at most $(k - 1)$ neighbors of degree $\geq k$.

George Merge u and v if all neighbors of u with degree $\geq k$ are also neighbors of v.
A node merged by Briggs' test can always be simplified from the graph once its neighbors of degree $< k$ are simplified, thus the graph remains greedy-k-colorable. Although George's test was defined only to merge a node v with a pre-colored node u, it can be applied for any two nodes for a greedy-k-colorable graph. Indeed, once all neighbors of degree $< k$ are simplified, a subgraph of the original graph remains, it is thus greedy-k-colorable too. These rules are good enough to coalesce graphs with low register pressure and few moves, but give poor results to coalesce the many moves introduced by a basic (without coalescing) out-of-SSA conversion, for example. The reasons are twofold.

First, both rules are local decisions, they depend on the degree of neighbors only. But these neighbors may have a high degree just because their neighbors are not simplified yet. In other words, the test may be applied too early in the simplify phase. This is the reason why George and Appel proposed "iterated register coalescing"[10]. Instead of giving up coalescing a move when the test fails, the move is placed in a sleeping list. It is woken up if the degree of one of the nodes implied in the rule, i.e., the neighbors of the nodes defining the move, changes. Thus, moves are in general tested several times. Also, move-related nodes should not be simplified too early for the moves to be tested.

Second, these two tests are used to coalesce moves in a sequential way. After each positive test, a unique merge occurs, resulting in a new greedy-k-colorable graph. This is a limitation. Indeed, to coalesce a set of moves, it may happen that coalescing all moves simultaneously leads to a greedy-k-colorable graph but, if moves are coalesced one by one, none of the intermediate graphs is greedy-k-colorable. An example of this situation is provided in [4]. In other words, aggressive coalescing followed by de-coalescing is, intrinsically, more likely to give better results than a conservative coalescing approach that coalesces one move at a time. We now try to overcome these two limitations.

3.1 Brute-force conservative coalescing

To address the first limitation, we evaluated the following approach. As in any incremental approach, we consider moves, one by one, in decreasing order of weights. But, to test a move, instead of relying on a local rule, we first merge the two corresponding nodes aggressively and check if the resulting graph is greedy-k-colorable (see Procedure `Brute_Force_Coalescing`). This amounts to perform a complete simplify phase, ignoring the other affinities. If the simplify succeeds, the coalesced graph is greedy-k-colorable and the coalescing was conservative. We then consider another affinity.

In Procedure `Brute_Force_Coalescing`, the simplify phase is conceptually performed on a copy G' of G, meaning that nodes are just temporarily simplified to check that G is greedy-k-colorable. They will be considered again when trying to coalesce the next affinity. However, nodes not involved in a move can be simplified for real before each new affinity is considered. In practice, there is no actual copy of the graph, nodes are just moved from the list of nodes in G to the simplify stack, and back to the list of nodes. Degrees can also be stored to avoid recomputing them.

Of course, the `Is_kGreedy` test is more costly than a local rule such as Briggs' and George's test, as its complexity is linear in the number of move-related moves. However, in iterated register coalescing, there is an overhead due to the fact that moves must be activated and evaluated several times. When the local test (Briggs' or George's) succeeds, the affinity is coalesced in the graph. When the test fails, the affinity is moved

from the main list to a sleeping list. In the background, each time the degree of a node becomes $< k$, the corresponding node is simplified. Also, each time the degree of a node decreases (thanks to a simplify or a coalescing), some sleeping affinities in its neighborhood (that might become coalesceable) are moved back to the main worklist. If this list is empty, a node is *frozen* (and hopefully simplified): all its affinities are removed from the graph (and the worklists). The process ends when the whole graph is simplified. A counting mechanism can be used to reduce the number of useless evaluations [14] but, still, the overall complexity of iterated register coalescing is *not* linear.

In our case, when `Is_kGreedy` returns FALSE, we have two possibilities, either we keep the move in some sleeping list for a possible re-evaluation, or we give up coalescing this move. The code of Procedure `Brute_Force_Coalescing` is the simple non-iterated version, where each move is evaluated only once and discarded if the test fails. Hence, only the main worklist is used, and there is no need for any freezing phase. As our experiments show, the `Is_kGreedy` test is much more powerful than local rules, thus testing each move only once does not degrade performance too much. On the contrary, keeping moves and reconsidering them each time the graph changes would be far too costly. In other words, we get an acceptable trade-off between execution time and performance, by spending more time in the test but avoiding the re-evaluation of moves. Other variants are possible, for example using local rules first to speed-up the process.

