Live-range Unsplitting for Faster Optimal Coalescing (extended version)

Sandrine Blazy* and Benoit Robillard

ENSIIE, CEDRIC

Abstract. Register allocation is often a two-phase approach: spilling of registers to memory, followed by coalescing of registers. Extreme live-range splitting (i.e. live-range splitting after each statement) enables optimal solutions based on ILP, for both spilling and coalescing. However, while the solutions are easily found for spilling, for coalescing they are more elusive. This difficulty stems from the huge size of interference graphs resulting from live-range splitting.

This report focuses on optimal coalescing in the context of extreme live-range splitting. We present some theoretical properties that give rise to an algorithm for reducing interference graphs, while preserving optimality. This reduction consists mainly in finding and removing useless splitting points. It is followed by a graph decomposition based on clique separators. The last optimization consists in two preprocessing rules. Any coalescing technique can be applied after these optimizations.

Our optimizations have been tested on a standard benchmark, the optimal coalescing challenge. For this benchmark, the cutting-plane algorithm for optimal coalescing (the only optimal algorithm for coalescing) runs 300 times faster when combined with our optimizations. Moreover, we provide all the solutions of the optimal coalescing challenge, including the 3 instances that were previously unsolved.

1 Introduction

1.1 Register Allocation, Graph Coloring and Integer Linear Programming

Register allocation determines at compile time where each variable will be stored at execution time: either in a register or in memory. Register allocation is often a two-phase approach: spilling of registers to memory, followed by coalescing of registers [3, 9, 16, 6]. Spilling generates loads and stores for live variables[1] that can not be stored in registers. Coalescing allocates unspilled variables to registers in a way that leaves as few as possible `move` instructions (i.e. register copies). Both spilling and coalescing are known to be NP-complete [27, 5].

* This work was supported by Agence Nationale de la Recherche, grant number ANR-05-SSIA-0019.

[1] A variable that may be potentially read before its next write is called a *live* variable.

Classically, register allocation is modeled as a graph coloring problem, where each register is represented by a color, and each variable is represented by a vertex in an interference graph. Two vertex which represent interfering variables are linked by an interference edge, while two vertex which represent variables involved in a *move* instruction are linked with an affinity edge. Given a number k of available registers, finding a register allocation consists in finding (if any) a k-coloring of the interference graph with respect to interference edges. When there is no k-coloring, some variables are spilled to memory (the corresponding vertex of the graph is thus removed). When there is a k-coloring, coalescing consists in choosing a k-coloring that removes most of the `move` instructions, or in other words, that maximizes the sum of weights of affinity edges having both extremities colored with the same color. Figure 1 shows a solved instance of 3-coloring problem with preferences. Dotted edges are preference edges and full edges are interference ones. In this example, the coloration is optimal since the two edges with largest weight have identically colored extremities and it is not possible to satisfy all of the 3 preferences.

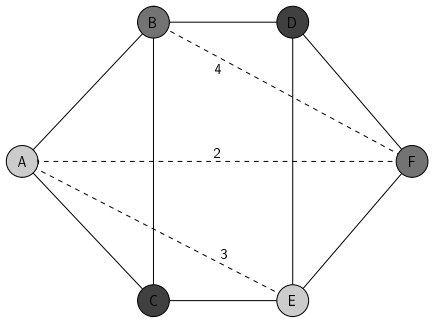

Fig. 1. A solved register allocation instance with three registers.

Determining if a K-coloring exists is generally NP-complete. Hence the register allocation is NP-complete too. Thus, many coloring heuristics have been designed to solve it, that mostly relies on the property that a vertex of low degree (*i.e.* of degree lower than or equal to K-1) can be removed from the graph and be colored after all other nodes. These former heuristics all handle spilling and coalescing simultaneously. The first of them, called aggressive coalescing [10], had the default that a colorable graph could become uncolorable along the algorithm, *i.e.* to introduce useless spills. To avoid this problem, Briggs [8] proposed a conservative heuristics, *i.e.* that preserves colorability. But this heuristics is too conservative and many possible coalescing are not done. Then came a more aggressive but conservative heuristics called iterated register coalescing [14]. If this last one remains the state-of-the-art algorithm, many others heuristics have been designed, in particular some specific to programs in SSA form.

