Advanced Conservative and Optimistic Register Coalescing[†]

Florent Bouchez   Alain Darte   Fabrice Rastello[*]
Compsys team, LIP
UMR 5668 CNRS—ENS Lyon—UCB Lyon—Inria, France
Firstname.Lastname@ens-lyon.fr

ABSTRACT

In the context of embedded systems, it is crucial to minimize memory transfers to reduce latency and power consumption. Stack access due to variable spilling during register allocation can be significantly reduced using aggressive live-range splitting. However, without a good (conservative) coalescing technique, the benefit of a good spilling may be lost due to the many register-to-register moves introduced. The challenge is thus to find a good trade-off between a too aggressive strategy that could make the interference graph uncolorable without more spilling, and a too conservative strategy that preserves colorability but leaves unnecessary moves. Two main approaches are "iterated register coalescing" by George and Appel and "optimistic coalescing" by Park and Moon. The first coalesces moves one by one conservatively. The second coalesces moves regardless of the colorability, then undo coalescings to reduce spilling. Focusing on greedy-*k*-colorable graphs—which are usually obtained after all spill decisions and, possibly, some split decisions—we show how these two approaches can be improved, optimistic coalescing with a more involved de-coalescing phase, incremental coalescing with a less pessimistic conservative technique. Unlike previous experiments, our results show that optimistic strategies do not outperform conservative ones. Our incremental conservative coalescing performs even better than our improved de-coalescing scheme and leads to about 15% improvement compared to the state-of-the-art optimistic coalescing.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers, Optimization*; F.2.0 [**Analysis of Algorithms and Problem Complexity**]

General Terms

Algorithms, Experimentation, Performance, Theory

[†]This work is supported by a contract with the compilation team of STMicroelectronics, Grenoble.

[*]F. Bouchez is PhD student at ENS-Lyon, A. Darte is a CNRS researcher, and F. Rastello is an Inria researcher.

Keywords

Register coalescing, Register allocation, Chordal graph, Coloring number, Greedy-*k*-colorable graph

1. INTRODUCTION

High-level source codes contain few copy (move) instructions, but this is not the case after the many optimization phases that occur in a compiler. For example, static single assignment (SSA) [11] introduces virtual "ϕ-functions" at join 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 either carefully removed [20, 19], or eliminated by a later register allocation phase. Another example is live-range splitting, which is mandatory to improve spilling (transfers between registers and memory) and thus to reduce latency and power consumption, two important issues in embedded systems. The extreme situation is when live-ranges are split at each program point so as to formulate the spilling problem as an integer linear program [3]. Loads/stores are then nicely optimized but many copies are created that need to be removed. Finally, register moves can also be added to handle, in a simple way, register constraints or calling conventions. A later phase is supposed to remove these additional moves. Thus, there are many reasons to try to get rid of copy instructions at assembly-code level. Register coalescing does this during register allocation and thus is tightly connected to spill optimization and register assignment.

Chaitin et al. popularized graph coloring register allocation [10]. Each node of the interference graph G corresponds to a program variable and an edge between two nodes means that they interfere, i.e., they 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 to 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 $G\backslash\{x\}$ 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 said to be *greedy-k-colorable* [6]. It is indeed k-colorable in a greedy way since each node can be popped from the stack and colored with one of the colors not used by its $< k$ previously-popped neighbors. This second phase is the *select phase.* Spilling and coalescing are always optimized with this simplify rule in mind, as it is the only simple strategy to color general graphs.

Spilling is usually done with Briggs et al. mechanism [7, 8]. During the simplify phase, if all nodes have at least k neighbors, one is removed from the interference graph, pushed on the stack, and marked as a *potential spill*. During the select phase, if 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" while coloring during the select phase (i.e., when the select phase succeeds but the graph is not greedy-k-colorable). 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 effect is that many moves are indeed deleted but the number of potential and actual spills almost always increases. To address this issue, two main approaches were explored, *conservative* coalescing and *optimistic* coalescing.

Conservative coalescing consists in coalescing a move only when one can ensure that, if the initial graph was k-colorable, the resulting graph is still k-colorable. In Briggs's test [7, 8], a move is coalesced only if the resulting node has $< k$ neighbors with degree $\geq k$. Initially, this test was used to reduce the number of moves introduced by live-range splitting whereas original moves were treated by aggressive coalescing. George and Appel then proposed *iterated register coalescing* [12], 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 (those used to represent dedicated machine registers): a node can be merged with a pre-colored node if its neighbors of degree $\geq k$ are all neighbors of the pre-colored node. As these two tests are fast but not exact (the test can fail even if the move can be conservatively coalesced), better results are obtained if moves are tested several times while simplifying nodes. For that, worklists of potentially-coalescable moves are created and updated during the simplify phase. This increases the running time of the complete register allocator, even if the smart implementation strategies defined by Leung and George [16] can reduce this overhead.

Iterated register coalescing is now very popular due to its clean and reproducible design, although, in terms of moves optimization, optimistic coalescing proposed by Park and Moon [17] 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, which is a positive effect. Thus, it seems better to perform aggressive coalescing first and then to decide, during the select phase, to undo some coalescings 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 if possible, the others are either colored at the very end of the select phase or spilled if no colors are available for them. Optimistic coalescing takes its benefit from the aggressive phase and, despite its naive de-coalescing phase, outperforms iterated register coalescing.

Recent work [3, 9, 4, 15] 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. But unlike the previously-mentioned approaches, it is a pure coalescing problem, with no additional spill: how to coalesce moves in a greedy-k-colorable graph so that it remains greedy-k-colorable. The same problem arises in the last iteration of graph coloring register allocators, i.e., when no potential spill is needed. Appel and George have adapted [3] the de-coalescing phase of Park and Moon so that split nodes can always be colored and are never spilled: their live-range splitting is done so that all nodes in the interference graph have degree $< k$. Hence, 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 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 (moves are coalesced, one by one, while keeping the graph k-colorable) really worse than optimistic coalescing (moves are coalesced aggressively then possibly de-coalesced to get a k-colorable graph)?

We develop advanced conservative and optimistic coalescing algorithms that allow us to give affirmative answers to the first four questions. Then, the evaluation of these new strategies let us think that the answer to the last question might well be *no*, for greedy-k-colorable graphs. Section 2 recalls some necessary definitions and elementary properties linked to register coalescing. Section 3 presents more involved conservative tests to decide if the graph remains k-colorable after a given move is coalesced. These tests are used in an incremental approach whose results outperform, by roughly 15%, state-of-the-art optimistic algorithms, even though it is conservative! In Section 4, we improve optimistic coalescing by developing advanced de-coalescing mechanisms. Section 5 is an analysis of our results on the collection of interference graphs provided by Appel (coalescing challenge [2]), which is now considered as the benchmark suite for evaluating coalescing algorithms. We evaluate variants and trade-offs between running times and quality of results, comparing also with optimal solutions provided by Grund and Hack [14]. Section 6 concludes, discussing possible improvements and open problems.

2. BACKGROUND MATERIAL

We first need to define more precisely some of the terms used in Section 1 and to recall some properties linked to greedy-k-colorable graphs. For details and proofs, we refer to our previous paper [6].