3.2 Chordal-based incremental coalescing

As mentioned before, coalescing moves one by one and requiring that, after each coalescing, the graph to be greedy-k-colorable is a limitation. Indeed, to coalesce a move and obtain a new greedy-k-colorable graph, it may be needed to coalesce other moves or even merge nodes not linked by an affinity. This fact was pointed out by Vegdahl [19] and also used by Park and Moon in their optimistic+ algorithm (extended optimistic coalescing) [15]. However, they perform these additional merges, in the hope of some benefit but with no guarantee that a coalescing is indeed enabled. Actually, for a given move, the corresponding optimization problem is as formulated in Section 2: Deciding if, after coalescing, the graph is k-colorable, i.e., greedy-k-colorable possibly with additional node merges. Such a test can coalesce moves beyond traditional conservative coalescing. As recalled in Section 2, it is polynomial for a chordal graph, NP-complete for a general graph. It is unfortunately still open for a greedy-k-colorable graph.

We first give the optimal algorithm for chordal graphs, which we extend as a heuristic for greedy-k-colorable graphs. We need a few preliminary lemmas.

Lemma 1. *A chordal graph with only two simplicial nodes is an interval graph. Furthermore, any perfect elimination scheme gives an interval representation.*

Proof. Let $G = (V, E)$ be a chordal graph with only two simplicial nodes u and v. (We recall that a node is simplicial if its neighbors form a clique.) A chordal graph has a perfect elimination scheme [11], i.e., a particular order of nodes $v_1, \ldots, v_n$, such that the neighbors of v_i in $G \setminus \{v_1, \ldots, v_{i-1}\}$ form a clique. Furthermore, there is such a scheme with $v_1 = u$ and $v_n = v$. Indeed, a chordal graph has always at least two simplicial nodes and a subgraph of a chordal graph is chordal. Therefore, we can first pick $v_1 = u$ and then always select a simplicial node $v_i \neq v$ in the perfect elimination scheme.

In this elimination scheme, for all $i > 1$, there exists $j < i$ such that $(v(i), v(j)) \in E$. Indeed, if this is not the case, the neighbors of $v(i)$ in $G \setminus \{v_1, \ldots, v_{i-1}\}$ are the neighbors

of $v(i)$ in G. Thus, $v(i)$ is simplicial in G, which is not possible unless $i = n$, as G has only two simplicial nodes, $v(1)$ and $v(n)$. For $i = n$, all neighbors $v(j)$ of $v(n)$ are of course such that $j < n$, unless $v(n)$ has no neighbor. But, in this case, $G \setminus \{v(n)\}$, which is chordal, has two other simplicial nodes, thus G has at least 3 simplicial nodes.

Each node is thus neighbor in G of a node eliminated (simplified) before. We can prove more: if $v(i)$ and $v(j)$, with $j < i$, are neighbors in G then $v(i)$ is also neighbor of $v(k)$ for all $j \leq k < i$. Indeed, suppose this is not the case. Let i be the smallest for which this property does not hold and let j be the largest such that $j + 1 < i$ with $v(j)$ a neighbor of $v(i)$ but $v(j+1)$ is not. Also, $v(j+1)$ is a neighbor of $v(j)$, otherwise i is not the smallest. This is impossible as the neighbors of $v(j)$ in $G \setminus \{v(1), \ldots, v(j-1)\}$ form a clique that contains both $v(j+1)$ and $v(i)$.

With the last property, we can view the nodes $v(i)$ as points on a line, drawn from left to right by increasing i, and G can be interpreted as the interference graph of n intervals. Each $v(i)$ corresponds to an interval that ends at $v(i)$ and starts at $v(j)$, for the smallest j such that $v(j)$ is neighbor of $v(i)$. Indeed, for all $j \leq k < i$, the interval corresponding to $v(k)$ intersects the interval corresponding to $v(i)$. We can prove more formally that G is an interval graph. A chordal graph is an interval graph if its complement is a comparability graph [11], i.e., if there is an order $\prec$ of nodes with the following property: for any three nodes x, y, z such that $(x, y) \notin E$, $(y, z) \notin E$, $x \prec y$ and $y \prec z$ imply that $x \prec z$ and $(x, z) \notin E$. This is true if $\prec$ is the perfect elimination scheme order as $(x, z) \in E$, $x \prec y \prec z$ would imply $(y, z) \in E$, i.e., y neighbor of z. ■

Lemma 2. *Let G be chordal and k-colorable, with nodes $v(1), \ldots, v(n)$. If for all i, $i \neq 1$, $i \neq n$, the degree of $v(i)$ in G is at least k and is minimum in $G \setminus \{v(1), \ldots, v(i-1)\}$, then $v(1), \ldots, v(n)$ define a perfect elimination scheme for G.*

Proof. As G is k-colorable, the size of any clique is at most k. Thus, only $v(1)$ and $v(n)$ can be simplicial for all other have degree $\geq k$. Furthermore, as G is chordal, Lemma 1 shows that it is actually an interval graph and any perfect elimination scheme that starts from $v(1)$ and ends at $v(n)$ gives a representation of intervals. We now show that $v(1), \ldots, v(n)$ define a perfect elimination scheme. If not, let i be the smallest such that $v(i)$ is not simplicial in $G \setminus \{v(1), \ldots, v(i-1)\}$. Thus, for all $k < i$, $v(k)$ is simplicial in $G \setminus \{v(1), \ldots, v(k-1)\}$. As $G \setminus \{v(1), \ldots, v(i-1)\}$ is chordal, one can complete $v(1), \ldots, v(n)$ into a perfect elimination scheme $w(1), \ldots, w(n)$, such that $w(n) = v(n)$ and, for all $k < i$, $w(k) = v(k)$. Following Lemma 1, this elimination scheme gives an interval representation, where the $w(k)$ are points on a line, drawn from left to right by increasing k, and where each $w(k)$ is the right end of an interval.