ILP-based approaches have also been applied to register allocation. Appel and George formulate spilling as an integer linear program (ILP) and provide optimal and efficient solutions [3]. Their process to find optimal solutions for spilling requires live-range splitting, an optimization also used by some heuristics that enables a more precise register allocation (e.g. avoiding to spill a variable everywhere). More precisely, their method relies on a specific and extreme form of live-range splitting, the extreme live-range splitting. While the solutions are easily found for spilling in this context, for coalescing they are more elusive. Indeed, live-range splitting generates huge interference graphs (with many `move` instructions) that make the coalescing harder to solve, hard enough to make usual approaches failing[3]. Nevertheless, this result implies that spilling and coalescing can be fully separated and thus reduces register allocation to coalescing under extreme live-range splitting hypothesis. The need for a better algorithm for the optimal coalescing problem gave rise to a benchmark of interference graphs called the optimistic coalescing challenge [2]. Recently, Grund and Hack proposed a cutting-plane algorithm [16] able to solve the whole optimal coalescing challenge (except three instances) but weakened by the large size of the graph.

1.2 Live-range Splitting

Splitting the live-range of a variable v consists in renaming v to different variables having shorter live-ranges than v and adding `move` instructions connecting the variables originating from v. Recent spilling heuristics benefit from live-range splitting: when a variable is spilled because it has a long live-range, splitting this live-range into smaller pieces may avoid to spill v. If the live-range of v is short, it is easier to store v in a register, as the register needs to hold the value of v only during the live-range of v.

There exists many ways of splitting live-ranges (e.g. region splitting, zero cost range splitting, load/store range analysis) [11, 7, 18, 19, 4, 12, 21]. Splitting live-ranges often reduces the interferences with other live-ranges. Thus, most of the splitting heuristics have been successful in improving the spilling phase. The differences between these heuristics stem from the number of splitting points (i.e. program points where live-ranges are split) as well as the sizes of the split live-ranges. These heuristics are sometimes difficult to implement.

The most precise live-range splitting is extreme live-range splitting, where live-ranges are split after each statement. Its main advantage is the preciseness of the generated interference graph. Indeed, a variable is spilled only at the program points where there is no available register for that variable. As in a SSA form, each variable is defined only once. Furthermore, contrary to previous heuristics, extreme live-range splitting is very easy to implement (it does not require any further computation).

Extreme live-range splitting helps in finding optimal and efficient solutions for spilling. But, it generates programs with huge interference graphs. Each renaming of a variable v to v_1 results in adding a vertex in the interference graph for v_1 and, consequently, some edges incident to that vertex.

1.3 Alternative Approaches

Other approaches to solve register allocation have been proposed, mainly designed for just-in-time compilation. Therefore many studies focus on other heuristics such as linear scan [23]. The global quality of this heuristics (including its enhancements as tree scan or second chance binpacking) is quite the same than graph coloring's ones. Nevertheless, graph coloring remains better adapted to a classical compiler, since the compile-time is not the main feature and that it tends to better register allocation in this case.

Very recently, a new approach by puzzle solving has been proposed by Palsberg and Pereira [20]. It has the advantage to be undependant of the architecture. However, practical results are almost of the same quality than coloring and the cost is quite similar too.

Finally, these models should not substitute, at least in their current shape, the state-of-the-art graph coloring model for the case of classical compilers. However, these alternatives are interesting topics of research in the domain and might supplant graph coloring when the possibilities of practical coloring heuristics will seem to be reached.