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 $w(a)$ that gives an evaluation of how often the corresponding copy instruction would be executed. A *coloring* of G is a function $f : V \to \mathbb{N}$ such that $f(u) \neq f(v)$ whenever $(u, v) \in E$. When $f(V)$ contains 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 obtained from G by merging any two nodes linked by a coalesced affinity.

A graph G is k-colorable if it has a k-coloring. It is *chordal* [13] iff any cycle of length ≥ 4 has a chord (edge between two nodes not consecutive in the cycle). Chordal graphs are of particular interest for register allocation as the interference graph of variables in a strict SSA program is chordal [4, 9, 15, 18]. This is because live-ranges of SSA variables can be viewed as subtrees of the dominance tree. As defined in Section 1, a graph G is *greedy-k-colorable* iff removing successively all nodes of degree $< k$ leads to the empty graph, i.e., if Function `Is_kGreedy`(G) (see next page) returns TRUE. Nodes can then be popped from the stack and colored if needed, picking for each node a color not used by its $< k$ already-colored neighbors. Note that *a k-colorable chordal graph is greedy-k-colorable* [6, Property 1]. In other words, the simplify/select phases of graph coloring register allocators always succeed to color a chordal graph with k colors if it is k-colorable.

```
Function Is_kGreedy(G)                    /* test of greedy-k-colorability */
  Data: Undirected graph $G = (V, E)$; $\forall v \in V$, degree[$v$] = #neighbors of $v$ in $G$
1 stack = $\emptyset$ ; worklist = $\{v \in V \mid \text{degree}[v] < k\}$ ;
2 while worklist $\neq \emptyset$ do
3     let $v \in$ worklist ;
4     foreach $w$ neighbor of $v$ do
5         degree[$w$] $\leftarrow$ degree[w]-1 ;
6         if degree[$w$] = $k - 1$ then worklist $\leftarrow$ worklist $\cup \{w\}$
7     push $v$ on stack ; worklist $\leftarrow$ worklist $\setminus \{v\}$ ;      /* Remove v from G */
8 if $V = \emptyset$ then return TRUE else return FALSE
```

The next sections consider several coalescing problems that arise in the heuristics used in register coalescing/allocation. We recall their formulation and we refer to our previous complexity study [6] for formal definitions and proofs. Coalescing problems are unweighted (optimization of the number of static moves) or weighted (optimization biased by an approximate dynamic execution count).

Aggressive coalescing consists in finding a coloring f (with no restriction on the number of colors) such that the cost of affinities $(u, v) \in A$ not coalesced, i.e., those with $f(u) \neq f(v)$, is minimized. A simple heuristic for this NP-complete problem is to sort affinities by decreasing weights and to coalesce each affinity, one after the other, if no interference prevents it. Some heuristics for out-of-SSA conversion [20, 19] try to exploit the structure of ϕ-instructions and consider several moves simultaneously, but they have never been integrated into a unified "coloring-coalescing" scheme.

Conservative coalescing consists in finding a k-coloring f such that the cost of affinities not coalesced is minimized. It is NP-complete even if one asks the graph G_f to be not only k-colorable, but also greedy-k-colorable. A traditional heuristic is to consider moves, one after the other, so that, after each coalescing, the graph remains k-colorable. (We are not aware of a heuristic that can consider several moves simultaneously.) In such an incremental approach, one has to answer, for each considered move $a = (x, y)$, the following question: is there a k-coloring f such $f(x) = f(y)$? This *incremental conservative coalescing* problem is NP-complete for a general graph and polynomially solvable for a chordal graph. In [6], we gave a conceptual proof of this result. Here, in Section 3, we give a linear-time algorithm, which can be used as a heuristic for greedy-k-colorable graphs. For these graphs, the problem complexity remains open. Note however that asking G_f to be not only k-colorable but also greedy-k-colorable is easy: coalesce a then run Function Is_kGreedy to check that the resulting graph is greedy-k-colorable. We study this *brute-force coalescing* in Section 3.1.

The last problem is linked to optimistic coalescing: how to undo an aggressive coalescing f to go back to a k-colorable graph. In other words, *de-coalescing* consists in finding a k-coloring g, where $g(u) = g(v)$ implies $f(u) = f(v)$, that minimizes the cost of affinities not coalesced. It is NP-complete even if the aggressive phase coalesced all moves. In practice, one seeks a de-coalescing g such that G_g is not only k-colorable but also greedy-k-colorable.

We now propose advanced heuristics for these coalescing problems and compare them with previously-proposed ones.

3. GREEDY-K-COLORABLE GRAPHS AND CONSERVATIVE COALESCING

So far, two existing conservative tests exist, the tests of Briggs [8] and of George [12]. Let us examine why they are conservative when applied on a greedy-k-colorable graph. *Briggs* merges u and v if the resulting node has $< k$ neighbors of degree $\geq k$. This node can always be simplified after its neighbors of degree $< k$ are simplified, thus the graph remains greedy-k-colorable. *George* merges u and v if all neighbors of u with degree $\geq k$ are neighbors of v. After coalescing and once all neighbors of degree $< k$ are simplified, we get a subgraph of the original graph, thus greedy-k-colorable too.

Originally, these rules were used for any graphs, not necessarily greedy-k-colorable, and with some pre-colored nodes (they represent the physical machine registers and form a clique). In this context, the rules have two restrictions. Briggs's rule does not apply to pre-colored nodes since they are never simplified. George's rule applies only if v is pre-colored otherwise, if the graph is not greedy-k-colorable, there is a risk of spilling u and v instead of only u. This cannot happen with pre-colored nodes since they are never spilled. We first make a simple but important remark: *for greedy-k-colorable graphs, both rules can be used for any two nodes*. For George's rule, this is obvious as there is no spill for a greedy-k-colorable graph. For Briggs's rule, we can also decide to simplify pre-colored nodes, when possible, as any other node. Indeed, they form a clique and will thus be given different colors in any coloring. Then, we can go back to original colors by a simple permutation of colors. Hence Briggs's rule also applies to pre-colored nodes.

Surprisingly, extending Briggs's and George's rules to any two nodes already leads to significant improvements (see Section 5). However, they still give insufficient results to coalesce the many moves introduced, for example, by a basic out-of-SSA conversion. 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, i.e., the test may be applied too early in the simplify phase. This is the reason why George and Appel proposed "iterated register coalescing" [12]. Instead of giving up coalescing when the test fails, the move is placed in a sleeping list and "awaken" when the degree of one of the nodes implied in the rule changes. Thus, moves are in general tested several times, and move-related nodes should not be simplified too early to ensure the moves get 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 theoretical limitation: 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 such a situation is provided in [6]. In the following sections, we will try to overcome these two limitations.

3.1 Brute-Force Conservative Coalescing

To address the first limitation, we developed a more expensive test based on the fact that greedy-k-colorability is easy to check. As in any incremental approach, moves are considered one by one in decreasing order of weights. To test a move, instead of using an overly-conservative local rule, the two corresponding nodes are merged aggressively, then, ignoring the other affinities, a complete simplify phase is done (Function Is_kGreedy) to test if the resulting graph is greedy-k-colorable. If this "brute-force" test fails, coalescing is not conservative and the two nodes are de-coalesced.