Consider the subgraph $H = G \setminus \{w(1), \ldots, w(i-1)\} = G \setminus \{v(1), \ldots, v(i-1)\}$. There are two cases, depending if $v(i)$ intersects $w(i)$ or not. If $v(i)$ is not a neighbor of $w(i)$, it cannot be a neighbor of any node already simplified because $w(1), \ldots, w(n)$ gives an interval representation. Therefore the degree of $v(i)$ in H is the degree of $v(i)$ in G, it is thus at least k, which is not minimum in H as the degree of $w(i)$ is at most $k - 1$. Impossible. If $v(i)$ is a neighbor of $w(i)$, any neighbor of $w(i)$ in H is a neighbor of $v(i)$ in H because $w(i)$ is simplicial in H. Therefore, the degree of $v(i)$ in H is larger than the degree of $w(i)$ in H and it cannot be minimum unless the degrees are equal. But, then, $v(i)$ has exactly the same neighbors as $w(i)$ in H and is simplicial. Impossible.

This proves that $v(1), \ldots, v(n)$ is a perfect elimination scheme. ■

Lemma 2 shows that, if G is a k-colorable chordal graph and (u, v) is an affinity to be coalesced, one can first simplify all possible nodes to end up with a subgraph G' where all nodes have degree $\geq k$, except u and v. G' is then an interval graph. The problem of coalescing (u, v) in a chordal graph is therefore reduced to the problem of coalescing (u, v) in an interval graph. Furthermore, one can get a representation of G' with intervals by a simple simplify phase: first simplify u, then simplify all other nodes, always selecting a node with minimum degree, other than v. In this particular case, there is no need to use a sophisticated algorithm for general interval graphs. Procedure `Chordal_Coalescing` uses the conceptual idea developed in [4, Theorem 5] for chordal graphs, associated with the algorithmic ideas of Lemmas 1 and 2.

Theorem 1. *Let G be a k-colorable graph and $a = (x, y)$ an affinity. If G is chordal then Procedure* `Chordal_Coalescing` *returns* TRUE *if and only if merging x and y leads to a k-colorable graph. If G is only greedy-k-colorable, the test is safe, but not exact. Furthermore, after all nodes in the path built by the algorithm are merged, the graph gets greedy-k-colorable again.*

Proof. If G is chordal, Lines 3-7 first simplify nodes so as to get an interval graph. Then, it is easy to understand intuitively that G has a coloring such that x and y have the same color if and only if there is sequence of (possibly dummy) intervals, from x to y, such that each interval ends just before the next one starts (see [4, Theorem 5]). Thus, it is sufficient to propagate a flag, starting from x, from ends of intervals to starts of intervals and check if y is reached. The computations of the variables like_x[w] and dummy_like_x in the repeat loop, Lines 13-22, do this propagation. The full optimality proof for a chordal graph is a bit technical. Its extension to greedy-k-colorable graphs consists intuitively in considering the graph as a subgraph of a larger chordal graph. All details are provided in the appendix. ■

Theorem 1 is the basis for many heuristic variants. The idea is, starting from a greedy-k-colorable graph, to find strategies to add interferences and/or merge nodes so that, after coalescing one particular affinity, the graph is still greedy-k-colorable. It is indeed very important to stay in the class of greedy-k-colorable graphs, as such graphs can be easily colored and previously-proposed heuristics, such as the tests of Briggs and George, can be used. Here, we proposed a technique based on chordal graphs, but variants are certainly possible. The best would be to find an optimal test for greedy-k-colorable graphs, unless this problem is shown NP-complete.

4 Aggressive coalescing + de-coalescing and greedy-k graphs

Merging two nodes can transform a non greedy-k-colorable graph into a greedy-k-colorable graph. This observation is the motivation of aggressive coalescing. It is indeed maybe more beneficial to first coalesce aggressively as many affinities as possible, then to try to undo (de-coalesce) some coalescings if the graph is not greedy-k-colorable.

But how to de-coalesce? After aggressive coalescing, Park and Moon [15] proceeds with the standard simplify/select phases. In the select phase, if the result of a merge is a potential spill, it is split into the original nodes. It is not safe to color all of them right now, as this would constrain too much the nodes simplified before, which are not colored yet. However, if some are colored with a unique color and the other discarded, the rest of the select phase can continue safely. Park and Moon thus choose heuristically which nodes to color and the other are colored or spilled after the select phase.