1.4 Motivation

Our work is based on graph coloring and extreme live-range splitting. It mostly focuses on coalescing and follows Grund and Hack's study [16] but is the first that takes in account properties of extreme live-range splitting. One of our main motivation was that solving an ILP problem is exponential in time and consequently that reducing the size of the ILP formulation can drastically speed up the solution. Rather than reasonning on the ILP model, we have chosen to use graph pecularities to reduce its size, and hence the one of the ILP formulation. The reduction is made of three optimizations.

More precisely, we begin by describing properties of *split interference graphs*, that is, interference graphss under the hypothesis of extreme live-range splitting. We establish that spilling and coalescing remain NP-complete in such graphs even if some specificities can be exploited to design efficient heuristics. Then, using these specificities, we present a very efficient reduction rule for split interference graphs, that only relies on extreme live-range splitting. We show that this reduction is equivalent to find splitting points that could have been useful for spilling but are useless for coalescing. The deletion of a splitting point is what we call *live-range unsplitting*

After that, we propose a decomposition of the graph that allows us to solve register allocation on each part of the decomposition rather than on the full graph, without breaking optimality. The next part presents the last optimization. Then, we discusses theoritical impact of our reduction on the only optimal algorithm for coalescing, the cutting-plane algorithm. The next part is devoted to experimental results on the optimal coalescing challenge. A first result is that our first optimization reduces the size of original graphs (i.e. before extreme live-range splitting) by up to 10, and thus extreme live-range splitting does not make

coalescing harder anymore. A second result is that Grund and Hack's cutting-plane algorithm for optimal coalescing runs 300 times faster when combined with our optimizations, thus enabling to solve all the instances of the optimal coalescing challenge, including the 3 instances that were previously unsolved. We end this report by presenting related work and concluding.

2 Concepts of Graph Theory

As mentioned above, our study mainly relies on graph theory. Therefore we will first summarize some usual concepts about it to be used throughout the study. For the rest of this section we consider a graph $G = (V, E)$ where V is the set of vertices and E the set of edges.

The most important concepts of those defined below are illustrated on figure 2 :

- A *graph coloring* is a function assigning a color to each vertex of the graph. It is said to be *proper* if for each edge the color assign to its extremities is not the same. In the whole report we will abusively use the term "coloring" instead of "proper coloring".
- A graph $G' = (V', E')$ is said to be *induced* (or *vertex-induced*) by V' iff V' is a subset of V and E' is a subset of E such that each edge belonging to E' has both its extremities in V'.
- A *path* between two vertices x and y is a list of edges (without redundancy) such that the end of an edge corresponds to the begin of the next one and that the begin of the first edge is x and the end of the last one is y.
- A *cycle* is a path between beginning and ending with the same vertex.
- A *chord* is an edge connecting two non-adjacent vertices of a cycle.
- A graph is said to be *chordal* if it does not contain any cycle with more than 4 vertices and no chord.
- An *interval graph* is a graph such that each vertex corresponds to an interval and two vertices are neighbours iff their corresponding intervals intersect. Notice that the class of interval graphs is a subclass of the one of chordal graphs since it is impossible to design a chordless cycle using an interval representation.
- A graph such that V can be partitionned into $V1$ and $V2$ and that each edge has exactly one extremity in both parts is said *bipartite*.
- A graph where each vertex is linked to all the others is said to be *complete*.
- A *clique* is a complete induced graph and, given an positive integer k, a k-clique is a clique of k vertices. A clique which cannot be enlarged with an other vertex is said *maximal*, and a *maximum* clique is a clique of maximal cardinality.
- A partition of the vertices of G such that each part is a clique is called a *clique partition*. A *minimum* clique partition is a clique partition of minimal cardinality.
- A *matching* of a graph is a set of non-adjacent edges. A *perfect matching* is a matching reaching each vertex.

- A *connected component* is a maximal (with respect to the inclusion) set of vertices S such that for any two vertices x and y belonging S, there exists a path between x and y.
- A graph is said *connected* if it has only one connected component.
- A *separation set* is a set of vertices whose removal strictly increases the number of connected components of the graph.
- A *2-connected component* is a connected component such that there exists two disjoint paths (*i.e.* which does not share any edge) between each pair of its vertices.