As our experiments show (Section 5), this test is much more powerful than local rules, and even iterated register coalescing. Of course, the Is_kGreedy test is more costly than a local rule such as Briggs's and George's tests, as its complexity is linear in the graph size. Thus the overall complexity, with no optimizations (to be discussed later), would be $O(|A|(|E| + |V|))$ (A is the set of affinities). Iterated register coalescing may seem to be a linear algorithm.

However, as mentioned earlier, there is an overhead due to the fact that moves are evaluated several times. The algorithm gives up coalescing only when the worklist of moves is empty: a node is chosen and its affinities are removed. It can then be simplified and the process proceeds with the remaining nodes and affinities. A counting mechanism can be used to reduce the number of useless evaluations [16] but, still, the overall complexity is still *not* linear.

Here, when `Is_kGreedy` returns FALSE, we have two choices, either keep the move in some sleeping list for a possible re-evaluation, or give up coalescing this move. We observed that, because the `Is_kGreedy` test is more powerful, testing each move only once degrades only marginally the quality of the result. On the contrary, keeping moves and reconsidering them each time the graph changes is 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. Also, with little effort, this "brute force" algorithm can become really competitive in terms of speed as follows. 1. Use local rules first: if Briggs's or George's rule applies, no need to check a move with brute-force coalescing. 2. Work with the original graph G instead of copies. Function `Is_kGreedy` is indeed not destructive, it can be implemented in a Chaitin-like fashion using worklists. 3. Nodes not involved in a move can be simplified once for all. This information can be given to Function `Is_kGreedy` so that it does not need to re-simplify them each time a move is considered. 4. Degrees can be stored to avoid recomputing them. 5. Instead of merging u and v before checking the graph is still greedy-k-colorable if (u, v) is coalesced, use degree tricks to simulate the merge.

Function `Brute_Force_Improved`(G, A, simplified, degree)

```
Data: Graph $G = (V, E)$, affinities $A$, weight function $w : A \to \mathbb{N}$
Output: TRUE if $G$ is greedy-$k$-colorable, FALSE otherwise.
Result: Graph $G$ is conservatively coalesced.
stack ← simplified ; $B \leftarrow A$ ;
Function update_worklist($x$) ; /* Function to move x to the right worklist. */
begin
    remove $x$ from the worklist it belongs to;
    if $\exists(u, v) \in B$, $u = x$ or $v = x$ then freeze_worklist← freeze_worklist $\cup \{x\}$ ;
    else if degree[$x$] < $k$ then simplify_worklist← simplify_worklist $\cup \{x\}$ ;
    else hi_degree_worklist← hi_degree_worklist $\cup \{x\}$ ;
end
foreach $x \in V\setminus$ simplified do update_worklist($x$)
while TRUE do
    if simplify_worklist $\neq \emptyset$ /* Now, simplify the graph */ then
        let $x \in$ simplify_worklist;
        simplify_worklist← simplify_worklist $\setminus \{x\}$ ;
        foreach $w$ neighbor of $x$ do
            degree[$w$] ← degree[w]-1 ;
            if degree[$w$] = k-1 then update_worklist($w$)
        push $x$ on stack ;
    else if $B \neq \emptyset$ No simplifiable node, try to coalesce */ then
        let $(x, y) \in B$ ; $B \leftarrow B \setminus \{(x, y)\}$;
        if $x$ and $y$ are neighbors /* cannot be coalesced */ then
            update_worklist($x$) ; update_worklist($y$) ;
        else if Briggs_George_Coalescing(x,y) then
            merge $x$ and $y$ into $xy$ in $G$ ;
        else
            merge $x$ and $y$ into $xy$ in $G$ ; let degree' = copy of degree ;
            if Brute_Force_Improved($G, \emptyset$, stack, degree') = FALSE then
                un-merge $x$ and $y$ in $G$ ;
                update_worklist($x$) ; update_worklist($y$) ;
        if node $xy$ exists /* i.e., x and y have been merged */ then
            remove $x$ and $y$ from any worklist ;
            update_worklist($xy$) ;
    else if hi_degree_worklist $\neq \emptyset$ then return FALSE else return TRUE
```

These optimizations lead to Function `Brute_Force_Improved`, a heuristic that outperforms all previous conservative approaches with acceptable running times. It uses the same framework as iterated register coalescing, but with recursive calls, and is thus not more complicated to implement. It is 2× slower than the basic iterated register coalescing presented by Appel and George [3] (without the speed improvements of Leung and George [16]) but reduces by a factor of 2 the cost of remaining moves for the same suite of graphs [2]. Surprisingly, at the price of a 3× slow down (roughly), it is even a lot better (15%) than state-of-the-art optimistic coalescing, which contradicts the common belief. These last performances are with a well-chosen order of affinities, see details in Section 5.

Brute-force coalescing is still an incremental strategy, meaning that the graph remains greedy-k-colorable after the coalescing of each particular move. We try to go even further in the next section.

3.2 Chordal-Based Incremental Coalescing

As mentioned before, it is a limitation to require the graph to be greedy-k-colorable after each coalescing. Indeed, to get a greedy-k-colorable graph after coalescing a move, it may be needed to coalesce other moves or even merge nodes not linked by any affinity. This fact was pointed out by Vegdahl [21] and also used in the optimistic+ algorithm (extended optimistic coalescing) of Park and Moon [17]. However, these additional merges are done blindly, without any guarantee that a coalescing is indeed enabled. Actually, for a given move, the real conservative coalescing problem is as formulated in Section 2: Deciding if, after one particular coalescing, the graph is still k-colorable, i.e., if it can become greedy-k-colorable thanks to some additional node merges. Such a test can coalesce moves beyond traditional conservative coalescing.

As recalled in Section 2, this can be checked in polynomial-time for a chordal graph, but the proof of this fact was only conceptual [6]. We developed a linear-time algorithm whose correctness is based on the following lemmas (proofs are available in [5]):

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

LEMMA 2. *Let G, with nodes $v_1, \ldots, v_n$, be a chordal k-colorable graph. If for all i, $1 < i < n$, 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.*

Suppose we want to coalesce x and y, two non-interfering nodes of a chordal graph G. First, G is simplified maximally (i.e., nodes with degree $< k$ are removed) without simplifying x and y. If no other node remains, x and y can be given the same color and coalescing them keeps the graph greedy-k-colorable. Otherwise, the two lemmas show that the remaining graph is an interval graph, for which we can easily compute a representation by removing in priority nodes of smallest degree. The rest of the algorithm looks for a "path" of non-interfering nodes between x and y in the interval graph, as suggested in [6]. The existence of such a path proves that x and y can be coalesced, along with all the nodes of this path, and G remains greedy-k-colorable. The non-existence of such a path proves that no matter how G is colored, x and y will always have a different color, hence they cannot be coalesced. See details in [5]. Function `Chordal_Coalescing` implements this strategy.