Procedure `Brute_Force_Coalescing`(G, A) /* coalesce if greedy-k-colorable */

Data: Graph $G = (V, E)$, affinities A, weight function $w : A \to \mathbb{N}$
while $A \neq \emptyset$ **do**
  $a = (x, y) \leftarrow$ get_max(A, w) ; $A \leftarrow A \setminus \{a\}$; /* *affinity with biggest weight* */
  if $(x, y) \notin E$ **then**
    $G' \leftarrow G$; merge x and y in G' ;
    if `Is_kGreedy`(G') = TRUE **then** merge x and y in G

Procedure `Chordal_Coalescing`(G, a)

Data: Graph $G = (V, E)$, $a = (x, y)$ affinity
Result: TRUE if a can be coalesced conservatively, possibly with additional node merges
if `Briggs_George_Coalescing`(G, a) **then** merge x and y in G; **return** TRUE ;
merge x and y into xy in G ;
nodes = V ; stack = $\emptyset$;
forall $v \in V$ **do** set degree[v] to the degree of v in G ;
while degree[xy] $\geq k$ and $\exists v \in$ nodes, degree[v] $< k$ **do**
  foreach w neighbor of v **do** degree[w] $\leftarrow$ degree[w]-1 ;
  push(v, stack) ; nodes $\leftarrow$ nodes $\setminus \{v\}$; /* *Simplify* */
if degree[xy] $< k$ **then return** TRUE ;
split back x and y in G ; replace xy in *nodes* by x and y ;
if degree[x] $\geq k$ **then** switch x and y ;
/* *Traverse the set of intervals from x to y* */
like_x[x] $\leftarrow$ TRUE; dummy_like_x $\leftarrow$ FALSE; alive $\leftarrow \{x\}$;
just_removed $\leftarrow \emptyset$; like_x[$\emptyset$] $\leftarrow$ FALSE;
repeat
  $v \leftarrow$ node in nodes$\setminus\{y\}$ with smallest degree ;
  dummy_like_x $\leftarrow$ dummy_like_x or like_x[just_removed] ;
  foreach w neighbor of v **do**
    degree[w] $\leftarrow$ degree[w]-1 ;
    if $w \notin$ alive **then** alive $\leftarrow$ alive $\cup \{w\}$; like_x[w] $\leftarrow$ dummy_like_x ;
  if #alive = k **then** dummy_like_x $\leftarrow$ FALSE;
  if #alive > k **then return** FALSE;
  push(v, stack) ; nodes $\leftarrow$ nodes $\setminus \{v\}$; alive $\leftarrow$ alive $\setminus \{v\}$; just_removed $\leftarrow v$;
until nodes = $\{y\}$;
if like_x[y] = FALSE **then return** FALSE;
/* *Else, construct the path linking x and y* */
path $\leftarrow \{y\}$; current $\leftarrow y$;
while v = pop(stack) $\neq x$ **do**
  if v not a neighbor of current **then**
    if like_x[v] **then** path $\leftarrow$ path $\cup \{v\}$; current $\leftarrow v$;
merge all $v \in$ path into a single node in G ;
return TRUE

Appel and George [2] pointed out that, if all nodes have degree $< k$ in the original graph, they can always be colored if they are not merged. Based on this observation, they modified Park and Moon de-coalescing to ensure that no spill occurs. Their exact mechanism is not clear but, from [2], one can guess that either they color some nodes at the very end, as Park and Moon, but with no spill, or they color them right away, even if this can create a cascade of other splits, even nodes that are not potential spill. How to adapt this heuristics to greedy-k-colorable graphs is not clear. Their results, for a large collection of graphs, are provided in the coalescing challenge web page [1].

The weaknesses of these approaches is that, instead of de-coalescing moves, possibly one by one, nodes are split back and many moves are de-coalesced, even if not needed. Also, as this process is done in the select phase, it is not clear how to combine it with, for example, conservative coalescing. Thus, we prefer to perform de-coalescing based on the graph structure itself, and not the particular order of nodes on the stack.

We proceed as in Procedure `De-coalescing`. As long as the graph is not greedy-k-colorable, we de-coalesce some affinities. We select these affinities as follows: during the check for greedy-k-colorability (Procedure `Is_kGreedy`), if we end up with a subgraph in which all nodes have degree $\geq k$, we select the cheapest affinity in this subgraph and de-coalesce it. We continue the simplify phase until the graph becomes empty. This way, first we avoid de-coalescing an affinity in the area where it is not needed (the nodes already simplified), second we give up the cheapest moves first. However, as for Park and Moon approach, de-coalescing a move can increase the degree of nodes already simplified. Therefore, in general, we need to perform several passes (in practice 3 maximum). Also, one can stop the de-coalescing even before, taking benefit of biased coloring, if we want the graph to be just k-colorable (see the call to `find_coloration` in the code). Other possible variants are discussed in Section 5.

5 Experiments and evaluation

As outlined in the introduction, the coalescing problem is challenging for graphs with many affinities and high register pressure. In particular, a graph based spill everywhere algorithm will lead to a too simple coalescing problem. The coalescing challenge [1] provides an interesting collection of graphs: existing state of the art coalescing algorithms have already been implemented. In particular, the low number of registers (6) allowed Grund and Hack to provide optimal solutions [12] for all but three graphs.