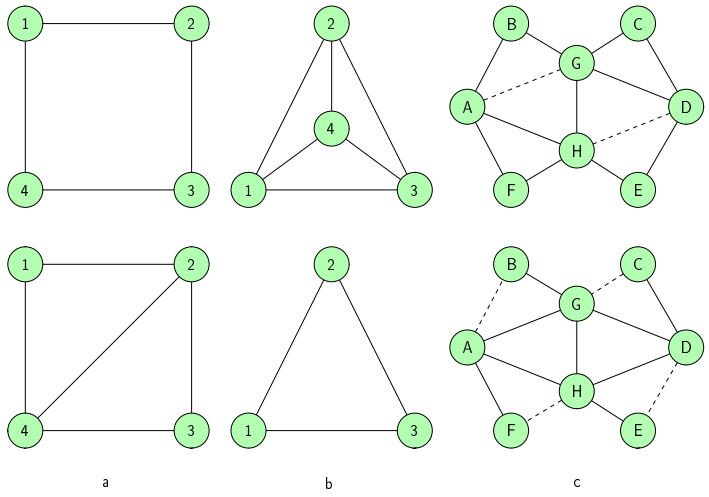

Fig. 2. Concepts of graph theory : (a) a cycle and a chordal graph, (b) a complete graph and a clique, (c) a matching and a perfect matching in dashed edges.

Moreover, since interference graphs contains two kinds of edges, we often use interference "set" (respectively preference "set") as notation. For example a inteference clique is a clique of interference edges, an affinity path is a path of affinity edges, etc.

3 Split interference graphs

This section defines split interference graphs and some graph concepts that are inherent and useful for our optimizations. We also treat of the complexity of spilling and coalescing in split interference graphs.

3.1 Definition

Register allocation is performed on an interference graph. There are two kinds of edges in an interference graph: interference (or conflict) edges and preference (or affinity) edges. Two variables *interfere* if there exists a program point where they are both simultaneously live, and if they may contain different values at this program point. A *preference edge* between two variables represents a `move` instruction between these variables (that should be stored in a same register or at the same memory location). Weights are associated to preference edges, taking into account the frequency of execution of the `move` instructions.

Given a number k of registers, register allocation consists in satisfying all interference edges as well as maximizing the sum of weights of preference edges such that the same color is assigned to both extremities. Satisfying most of the preference edges is the goal of register coalescing.

Interference graphs are built after a liveness analysis [10]. In an interference graph, a variable is described by a unique live-range. Consequently, spilling a variable means spilling it everywhere in the program, even if it could have been spilled only on a shorter live-range. Figure 3 illustrates this problem on a small program consisting of a `switch` statement with 3 branches (see [24] for more details). The program has 3 variables but only 2 variables are updated in each branch of the `switch` statement. Thus, its corresponding interference graph is a 3-clique, that is not 2-colorable, although only 2 registers are needed.

switch(...){

case 0:	case 1:	case 2:
$l_1 : a := \ldots$	$l_4 : a := \ldots$	$l_7 : b := \ldots$
$l_2 : b := \ldots$	$l_5 : c := \ldots$	$l_8 : c := \ldots$
$l_3 : \ldots := a + b$	$l_6 : \ldots := a + c$	$l_9 : \ldots := b + c$

}

Fig. 3. Excerpt of a small program such that its interference graph is a 3-clique.

The usual way to overcome the previous problem is to perform live-range splitting. Extreme live-range splitting splits live-ranges after each statement, and thus generates some renamings that are not useful for coalescing. When v is renamed to v_1 and v_2, if after optimal coalescing v_1 and v_2 share a same color, then the renaming of v_2 is useless: v_2 can be replaced by v_1 while preserving the optimality of coalescing.

Moreover, the number of affinity edges blows up during extreme live-range splitting since there is an affinity edge between any two vertices which represent the same variable in two consecutive statements. Figure 4 shows the split interference graph of a small program given in [1]. In the initial interference graph, each vertex represents a variable of the initial program (the array `mem` is stored in memory); the preference edges correspond to both assignments `d:=c` and `j:=b`.