In terms of complexity, chordal-based coalescing is similar to a complete simplification (with a return phase such as the classical "select" coloring phase). Thus, its complexity is similar, in order of magnitude, to brute-force coalescing. Also, used in complement to brute-force coalescing, its use does not increase the running times too much as most simplifications are already done. We can also apply the same algorithm even if the graph obtained after the first phase of simplification is not chordal. Then, however, it is just a heuristic for greedy-k-colorable graphs that amounts to add interferences on the fly so that the graph "looks" chordal. Variants are certainly possible, with strategies to add interferences and/or merge nodes, but we did not try to go further in this direction. However, what can easily be done is to select a path that, if possible, coalesces additional moves and avoids merging nodes not related by affinities so as to not constrain too much the resulting graph.

Function `Chordal_Coalescing`(G, a)

```
   Data: Graph $G = (V, E)$, $a = (x, y)$ affinity
   Result: TRUE if coalescing $a$ is conservative, possibly thanks to other merges
1  if degree[$x$] $\geq k$ then $x, y \leftarrow y, x$ ;
   /* Traverse the set of intervals from x to y */
2  like_x[$x$] $\leftarrow$ TRUE; alive $\leftarrow \{x\}$ ; dummy_like_x $\leftarrow$ FALSE;
3  just_removed $\leftarrow \emptyset$ ; like_x[just_removed] $\leftarrow$ FALSE;
4  repeat
5      let $v \in$ nodes $\setminus \{y\}$ with smallest degree ;
6      dummy_like_x $\leftarrow$ dummy_like_x $\vee$ like_x[just_removed];
7      foreach $w$ neighbor of $v$ do
8          degree[$w$] $\leftarrow$ degree[$w$]-1 ;
9          if $w \notin$ alive then
10             alive $\leftarrow$ alive $\cup \{w\}$ ; like_x[$w$] $\leftarrow$ dummy_like_x ;
11     if #alive = $k$ then dummy_like_x $\leftarrow$ FALSE;
12     if #alive > $k$ then return FALSE ; /* Cannot happen for chordal graphs. */
13     push $v$ on stack ;
14     nodes $\leftarrow$ nodes $\setminus \{v\}$ ; alive $\leftarrow$ alive $\setminus \{v\}$ ; just_removed $\leftarrow v$ ;
15 until nodes = $\{y\}$ ;
16 if like_x[$y$] = FALSE then return FALSE;
   /* Else, construct the path linking x and y */
17 path $\leftarrow \{y\}$ ; current $\leftarrow y$ ;
18 while $v \leftarrow$ pop stack, $v \neq x$ do
19     if $v$ not a neighbor of current then
20         if like_x[$v$] then
21             path $\leftarrow$ path $\cup \{v\}$ ; current $\leftarrow v$ ;
22 merge all $v \in$ path into a single node in $G$ ;
23 return TRUE
```

In terms of quality of results, our experiments show that chordal-based coalescing does improve brute-force coalescing, but only slightly for the graphs of Appel and George. Section 5 discusses how far it is from optimality, thanks to an optimal integer linear programming approach. In practice, because it is more complicated to implement, with only a marginal improvement, we believe it is maybe not worthwhile. However, in theory, this is the most advanced conservative algorithm proposed so far. The next section aims at developing more advanced *optimistic* coalescing strategies.

4. OPTIMISTIC: DE-COALESCING AFTER AGGRESSIVE COALESCING

Merging two nodes can transform a graph that is not greedy-k-colorable into a greedy-k-colorable graph. This observation is the main motivation for aggressive coalescing. It may be 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 (i.e., split back)?

After aggressive coalescing, Park and Moon [17] proceed with the standard simplify/select phases. In the select phase, if the result of a merge cannot be colored (potential spill), it is split back into its original nodes. It is not safe to color all of them right away, even when it is possible, because this could prevent the coloring of nodes that are still on the stack and not colored yet. However, if some are colored with a unique color and the others discarded for now, the rest of the select phase can continue safely: hence Park and Moon choose heuristically which nodes to color, and the others are scheduled to be colored or spilled after the select phase.

Appel and George [3] pointed out that, if all nodes (except pre-colored nodes) have degree $< k$ initially, they can always be colored whatever the other colors. Based on this observation, they modified Park and Moon de-coalescing but it is not clear from their explanations whether they color some nodes at the very end, as Park and Moon do, but with the guarantee that no spill occurs, or if they color all of them immediately, even if this can create subsequent splits, even of nodes that were not marked as potential spills. To try to retrieve their results, provided for a large collection of graphs in the coalescing challenge web page [2], we developed several split strategies during the select phase. When no color is found for a node, we split it back into its primitive nodes as follows. Let P be the set of such primitive nodes. We first compute the colors available for each of them and we consider the set A_P of affinities (x, y) such that either both x and y belong to P, or one is in P and the other is already colored. Then we tried the three following strategies.

1. *Appel-George type:* Consider all affinities (x, y) in A_P in decreasing order of weights and coalesce x and y if there exists a color suitable for both x and y.

2. *Park-Moon type:* Select a color c that maximizes the sum of the weights of the affinities $(x, y) \in A_P$ such that x and y can be colored with c. Coalesce the corresponding nodes, color the resulting node with c, and color all other nodes at the end of the select phase, using a biased coloring.

3. *Iterated Park-Moon:* Select a color c and coalesce nodes as in the second strategy. Repeat the color selection for the remaining affinities in P until $P = \emptyset$.

Strangely, none of these approaches give results close to those of Appel and George. Also, surprisingly, Strategy 1 is the worst, certainly because it does not have a global view on the affinities within the split node. Nevertheless, we point out that the three of them are guaranteed to be spill-free only if all initial nodes have degree $< k$, and it is not clear how to adapt these heuristics to general greedy-k-colorable graphs. Furthermore, even if the initial degrees are $< k$, except pre-colored nodes, a problem may occur with Strategies 1 and 3 if pre-colored nodes can be simplified too (as we do). As several colors are given right away to the different primitive nodes, a simplified node might become not colorable. The node can then be split into its primitive nodes, all of degree $< k$, unless one is pre-colored. Luckily, this potential problem never occurred in any of the 474 graphs. In addition to these applicability limitations, a weakness of these 3 approaches in terms of quality of results is that, instead of de-coalescing moves, possibly one by one, many moves are de-coalesced, even if not needed, when one node is split back. As this process is done in the coloring phase, only a form of biased coloring can help re-coalescing these useless de-coalesced moves. It is not clear how to use classical conservative coalescing techniques. For these reasons, we prefer to de-coalesce based on the graph structure itself, not on the particular order of nodes in the stack. An interesting side-effect of this approach is that the final graph is then greedy-k-colorable and that we can use conservative techniques afterwards to improve our de-coalescing phase.