In their spill algorithm, Appel and George [2] performed live-range splitting at every program point. Hence the corresponding control flow graphs can be easily rebuilt. There are 474 graphs that correspond to regions (procedures?) of the Standard ML of New Jersey benchmark suite, compiled for a Pentium with 6 general-purpose registers. On average, there are ≈ 26.7 basic blocks per region, with a maximum of ≈ 1090, and ≈ 231 instructions, with a maximum of ≈ 8300. Notice also that the architecture has many instructions with register constraints, which constrains the graph coloring (without necessarily simplifying it). This collection of graphs is thus interesting and representative. However, we should notice that, even if the coalescing problem is already hard with few registers [4], here 6, it becomes much harder with more registers.

Clearly the order in which affinities with identical weight are considered for coalescing is crucial. It seems reasonable to guide the ordering using both the knowledge of the program structure, in addition to graph properties (such as the degree of nodes).

To provide reproducible results, we chose a deterministic ordering, the one given by the graph description file. In fact, this ordering is not arbitrary, since it exactly follows the control flow graph. Notice that using another ordering does *not* change the overall *relative* comparisons of the different schemes. Also, to compare with Appel and George's version of optimistic algorithm, we decided to compare to the results provided in [1] and not implement it ourselves as its description in [2] misses too many details.

From now on, `BG` refers to the rules of Briggs/George, `Chordal` refers to the heuristic with chordal-based coalescing only, *k*-`Greedy` to the heuristic with brute-force coalescing only, `Aggr` is the aggressive coalescing plus de-coalescing, and our preferred scheme, called `Pref.`, is `Aggr`+`BG`&`Chordal` (by packet). Table 1 compares the different conservative tests `BG`, *k*-`Greedy`, and `Chordal`. For this, we considered the following general scheme that contains four actions: `Simplify` (each time their exists a non move-related node of degree $< k$); `BG` (when $worklistBG \neq \emptyset$), `Chordal` (when $worklistCh \neq \emptyset$ and $worklistBG = \emptyset$); `Freeze` (when both worklists are empty). *k*-`Greedy` consists in the first 8 lines of `Chordal`. To run `BG` (resp. `Chordal`) alone, `Chordal` (resp. `BG`) is disabled. Concerning the sleeping strategy, if `BG` fails, the affinity is put in a sleeping worklist, here *worklistCh*; if `Chordal` fails, the affinity is discarded immediately in the normal mode and moved to a sleeping worklist in the *sleeping* mode. It might be woken up to *worklistBG*, pessimistically, if the degree of a neighbor decreases. Finally, in the "packet" version, affinities of equal weight are grouped and all worklists (for `BG` and `Chordal`) must be empty before processing the next packet. This allows `BG` to speed-up `Chordal` while avoiding to coalesce light-weight affinities too early, and gives results comparable to `Chordal` alone. Finally, we have either performed the conservative coalescing as a post-pass phase of an `Aggr` phase, or alone.

For each graph, the cost of the coalescing is the sum of the weights of the affinities not coalesced. Table 1 gives the cost ratios optimal/heuristic. The average value is obtained by weighting each graph with its number of instructions. With this metric, `Aggr` without any conservative post-pass is valued at 0.667, Appel and George's optimistic algorithm at 0.737. If affinities are ordered by decreasing number of neighbors when their weights are equal, `Aggr`+`BG` is valued at 0.835! The table shows that i) the best purely conservative approach is very competitive at 0.810; ii) `Chordal` improves only slightly the *k*-`Greedy` test: our test only exposes interval graphs which seem to be too constrained. Improving this test seems possible, especially since the size of the graphs (after the initial simplification of `Chordal_Coalescing`) have <40 nodes in practice; iii) `BG` performs poorly compared to a pure *k*-`Greedy` approach and degrades the result when associated to a chordal based (`BG` & `Chordal`), unless it is used by packet.

We now give some very rough running time comparisons. Please note that our implementations are not yet tuned for speed, so this may bias the results. In particular, there is two highly time-consuming facts: i) our search for maximum weight affinities takes $O(\#\text{list})$: this is bad for `BG` which frequently makes such requests; ii) at each step, `Chordal` always re-simplifies the full graph while non move-related nodes should be simplified once and for all; iii) Line 1 of `Chordal_Coalescing` is not used.