The bottom of figure 4 is an example of extreme live-range splitting. By lack of space in the figure, only the beginning of the transformed program is shown. The split interference graph is generated using the following process. Every variable that is live and unchanged between program points p_1 and p_2 is copied. A variable that should go dead at p_2 is not copied. All the copies and the statement are executed in parallel. This process is very similar to the one described by Appel and George for the construction of the optimal coalescing challenge. The difference between both processes is minor and has no influence on the properties of split interference graphs that we use.

In other words, each live variable is renamed in parallel to each statement, except if it is killed in this statement. For instance, $k0, k1$ and $k2$ are copies of k. As k is live initially, it is renamed to $k0$ (and so is j). Similarly, g and j are live at the exit of the first statement. Thus, they are renamed after that statement.

Edges corresponding to renamed variables are added in the split interference graph. Preferences edges between renamed variables are also added, as well as interference edges related to renamed variables. For instance, renaming j to $j0$ generates the preference edge $(j, j0)$. In the initial graph, the interference edge (j, k) corresponds to two interference edges in the split interference graph, because there are two program points where j and k interfere.

3.2 Complexity

Spilling Spilling is known to be NP-complete [27] since it is often seen as the search of a maximum K-colorable subgraph of the interference graph. This latter problem is easy for split interference graphs since their structure is very specific. However, spilling is not.

Theorem 1. *After extreme live-range splitting, a statement corresponds to an interference connected component of the split interference graph. Moreover, such a component is an clique, that we call a statement clique.*

Proof. See appendix B. □

Each statement corresponds to a set of vertices which form a clique, which we will call *statement cliques*, since they are all interfering together. Notice also that any statement clique is a connected component of the graph, and reciprocally. As a corrolary, split interference graphs are interval graphs : we can assign to every vertex of an statement clique the same interval such that two different cliques have disjoint intervals. Hence, recent results based on interval or chordal graphs still hold [22] [16].

In our case, the search of a maximum $K-$colorable subgraph is equivalent to the search of such subgraphs in each statement clique. This search is easy : the number of colors needed to color any clique is equal to this clique cardinal. This reasonment also gives a linear time algorithm to find a maximum $K-$colorable subgraph of a split interference graphs.

However, finding a maximum K-colorable subgraph is not enough to provide an optimal spilling because it minimizes the number of variables to spill at