Our approach is described in Function `De-coalescing`. It applies to any greedy-k-colorable graph and produces a greedy-k-colorable graph. After aggressive coalescing, some affinities are de-coalesced as long as the graph is not greedy-k-colorable. These affinities are selected as follows. During the check for greedy-k-colorability (Function `Is_kGreedy`), if all nodes of the current subgraph have degree $\geq k$, the cheapest node to de-coalesce is chosen and split into two nodes. It may be necessary to de-coalesce several affinities to get two separated nodes, so we use a min-cut algorithm on the affinities coalesced in the node, but simpler approaches are possible. To choose quickly which node to de-coalesce, each node is given a lower bound of the cost of its de-coalescing, i.e., the min-cut cost. Currently, this bound is set to the smallest affinity weight coalesced in the node. After de-coalescing, the simplify phase continues until the graph becomes empty or another de-coalescing is necessary. This way, first we avoid de-coalescing an affinity in the area where it does not help, i.e., among the nodes already simplified, second we give up coalescing the cheapest moves first. However, as in Park and Moon approach, de-coalescing a move can increase the degree of nodes already simplified, thus, in general, we need to perform several passes (at most 3 in practice) until no de-coalescing is done, which ensures the graph is greedy-*k*-colorable.

```
Function De-coalescing($G, A, w$)
Data: Graph $G = (V, E)$, affinities $A$, weight function $w : A \to \mathbb{N}$
repeat
    nodes ← $V$ ; stack = $\emptyset$ ; some_de-coalescing ← FALSE;
    while $nodes \neq \emptyset$ do
        if $\exists v \in$ nodes, degree[$v$] < $k$ then
            nodes ← nodes \ {$v$} ; push $v$ on stack ;          /* Simplify v */
            foreach $w$ neighbor of $v$ do  degree[$w$] ← degree[$w$]-1 ;
        else
            $n$ ← get_min_cost(nodes) ;     /* coalesced non-simplified node,
            with smallest cost to de-coalesce */
            $(x, y)$ ← affinity in $n$ with smallest weight ;
            $G$ ← de-coalesce_min_cut($G$, $(x, y)$) ;        /* de-coalesce x and y
            using a min_cut on the affinities of n */
            nodes ← (nodes \ {$n$}) ∪ {$x, y$} ;
            compute degree[$x$] and degree [$y$] ;
            some_de-coalescing ← TRUE;
until some_de-coalescing = FALSE ;
```

As we show in Section 5, our de-coalescing scheme alone is as good as the state-of-the-art optimistic coalescing algorithm (here we mean in terms of coalesced affinities as we compare algorithms that do not spill at all.) However, it has two strong advantages. First, it can be applied to any greedy-*k*-colorable graphs, without requiring any spill. Second, as it is not intermixed with coloring (the select phase), it can be followed by conservative coalescing to clean up possible useless de-coalescings: then, even a simple conservative coalescing such as Briggs's and George's rules improves the results by 8%. To our surprise, however, it does not equal the quality of "chordal-based" coalescing, not even the quality of "brute-force" coalescing. The problem may come from our way to choose nodes to de-coalesce. But it is hard to define a good indicator of the benefit and cost of a de-coalescing, so as to guide the node selection. Also, it is possible that some bad decisions are made by the aggressive phase (for example, coalescing an apparently-expensive affinity that prevents the coalescing of many cheap ones), which are then difficult to repair by a greedy de-coalescing.

5. EXPERIMENTS AND EVALUATION

5.1 Methodology

Coalescing is challenging for graphs with many affinities and high register pressure. In particular, a graph-based spill-everywhere algorithm leads to a too simple coalescing problem. It this thus difficult to find a good set of benchmarks, hard enough to solve, large enough, to experiment and evaluate various strategies in detail. The coalescing challenge [2] provides such an interesting collection of graphs on which state-of-the-art coalescing algorithms were tested. Also, the low number of registers (six) allowed Grund and Hack to give optimal solutions for all but 3 graphs [14]. We use them to compare the different heuristics, either optimistic or conservative. We point out that, to measure the quality of coalescing algorithms, it is more fair to work on such a collection of graphs than to measure execution time of codes for some platform: the latter gives results that are biased by many factors and are not reproducible, and anything can be claimed. Here, we measure the number (or weight) of coalesced moves, i.e., of copies that are removed, which is always, in general, a benefit for the generated code.

Benchmarks suite In their spill algorithm, Appel and George 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 (maybe 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.

Performance criterion As the graphs correspond to live-ranges that are split at each program point, there are a large number of affinities in the graphs (compared to standard graphs where affinities usually correspond to original moves, ABI constraints, or possibly SSA splits) and most of those are straightforward to coalesce. Therefore, it does not make sense to evaluate the quality of heuristics using a ratio with the total weight of initial affinities. It is more significant to focus on the moves that are hard to coalesce, i.e., to use a ratio with the moves that are *not* coalesced.

For that, to evaluate the quality of an heuristic h, we compute, for each graph, the cost $c(h)$ of the coalescing given by h, i.e., the sum of the weights of the affinities *not* coalesced (remaining moves):

$$c(h) = \sum \{w(a) \mid a = (x, y) \in A, a \text{ is not coalesced by } h\}$$

We then divide the cost $c(o)$ of the optimal solution o (provided by Grund and Hack) by the cost $c(h)$ of the heuristic h. This gives us a performance ratio $q(h) \leq 1$ that measures the percentage (in weight) of the remaining moves that could still be coalesced:

$$q(h) = \frac{c(o)}{c(h)} \qquad c(o) = \text{optimal cost (remaining moves)}$$

For example, if a heuristic h reaches 0.8, it means that, in the optimal solution o, there are 20% (in weight) less moves than for the heuristic h. The traditional performance ratio when evaluating algorithms is $c(h)/c(o)$, comparison with the optimal solution. The inverse gives us a quick way to compare two heuristics h_1 and h_2 with $q(h_1) - q(h_2)$. For example, if $q(h_1) - q(h_2) = 0.1$, we say that h_1 improves h_2 by 10%. Actually, to get an exact percentage when $q(h_1) \geq q(h_2)$, we should compute $1 - c(h_1)/c(h_2)$, but $1 - c(h_1)/c(h_2) = 1 - q(h_2)/q(h_1) = (q(h_1) - q(h_2))/q(h_1) \geq q(h_1) - q(h_2)$ and $1 - c(h_1)/c(h_2) \approx q(h_1) - q(h_2)$ when $q(h_1) \approx 1$. Thus, this is a conservative estimation, h_1 actually improves h_2 by a bit more than 10% in our example. Figures 1 and 2 give the average value of $q(h)$ for each heuristic. We also give a "weighted" ratio where graphs are weighted by their number of instructions so that bigger graphs (such as #139), usually more difficult to coalesce, get more importance than very small ones (such as #001). As we will see, both average ratios lead to the same conclusion, i.e., if h_1 improves h_2 with one ratio, it also improves it with the other ratio. For this reason, and because we believe the weighted ratio is a better indicator of the quality of a heuristic, we refer to the weighted version when giving any numbers (percentages) in the discussions.