Time comparison of the different heuristics. The graphs vary a lot in size, and so does the execution time. In Table 2, we consider five groups of graph for each `BG`, `Chordal` and `Pref.` heuristics: the graphs that take <1s to be computed on a 2.6GHz processor,

Procedure `Aggressive_Coalescing`(G, A, w)

Data: Graph $G = (V, E)$, affinities A, weight function $w : A \to \mathbb{N}$

while $A \neq \emptyset$ **do**
  $a = (x, y) \leftarrow$ get_max(A, w) ; /* *affinity with biggest weight* */
  $A \leftarrow A \setminus \{a\}$;
  if $(x, y) \notin E$ **then** merge x and y in G

Procedure `De-coalescing`(G, A, w)

Data: Graph $G = (V, E)$, affinities A, weight function $w : A \to \mathbb{N}$

repeat
  nodes $\leftarrow V$; stack $= \emptyset$;
  while *nodes* $\neq \emptyset$ **do**
    if $\exists v \in$ nodes, degree$[v] < k$ **then**
      nodes $\leftarrow$ nodes $\setminus \{v\}$; push(v, stack) ; /* *Simplify v* */
      foreach w neighbor of v **do** degree$[w] \leftarrow$ degree$[w]$-1 ;
    else
      $a \leftarrow$ get_min_affinity(nodes, A) /* *coalesced affinity for a non-simplified node, with smallest weight* */
      $G \leftarrow$ de-coalesce(G, a) ; /* *de-coalesce the affinity* */
until find_coloration(stack, G) ;

no `Aggr`	w/ `Aggr`	heuristic
0.704	0.747	`BG`
0.800	0.790	*k*-`Greedy`
0.810	0.795	`Chordal`
0.772	0.776	`BG` & `Chordal`
0.769	0.800	Idem by packet
0.808	0.789	*k*-`Greedy` (w/ *sleeping*)
0.810	0.795	`Chordal` (w/ *sleeping*)
	0.737	optimistic
	0.835	`Aggr`+`BG` w/ ≠order

Table 1. Comparisons vs optimal.

Execution time	`BG`	`Chordal`	`Pref.`
t < 1s	432	313	428
1s < t < 10s	27	125	32
10s < t < 1min	8	21	7
1min < t < 10min	4	10	6
10min < t	3	5	1
Total	≈4h	≈10h	≈2h

Table 2. #graphs per processing time.

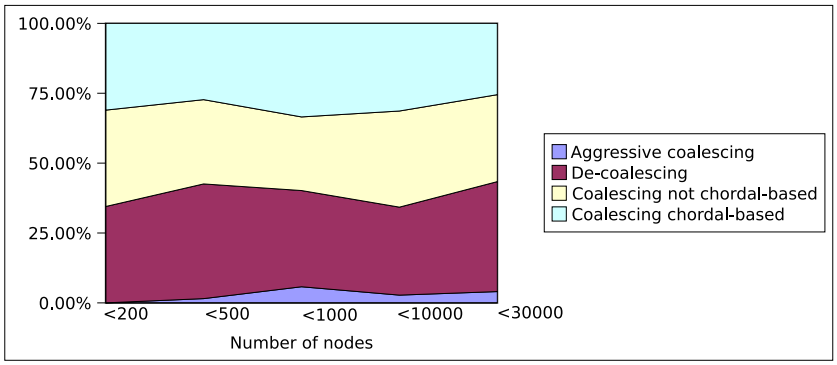

Fig. 1. Time used by the different phases of `Pref.`

those between 1s and 10s, and so on. The table gives the number of graphs in each category. First, `Chordal` is much slower than `Pref.`. We measured that it is more than 10x times slower on average on graphs with <1000 nodes, and more than 5x slower on larger graphs. Second, `Pref.` runs even faster than the simpler `BG` heuristic: a bit faster on smaller graphs, >0.3x faster on graphs between 1k and 10k nodes, and sometimes 2x or 3x faster on the biggest graphs of ≈20k. This even holds when deactivating the initial `Aggr` approach: on medium to big graphs, `BG` takes more time testing a particular move, over and over, than the time needed by `Chordal` to discard it.

Inside the `Pref.` heuristic. Figure 1 gives the time spent in each phase of the `Pref.` heuristic in groups of graphs with similar number of nodes. The ratios are quite stable: roughly 40% of the time is spent in the `Aggr` part, the remaining time in conservative coalescing. The aggressive part is a lot faster than the de-coalescing part, as expected, since the de-coalescing phase may need to iterate and includes a simplify process. In practice, we measured that there are 1.6 passes of de-coalescing on average, and at most 3 passes are required: this concerns about 6% of the graphs, but none of the biggest ones. A similar time is spent in `Chordal` and for the rest of the conservative coalescing (checking `BG` rules, maintaining lists of nodes or affinities, simplifying). This may seem a lot, as it performs only about 8% of the conservative coalescing, but it is worth the time since it discards affinities that `BG` will not need to check again.

Which rule coalesces? On all the 474 graphs of the coalescing challenge, the `Pref.` scheme de-coalesced 92896 affinities in 747 passes. Out of these and the other ones that the aggressive did not coalesce, 90312 were conservatively coalesced. `BG` coalesced itself 83179, i.e., 92% of them. `Chordal` tried 10232 affinities, successfully coalescing 7133, the remaining 8%. 97% of these 8% can be found by `Brute_Force_Coalescing` (6893), so only 3% (240) actually used the "path-searching"part of `Chordal`. However, 3099 affinities were promptly discarded because there was no path (33%) or the `alive` set grew bigger than k (67%): for 3/4 of those, no node with degree $< k$ could be removed. The whole test took about 1 hour and 55 minutes and no animals were harmed.