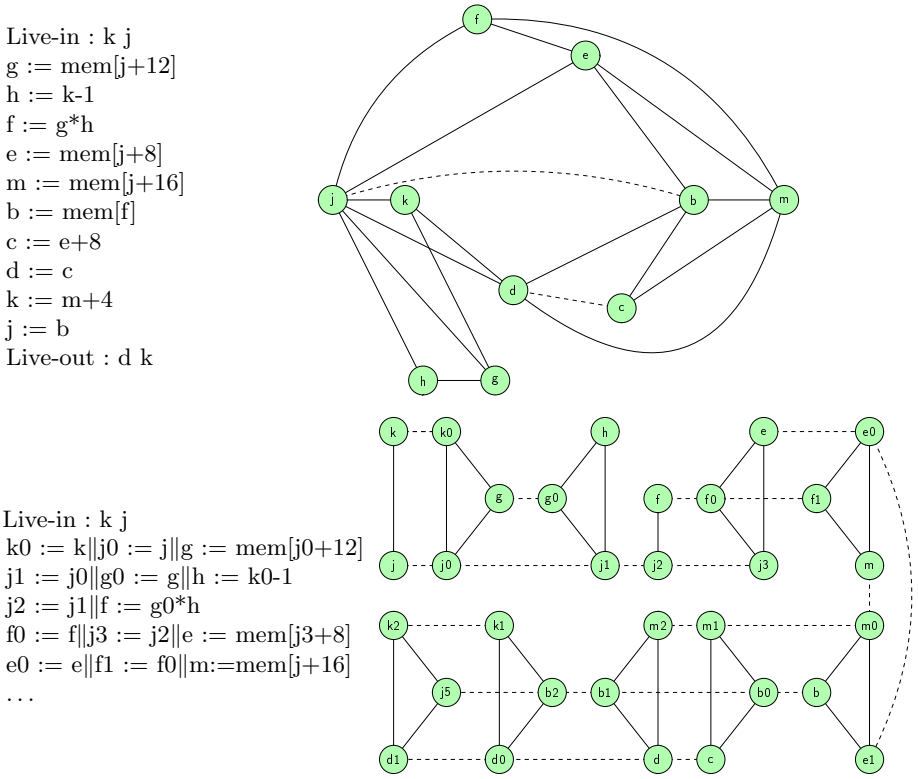

Fig. 4. A small program and its interference graph (top). The same program after extreme live-range splitting and its split interference graph (bottom). The end of the second program is omitted in the figure.

each point instead of the number of spill instructions (*i.e.* of load and store instructions). Indeed, spilling the same variable at many consecutive points only corresponds to two instructions. Thus we can refine the reasonment proposed above taking account of this idea : spilling a variable at a point is equivalent to spill it from its last previous to its next use. Thus we come back to the usual spilling model, and hence to a NP-hard problem.

Coalescing We now deal with coalescing, supposing that spilling has already been done. Hence, each statement clique's size is lower than or equal to K. We will show that the problem remains NP-hard. We first prove it for $K = 2$, then explain how to derive a proof for any $K > 2$. Notice that the proof for $K = 2$ is also a proof that coalescing is NP-hard in bipartite interval graphs.

Theorem 2. *The coalescing is NP-hard, even if each connected component of interference edges is a 2-clique and with two registers and for unit weights.*

Proof. See Appendix A. □

We derive from this result two worthwhile corollaries.

Corollary 1. *The coalescing is NP-hard, even if each connected component of interference edges is a clique of size lower than K and for unit weights.*

Proof. See appendix A. □

Corollary 2. *The coalescing is NP-hard, even if the interference edges form a bipartite interval graph and for unit weights.*

Proof. See appendix A. □

Finally, it also establishes a frontier between NP-hardness and polynomiality, as the following theorem shows.

Theorem 3. *The coalescing is polynomial if the interference edges form a connected bipartite graph and for $K = 2$.*

Proof. See appendix A. □

These complexity results show that both spilling and coalescing do not become easier in split interference graphs. Thus, an optimal solution will remain exponential. However, some properties of split interference graphs can be exploited to develop new heuristics or reduction rules.

3.3 Some Worthwile Properties

The main drawback of extreme live-range splitting is that it generates huge graphs. There are 2 kinds of affinity edges in a split interference graph: edges representing coalescing behaviors, and edges added by variable renaming during live-range splitting. A lot of affinity edges and vertices (as well as some associated edges) corresponding to variable renaming are added in the graph. This section gives 2 properties of these edges and vertices. They are useful for reducing the graphs.

Definition 1. *Parallel clique, dominant and dominated cliques. Let C_1 and and C_2 be 2 maximal interference cliques. C_1 dominates C_2 if there exists an affinity matching M such that :*

1. *M contains only edges having an extremity in C_1 and the other in C_2,*
2. *each vertex of C_2 is reached by M,*
3. *no edge of M has extremities precolored with different colors,*
4. *for each vertex v of C_2, the weight of the edge M that reaches v is greater than or equal to the total weight of all others affinity edges reaching v.*

We also say that C_1 and C_2 are parallel cliques, C_1 is a dominant clique and C_2 is a dominated clique. Moreover, M is called a dominant matching.

Parallel cliques represent splitting points. Figure 5 shows a subgraph of the split interference graph given in figure 4. The condition on the weights may seem to be very restrictive. Actually, it is not. The weight of an affinity edge is often the middle part of the total weight of incident affinity edges reaching its extremities. Indeed, the number of copy statements does not change, except when entering in or exiting from a loop. Hence, many dominations appear. The following property of dominated parallel cliques enables us to remove the splitting points that have created this domination, without worsening the quality of coalescing.