Note on affinity ordering The order in which affinities are considered for coalescing is crucial, and it is a good idea to consider them in decreasing order of their weight, so that bigger affinities get coalesced first. However, many affinities have the same weight, and some total ordering must be chosen. This is never mentioned in the

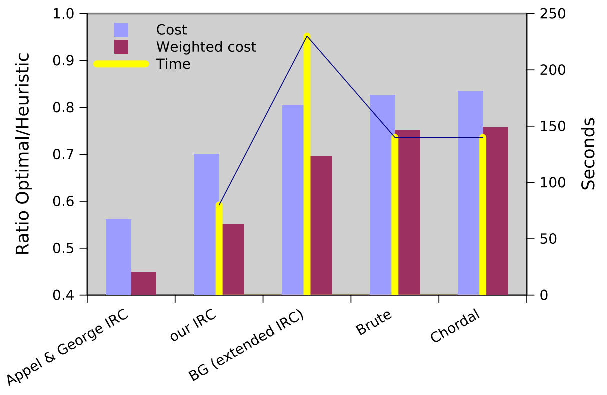

Figure 1: Conservative heuristics.

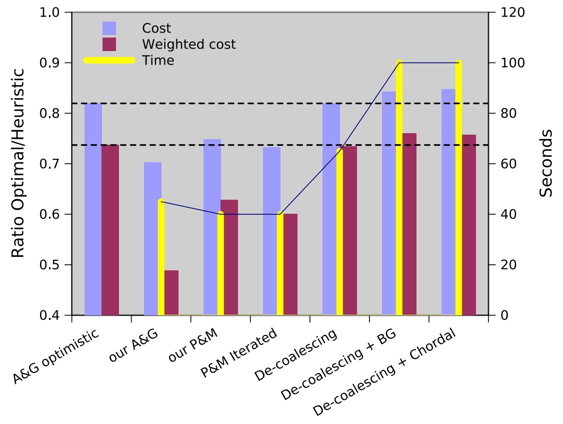

Figure 2: Aggressive heuristics.

literature but, as we will see, this ordering choice has a strong impact on the quality of the results. It seems reasonable to guide the ordering using, for example, the knowledge of the program structure or some graph properties (e.g., node degrees). We will discuss several orderings in Section 5.4. Before, to provide reproducible results, we chose a deterministic ordering: in case of equal weight, we use the order in which affinities appear in the graph description file ("first seen, first taken"). 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, except for the external results (given in [2]) for which we do not know the ordering that was used. Nevertheless, we also give the results for our implementation of these schemes.

5.2 Conservative Heuristics

Figure 1 shows the quality of the conservative heuristics for the criterion $q(h)$. The optimal has value 1, hence the higher a heuristic, the better. Each heuristic is evaluated by the average on all graphs (left column), the weighted average on all graphs (right column), and, when available (i.e., for our implementations), the overall time (middle bar) spent by the heuristic on all 474 graphs.

The first results are the performance of iterated register coalescing (IRC) as given on the coalescing challenge web page [2]. The second heuristic is our implementation of IRC, which gives 10% better results, although it is the same algorithm.[1] The third heuristic, BG, uses the rules of Briggs and of George, extended to any type of nodes (pre-colored or not) as explained in Section 3. This simple change improves by 15% the quality of our IRC implementation, but at the price of a (roughly) 3× slowdown. Finally, the last heuristics, `Brute` and `Chordal`, both implement Function `Brute_Force_Improved`, with an additional call to Function `Chordal_Coalescing` for the second, as described in Section 3. The `Brute` heuristic improves BG by 5%, hence it is 20% better that standard IRC (30% if compared to the results provided by Appel and George in [2]), while being only 1.7× slower.

For our developments, we started with an implementation of IRC and we experienced that the effort to extend it into an implementation of `Brute` was small, making this improvement worthwhile. Also, we measured that, without the improvements proposed in Section 3.1, i.e., with a naive use of Function `Is_kGreedy` for each tested affinity, the heuristic `Brute` processes all graphs in more than one hour instead of 135 seconds. It is 10× slower on average and more than 30× slower on the biggest graphs. Finally, the chordal rule added in `Chordal` only improves `Brute` by around 1%, while being more complicated to implement. However, the execution time overhead is not significant, so it is a "free" percent for whoever needs it and is ready to implement it.

[1]This may be due to the use of a different affinities ordering.

5.3 Optimistic Heuristics

Figure 2 shows the quality of the aggressive heuristics. The first heuristic is the variant of Park&Moon optimistic coalescing developed by Appel and George [3] and whose results are, again, provided in [2]. The next three heuristics are our (unsuccessful) attempts to reproduce these results: these are the heuristics 1, 2 and 3 of Section 4, i.e., Appel&George type, Park&Moon type, and our "iterated" version of Park&Moon. They give results worse by 10% to 25%. The fifth heuristic, `De-coalescing`, uses our de-coalescing scheme, after an aggressive part, as explained in Section 4. It alone gives results of the same quality as the optimistic provided by Appel and George, but requires more time than our implementation of optimistic coalescing. However, our scheme produces a greedy-k-colorable graph and was designed to enable a conservative coalescing post-pass. So, we tried, after `De-coalescing`, two conservative techniques, the cheapest and less aggressive one, i.e., Briggs's and George's rules (BG), and the most aggressive one, our chordal rule (`Chordal`). These rules use a little more time and improves the results by 2.5%. This is not much compared to Appel&George version of optimistic coalescing (first column), but it is 13% better than our implementations of optimistic coalescing, with a 2× slowdown. According to these results, it appears that, after our de-coalescing, Briggs's and George's rules are enough.

Compared to the conservative heuristics, the best optimistic coalescing scheme equals the best conservative one (the unweighted average is 2% better with optimistic coalescing) and it is about 30% faster. Also, the results are just slightly better than Appel&George version of optimistic coalescing. However, the next section shows that far better results can be obtained, especially with conservative coalescing, thanks to different (better) affinities orderings.

5.4 Ordering the Affinities

In Section 5.1, we mentioned that the ordering of affinities of equal weight has a strong impact on the quality of the results. The results of Sections 5.2 and 5.3 correspond to a particular canonical ordering (order of the program, basically). We now show that, with an adequate ordering of the affinities, our algorithms can be improved further. All the orderings we tried consider affinities by decreasing order of weights, since it is usually better, though of

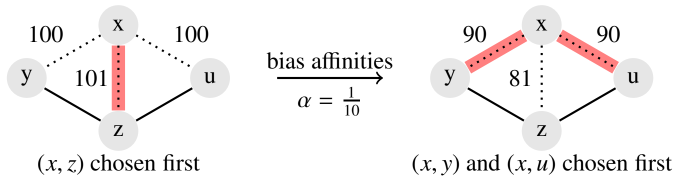

Figure 3: Motivation for biased affinity weights.