6 Conclusion

It is a common belief that optimistic coalescing outperforms conservative coalescing. Based on previous theoretical work [4], we developed a more involved incremental conservative strategy than the well-known iterated register coalescing. Our heuristic outperforms the optimistic approach by about 5%. Our conservative tests (brute force or chordal-based) while being much more costly than a simple Briggs/George test, have a competitive amortized cost: the iterated nature of iterated register coalescing leads to testing many times the same affinity while our heuristic tests each affinity, only once, to coalesce or discard it. We also developed a more aggressive optimistic approach that works for any greedy-k-colorable graph (as opposed to Appel and George's optimistic algorithm) and ensures it is still greedy-k-colorable after coalescing. The two algorithms can be combined to get about 10% improvement over traditional optimistic coalescing.

While playing with the ordering in which affinities with equal weights are considered, we measured noticeable differences of the quality of the final result. In particular,

we verifi ed that guiding this ordering, using both the program structure and simple considerations such as node's degree, provides better results in average. We believe this point is worth exploring for both the aggressive and conservative coalescing problems.

Finally, not surprisingly, our incremental conservative test for greedy-k-colorable graphs provides poor improvements over the brute force test: it is a too direct adaptation of the optimal test for chordal graphs. Improvements are possible, especially since we measured that the resulting graphs, after the initial simplifi cation, have only < 40 nodes.

Acknowledgments

Many thanks to C. Guillon and B. Dupont de Dinechin for their help and support, and S. Hack and D. Grund for their optimal ILP-based results and for fruitful discussions.

References

1. A. Appel and L. George. Coalescing challenge. `http://www.cs.princeton.edu/~appel/coalesce`, 2000.
2. A. W. Appel and L. George. Optimal spilling for CISC machines with few registers. In *Proc. of the ACM SIGPLAN Conference PLDI'01*, pages 243–253, Utah, USA, 2001. ACM Press.
3. F. Bouchez, A. Darte, C. Guillon, and F. Rastello. Register allocation and spill complexity under SSA. Technical Report RR2005-33, LIP, ENS-Lyon, France, Aug. 2005.
4. F. Bouchez, A. Darte, and F. Rastello. On the complexity of register coalescing. In *Int. Symp. on Code Generation and Optimization (CGO'07)*, pages 102–114. IEEE Comp. Press, 2007.
5. P. Briggs. Register allocation via graph coloring. PhD Thesis Rice COMP TR92-183, Department of Computer Science, Rice University, 1992.
6. P. Briggs, K. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.
7. P. Brisk, F. Dabiri, J. Macbeth, and M. Sarrafzadeh. Polynomial time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*, June 2005.
8. G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.
9. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.
10. L. George and A. W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.
11. M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.
12. D. Grund and S. Hack. A fast cutting-plane algorithm for optimal coalescing. In *Compiler Construction 2007*, volume 4420 of *LNCS*, pages 111–125. Springer, 2007. Braga, Portugal.
13. S. Hack, D. Grund, and G. Goos. Register allocation for programs in SSA-form. In *Compiler Construction 2006*, volume 3923 of *LNCS*. Springer Verlag, 2006.
14. A. Leung and L. George. A new mlrisc register allocator. Technical report, New York University, 1999.
15. J. Park and S.-M. Moon. Optimistic register coalescing. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 26(4), 2004.
16. F. M. Q. Pereira and J. Palsberg. Register allocation via coloring of chordal graphs. In *Proc. of APLAS'05, Asian Symp. on Progr. Languages and Systems*, pages 315–329, Japan, 2005.

17. F. Rastello, F. de Ferrière, and C. Guillon. Optimizing translation out of SSA using renaming constraints. In *Proc. of the Intern. Symp. CGO'04*, pages 265–278. IEEE Comp. Soc., 2004.
18. V. C. Sreedhar, R. D. Ju, D. M. Gillies, and V. Santhanam. Translating out of static single assignment form. In *SAS'99*, volume 1694 of *LNCS*, pages 194–210. Springer Verlag, 1999.
19. S. R. Vegdahl. Using node merging to enhance graph coloring. In *Proc. of the ACM SIGPLAN Conf. PLDI'99*, pages 150–154, New York, NY, USA, 1999. ACM Press.

Appendix

A Complete proof of Theorem 1

Proof. Lines 1, 2, 8, 9, 10, and 20 are not needed for the correctness of the algorithm if G is chordal. They are here just to speed-up the process and for the extension to greedy-k-colorable graphs. Lines 1 uses Briggs' and George's tests to decide more quickly. The test `Is_kGreedy` used in `Brute_Force_Coalescing` and Lines 2-8 are similar, except that, here, the simplify phase ends possibly earlier in case the merged node xy itself can be simplified. If not, the algorithm for chordal graphs begins at Line 11. All nodes with degree $< k$, except x and y, have been simplified, and even the common neighbors of x and y with degree $\leq k$, as those will be simplifiable if x and y are finally merged. Let us concentrate on the decision part, from Line 11 to Line 23.