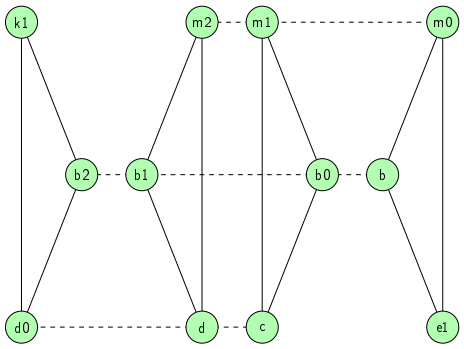

Fig. 5. Some parallel cliques. The cliques {m1, b0, c} and {m2, b1, d} are iso-parallel.

Theorem 4. *If C_1 and C_2 are two parallel cliques such that C_1 dominates C_2, then there exists an optimal coalescing coloring extremities of each edge of the dominant matching with the same color.*

Proof. See appendix B. □

4 Size Reduction Using Parallel Cliques

In this section, we detail the first optimization for simplifying significantly the split interference graphs resulting from spilling, and thus improving optimal coalescing. This optimization do not affect the global quality of coalescing. It consists in removing the splitting points that could have been useful for spilling but that are useless for coalescing. This reduction relies on a subgraph, called *dominated parallel cliques*, representing the splitting points that can be removed from the program. This section details 2 algorithm that respectively find dominated cliques and merge parallel cliques.

Finding Dominated Cliques A reduction rule arises from the theorem 4. However, this rule supposes that finding dominated cliques can be found effi-

ciently.Ss Indeed, it is possible. The algorithm 1 does it in O(km_{C_2}), where m_{C_2} is the number of affinity edges having an extremity in C_2.

Algorithm 1 parallel cliques (C_1,C_2)

Require: Two maximal interference cliques C_1 and C_2
Ensure: A dominant matching M if C_2 is dominated by C_1, NULL otherwise

E := {affinity edges having an extremity in C_1 and the other in C_2}
delete every edge having extremities precolored with different colors from E
for all color c **do**
    if there exist $v_1 \in C_1$ and $v_2 \in C_2$ both colored with c **then**
        if v_1 and v_2 are linked with an affinity edge **then**
            delete from E every affinity edge reaching v_1 or v_2 except (v_1, v_2)
        else
            return NULL
        end if
    end if
end for
for all $v \in C_2$ **do**
    $Pref\_weight(v) = \sum_{x \in Pref\_Neighbors(v)} weight_{(v,x)}$
end for
for all $v \in C_2$ **do**
    for all v' such that $(v, v') \in E$ or $(v', v) \in E$ **do**
        if $weight_{(v,v')} < \frac{1}{2} Pref\_weight(v)$ and v_1 and v_2 are not precolored with the same color **then**
            delete (v, v') from E
        end if
    end for
end for
M := maximum matching included in E
if cardinal(M) = number of vertices of C_2 **then**
    return M
else
    return NULL
end if
add back deleted edges

The first two loops of algorithm 1 compute E, the set of affinity edges that may belong to a dominant matching. The first loop removes the edges that cannot respect precoloring constraints, i.e. that have extremities precolored with different colors or an extremity colored with a color which cannot be affected to the second extremity. The second loop removes every edge such that its weight is not high enough to be dominant. More precisely, an affinity edge can be deleted if its weight is not greater than the half of the total weight of its extremity that belongs to the potential dominated clique. The second part of the algorithm is a search for a maximum affinity matching included in E. This problem is nothing

but the search of a maximum matching in a bipartite graph, which can be solved in polynomial time [26]. Finally, there is only to check if each vertex of C_2 is an extremity of an edge of the matching. That can be done by checking the equality between the cardinal of the matching and the number of vertices of C_2.