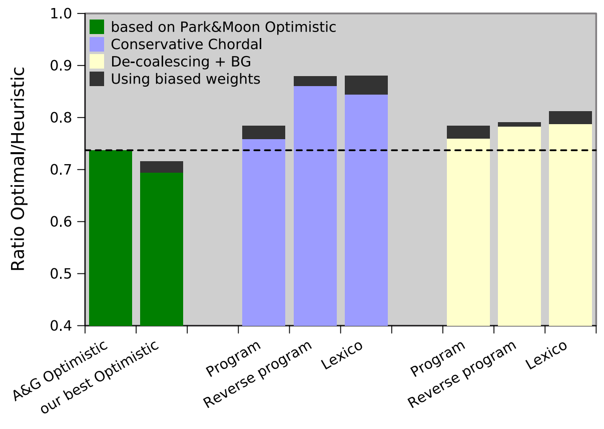

Figure 4: Quality results for different affinities orderings.

course not optimal, to first coalesce affinities which cost more. The tie-breakers we tried in case of equal weights are the following:

1. *Program*: the affinity appearing *first* in the program, i.e., in the graph description file, gets the priority.
2. *Reverse*: the affinity appearing *last* in the program, i.e., in the graph description file, gets the priority.
3. *Lexico*: affinities are represented by (x, y) with $x < y$, where $x < y$ if, in the graph description file, x (or one of its constituting nodes if x is a coalesced node) appears before y (or any of its constituting nodes if y is a coalesced node). Then, (x, y) gets priority over (x', y') iff $x < x'$ or $x = x'$ and $y < y'$.

It is also important to update carefully the affinities when the graph changes to keep the benefit of using a good ordering. If (x, z) and (y, z) are two affinities, and x is merged with y to form the node xy, the two affinities are replaced by (xy, z) with weight $w(x, z) + w(y, z)$. Also, for *Program* (resp. *Reverse*), the order of (xy, z) is the minimum (resp. maximum) order of (x, z) and (y, z).

Biased affinity weights We also modified the global ordering of affinities because of the following remark. Suppose (x, y) and (x, z) are two affinities such that y and z interfere (see Figure 3). Coalescing both is not possible as coalescing one constrains the other. When coalescing (x, y), $w(x, y)$ is saved and $w(x, z)$ is lost. This becomes a problem if there is another affinity (x, u) where z and u interfere. If (x, z) has a weight even slightly greater (say 101 versus 100), it will be chosen first, and this will prevent coalescing the two others. Here, the final cost will be 200 while choosing to coalesce (x, y) and (x, u) leaves a cost of 101. To avoid this situation, we devised a strategy called *bias*. When applied, our algorithm works with modified weights, computed from the initial weights. Of course, the final cost of the remaining affinities is still calculated using the initial weights. For each affinity a, we initialize $w_{bias}(a)$ to $w(a)$. Then, whenever we find a triangle x, y, z such as in Figure 3, we subtract $\alpha w(x, z)$ to $w_{bias}(x, y)$ and $\alpha w(x, y)$ to $w_{bias}(x, z)$. We fixed $\alpha = \frac{1}{10}$, arbitrarily, which gives the desired behavior.

Results Figure 4 shows the results of our best optimistic and conservative algorithms with different affinities orderings. The first two columns are two versions of the optimistic coalescing: the one provided by Appel and George, and our best implementation (using the *Lexico* ordering and *bias*), which is still not as good. The next three columns show the conservative results. We displayed only results for `Chordal`, which performs best, but the effects of ordering are similar on the other conservative techniques (in particular `Brute`, which has equivalent results). Here, using the *Reverse* order instead of the normal program order greatly improves (by 10%) the quality. When using *bias*, both *Lexico* and *Reverse* orderings give the best overall results we managed to achieve: 0.88. This means that the optimal solution can improve the result only by 12%, while this is more than 15% better that Appel and George's version of optimistic coalescing, but for a 3× slow-down. Finally, the last three columns show the effects of ordering on our best optimistic technique, `De-coalescing` followed by `BG`. So far, it is not clear which ordering works better (in particular *Reverse*) and why.

For the optimistic strategy, the affinity ordering also plays a role, but not as important as for our conservative techniques. The same holds for *bias*. Our interpretation is that the aggressive part may coalesce the wrong affinities, and since it is aggressive, it will coalesce them whatever the ordering is. The current de-coalescing phase is then unable to decide to de-coalesce the "bad" nodes created. A simple experiment confirms these doubts: on the one side, `Brute` only, and on the other side, `Brute` followed by aggressive coalescing, then de-coalescing, then `Brute` again. The second strategy is more that 5% worse than the first one. This means that, among the moves not coalesced by `Brute`, the aggressive part chooses some that are not de-coalesced later. Instead, some nodes created by merges in the first phase of `Brute` are de-coalesced.

In conclusion, without a better ordering for aggressive coalescing, or a better de-coalescing part, our optimistic techniques do not reach the quality of our conservative heuristics, `Brute` or `Chordal`, which work better than the other techniques. In other words, the non-conservative decisions taken by aggressive coalescing are hard to repair in the de-coalescing phase, compared to an advanced conservative approach. In past studies, optimistic coalescing appeared better than conservative coalescing because Briggs's and George's tests are far too conservative, even in an iterated framework.

5.5 Inside the `Chordal` Scheme

We end our experimental study with a deeper analysis of our most advanced conservative technique, `Chordal`, analyzing which rule decided to coalesce and the time it needed to take the decision.

When `Chordal` is run, affinities are coalesced either by the `BG` rule, by the `Brute` rule if `BG` fails, or by the final `Chordal` rule if `Brute` fails. To evaluate the quality of these different rules, we implemented an "exact" conservative test using integer linear programming (ILP): if the "path search" of `Chordal` fails on the remaining graph simplified by `Brute`, the low number of nodes (usually between 10 and 40) allowed us to check, exactly, if there exists a coloring of this remaining graph such that the affinity being tested can be coalesced. If such a coloring exists, the two ends of the affinity and every other node with the same color are merged in the graph, provided that the resulting graph is still greedy-k-colorable.[2]

[2]If the graph is only k-colorable, merging all the nodes with the same color, for all colors, produces a k-clique, hence a greedy-k-

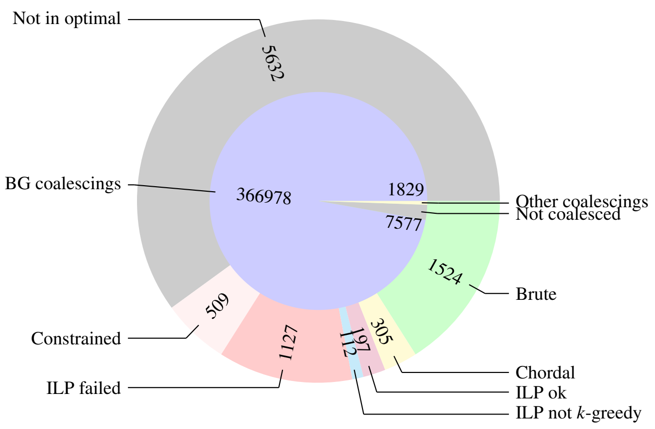

Figure 5: Effort of basic coalescing rules.