According to Lemma 2, the remaining graph is an interval graph and an interval representation is obtained by simplifying x first, then each node $\neq y$ with minimum degree (Line 14). In the repeat loop, Lines 13-22, at the i-th iteration, v (set at Line 14) is the node $v(i)$ in Lemma 2, just_removed is $v(i-1)$ before being updated at Line 21 for the next iteration, and alive (updated at Line 18) contains all intervals live at point $v(i)$. Indeed, alive is completed by all neighbors w of $v(i)$ not already in alive. The corresponding intervals thus start just after $v(i-1)$ ends. Also, if at most $(k-1)$ intervals are live at point $v(i)$, one considers that a dummy interval is live at this point. If it does not exist, dummy_like_x is FALSE (Line 19). This situation is depicted in Figure 2.

It is easy to see that G has a coloring such that x and y have the same color if and only if there is sequence of intervals, possibly dummy, such that each interval ends just before the next one starts, and the first interval is x, the last is y. Therefore, it is sufficient to propagate a flag, starting from x, from ends of intervals to starts of intervals and see if y can be reached. This is what the propagation of the variables like_x[w] and dummy_like_x does. If there is a coloring in which x and y have the same color, then the propagation from like_x[x] = TRUE reaches y. Let us see the converse more formally.

Let us prove that if like_x[w] is set to TRUE at iteration i (Line 18), the following holds: if there is a k-coloring of w and of all nodes simplified *later* (thus colored *before* in the select phase) and such that w and y have the same color, then one can build a coloring of G such that y and x have the same color. A similar property holds for the dummy interval at $v(i)$ as like_x[w] and dummy_like_x have the same value. Let us prove

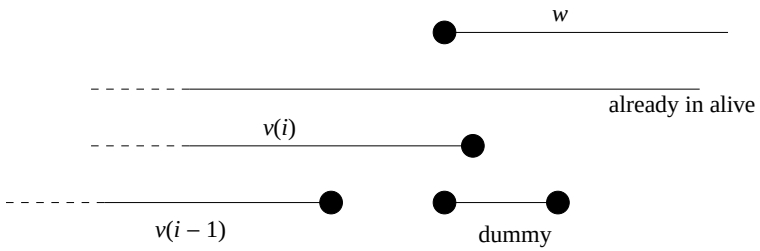

Fig. 2. Position of relative intervals at iteration i.

this property by induction on i, the implicit loop counter of the repeat loop. This is true for $x = v(1)$ due to the initialization at Lines 11-12. Assume the property is true for all $j < i$. Let w a node added in alive at iteration i and such that like_x[w] = TRUE (the same argument can be used for the dummy interval at $v(i)$). Consider a coloring of all nodes simplified after w such that w and y have the same color. As G is greedy-k-colorable, this coloring can be extended, popping nodes from the stack, to all nodes from w down to $v(i)$, without using the color of w because all these nodes interfere with w. If like_x[w] is TRUE because like_x[$v(i-1)$] (resp. dummy_like_x) is TRUE, one can color $v(i-1)$ (resp. the dummy interval at $v(i-1)$) with the color of w. Finally, by induction hypothesis at iteration $i-1$, we can extend the coloring to G such that x and y have the same color.

It remains to prove that, with additional node merges, we can obtain a greedy-k-colorable graph. This is done by following a path from y to x when popping nodes from the stack (Lines 24-27). Merging all these nodes amounts to form a complete interval from x to y, thus transforming the graph into another interval graph, with at most k live intervals, thus greedy-k-colorable. The only subtlety is that we should also merge some dummy intervals, which means we should add interference edges with some other intervals alive at these points. But, these interferences already exist after the merging of the path, otherwise, one of these intervals is fully included in a portion covered by dummy intervals colorable with the same color as x and y, thus it would have added in the path at Line 27, instead of a later node.

If G is only greedy-k-colorable, but not necessarily chordal, nodes are simplified in some order $v(1), \ldots, v(n)$, and the repeat loop behaves as if G was an interval graph G' where each interval goes from $v(j)$ if it is added in the set alive at iteration j to $v(i)$ if it is simplified at iteration i. This amounts to assume that all nodes in the set alive form a clique, even if they do not interfere, but there is no need to add these edges explicitly. The result of Procedure `Chordal_Coalescing` is then given with respect to this interval graph G', for which G is a subgraph. It is possible that G' is not k-colorable (Line 20) but if Procedure `Chordal_Coalescing` returns TRUE, it is safe to merge x and y. The additional merging of the nodes on the path then transforms G' into another interval graph and G into a subgraph of it, it is therefore greedy-k-colorable. Taking into account all nodes previously simplified (Lines 3-7), the graph is still greedy-k-colorable. ■