Compared to a pure `Chordal`, this strategy changes the affinities that are finally coalesced, but can still give an interesting view of the relative performance of each coalescing rule. The double pie chart in Figure 5 analyzes which rule was activated. The inner pie shows what happened to the 376,496 affinities of all 474 graphs. The `BG` rules managed to coalesce 97.5% of them and 0.5% were coalesced by `Brute` or `Chordal`. This is the reason why `Chordal` and `Brute` remain fast. The outer pie classifies the 9406 (2.5%) affinities not coalesced by `BG`. Our heuristics achieve 19%: `Brute` coalesced 1524 affinities (16%) and `Chordal` 305 more (3%). Our ILP test found that 309 affinities (3%) could have been coalesced but only two third of them (197) led to a greedy-k-colorable graph. For 1127 affinities (12%), there was no coloring, and the remaining 6141 (66%) were constrained by other affinities and never tested. In comparison, the optimal solution (although optimized for weight and not number of affinities) leaves 5632 uncoalesced affinities (60%), which means that `Brute` + `Chordal` coalesce 48% of hard-to-coalesce affinities and that $(6141+1127-5632)/(9406-5632) = 43\%$ of them *cannot be satisfied* even with an optimal incremental conservative technique, unless affinities are chosen in a different order. To conclude, this proves that `Brute` and `Chordal` perform quite well since they coalesce 19% of these affinities and an optimal incremental technique could coalesce at best 22%.

Surprisingly, for every affinity coalesced by `Chordal`, the remaining graph after simplification was an interval graph. Thus, `Chordal` applied positively only to the cases where it is optimal. Thus, using `Chordal` as a heuristic for greedy-k-colorable graphs does not seem to be efficient enough, at least for the graphs encountered here, to catch some of the remaining 3% available.

Figure 6 shows the time spent (a total of 135 seconds), for the whole set of graphs, in the different parts of the `Chordal` heuristic. The `BG` rules and the overhead due to the chordal rule represent, respectively, only 1.8% and 0.5% of the time, less than the 5.5% spent in initialization. The 31.5% used for maintenance concern graph and worklists updates (merging nodes, computing degree, sorting affinities, etc.), but does not take into account maintenance of worklists inside `Brute`. The latter is included in the biggest part of the pie-chart, the `Brute` rule, which takes 60.7% of the time. We also measured that, for small graphs, most of the time is spent in initialization, or in updating the graph and the worklists. As graphs grow bigger, the `Brute` part takes proportionally more and more time, as each test needs to simplify the whole graph. Actually, 80 of

colorable graph. But this would constrain too much the remaining graph and may prevent further coalescings.

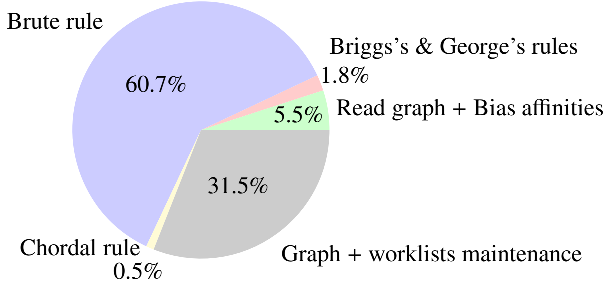

Figure 6: How the 135 seconds of `Chordal` were spent.

the 135 seconds were spent on the 3 biggest graphs, which contain respectively 18445, 20011, and 28640 nodes. We also point out that beyond the $2^{14} = 16,384$ nodes limit, our graph library cannot use an adjacency bit matrix anymore because of memory limitations. We need to rely on searches in adjacency lists instead. However, we measured that this is not the limiting factor.

6. CONCLUSION AND FUTURE WORK

It is a common belief that optimistic coalescing outperforms conservative coalescing. Based on our previous theoretical work [6], we developed an advanced incremental conservative strategy that can coalesce more than half (in weight) of the moves left by the well-known iterated register coalescing. It also outperforms traditional optimistic coalescing by about 15%. Our conservative tests (brute force or chordal-based) are more costly than the simple tests of Briggs and of George. However, by using them only when the quick tests fail and with additional implementation strategies, our final algorithm is less than 2× slower than the iterated register coalescing of Appel and George. This is also because iterated register coalescing may test some affinities several times while our heuristic tests each affinity only once and decides 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 version of the optimistic algorithm) and ensures it is still greedy-k-colorable after coalescing. This enables the use of an additional (quick) phase of Briggs/George conservative coalescing, which leads to results better than traditional optimistic coalescing but still a bit worse than our conservative strategy. We think this is because it is more difficult to devise a good de-coalescing indicator.

During our experiments, we measured noticeable differences for the quality of the final results, depending on the ordering in which affinities with equal weights are considered. In particular, we verified that guiding this ordering, using for example the program structure and simple considerations such as a lexicographic order, provides better results in average. We believe this point is worth exploring for both the aggressive/optimistic and conservative coalescing problems, especially for conservative coalescing.

Finally, our linear-time incremental chordal-based conservative test for general greedy-k-colorable graphs provides only slight improvements over our "brute force" test. It is maybe a too direct adaptation of the optimal test for chordal graphs. However, we measured, thanks to an ILP formulation of the optimal solution, that our test misses few of the remaining removable moves.

Acknowledgments

The authors thank C. Guillon and B. Dupont de Dinechin for their help and support, and S. Hack and D. Grund for making their optimal ILP-based results available and for fruitful discussions.

7. REFERENCES

[1] A. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[2] A. Appel and L. George. Coalescing challenge. `http://www.cs.princeton.edu/~appel/coalesce`, 2000.

[3] A. W. Appel and L. George. Optimal spilling for CISC machines with few registers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, pages 243–253, Utah, United States, June 2001. ACM Press.

[4] 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.

[5] F. Bouchez, A. Darte, and F. Rastello. Improvements to conservative and optimistic register coalescing. Technical Report RR2007-41, LIP, ENS-Lyon, France, Oct. 2007.

[6] F. Bouchez, A. Darte, and F. Rastello. On the complexity of register coalescing. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 102–114. IEEE Computer Society Press, March 2007.

[7] P. Briggs. Register allocation via graph coloring. PhD Thesis Rice COMP TR92-183, Department of Computer Science, Rice University, 1992.

[8] 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.

[9] 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.

[10] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[11] 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.

[12] L. George and A. W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[13] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

[14] D. Grund and S. Hack. A fast cutting-plane algorithm for optimal coalescing. In S. Krishnamurthi and M. Odersky, editors, *Compiler Construction 2007*, volume 4420 of *Lecture Notes In Computer Science*, pages 111–125. Springer, March 2007. Braga, Portugal.

[15] 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.

[16] A. Leung and L. George. A new MLRISC register allocator. Technical report, New York University, 1999.

[17] J. Park and S.-M. Moon. Optimistic register coalescing. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 26(4), 2004.

[18] F. M. Q. Pereira and J. Palsberg. Register allocation via coloring of chordal graphs. In *Proceedings of APLAS'05, Asian Symposium on Programming Languages and Systems*, pages 315–329, Tsukuba, Japan, Nov. 2005.

[19] F. Rastello, F. de Ferrière, and C. Guillon. Optimizing translation out of SSA using renaming constraints. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*, pages 265–278. IEEE Computer Society, 2004.

[20] V. C. Sreedhar, R. D. Ju, D. M. Gillies, and V. Santhanam. Translating out of static single assignment form. In A. Cortesi and G. Filé, editors, *6th international Symposium on Static Analysis*, volume 1694 of *LNCS*, pages 194–210. Springer Verlag, 1999.

[21] S. R. Vegdahl. Using node merging to enhance graph coloring. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI'99)*, pages 150–154, New York, NY, USA, 1999. ACM Press.