Merging Parallel Cliques The main idea of our first optimization is to reduce the size of the split interference graph by removing most of the splitting points. For that purpose, we define a split-block (per analogy to basic blocks) as a set of statement that are not separated with a splitting point. We also define *SB-cliques*, as interference cliques of interference graphs that represent split-blocks. Initially, split-blocks are statement (since we use extreme live-range splitting), and thus SB-cliques are statement cliques. Deleting a splitting point is equivalent to merge the split-blocks they represent. In term of cliques, if two SB-cliques are parallel, then they can be merged (resulting in a new SB-clique), since each pair of vertices linked by an edge of the dominant matching can be coalesced (the splitting point was useless for coalescing). Indeed, there exists an optimal solution that assigns the same color to both vertices. That is what we call live-range unsplitting. This merge leads to a graph where new dominations may appear, as well as vertices with no preference edges. These vertices can be removed from the graph since the interference degree of any vertex is lower than k.

Merging two SB-cliques is equivalent to removing the splitting point that separates the split-blocks they represent and, hence, removing copies that have been created by the deleted splitting point. In other words, merging two split-blocks is equivalent to undo a splitting. Moreover, since merging two cliques yields a new SB-clique, the reduction can be performed until the graph is left unchanged. In order to speed up the process, for each clique i, we first compute the set $N(i)$ of SB-cliques j such that there exists an affinity edge having an extremity in i and the other in j. Then, there is only to find and merge parallel-cliques, and update the graph. This process is iterated as long as there are parallel-cliques in the graph.

Algorithm 2 details our reduction. When applied to the graph of figure 4, it yields an empty graph, meaning that this instance can be optimally solved in polynomial time. Moreover, the solution requires only 3 colors. Iterated register coalescing (a state-of-the-art coalescing heuristics) requires 4 colors when applied to the original interference graph. Actually, if j and b are coalesced then any coloring of the classical interference graph requires at least four colors. Indeed, if jb is the vertex obtained by coalescing j and b, then e, f, m and j, b form a clique of 4 vertices. Thus these 4 vertices must have different colors, and four colors (at least) are needed. It shows, again, that live-range splitting can provide better solutions because variables belonging to split live-ranges may be stored in different registers.

Algorithm 2 graph reduction (G)

Require: A split interference graph G
Ensure: A reduced split interference graph

```
remove vertices that do not belong to any affinity edge
compute statement cliques
for all statement clique i do
    N(i) = {statement cliques linked to i with an affinity edge}
end for
red = 1
while red ≠ 0 do
    red = 0
    for all SB-clique i do
        for all j ∈ N(j) such that |j| ≥ |i| do
            M = dominated parallel cliques(i, j)
            if M ≠ NULL then
                merge each pair of M and compute new weights
                red = red+1
                N(ij) := N(i) ∪ N(j)
                for all k ∈ N(i) ∪ N(j) do
                    N(k) := N(k) ∪ ij\{i, j}
                end for
            end if
        end for
    end for
end while
```

5 Decomposition by Clique Separators

Our second optimization is a decomposition based on clique separators, inspired from [25, 17]. First, we did not know the existence of these works. Thus, all the decomposition process is explained. Contrary to the first one, this one is not specific to split interference graphs. However, we show that properties of split interference graphs can be exploited to improve the algorithm for decomposing.

5.1 Presentation of the Decomposition

The main idea is to use SB-cliques as separation sets. Then, the coalescing will not be solved on the whole graph, but on each component resulting from the decomposition.

This part shows how the problem can be decomposed to provide a faster solution. The idea of this heuristic is to use cliques, that is a structure of the graph for which all the coloring are permutations, as separable sets. Before describing it, we have to define some concepts.

Definition 2 (clique graph). *Given a clique partition of the interference graph, the clique graph is the graph where each vertex is a clique and where edges are the merge of edges between vertices belonging to the cliques.*

Definition 3 (separation clique). *An interference clique is a separation clique iff the removal of it strictly increases the number of connected components of the graph.*

Definition 4 (clique-block). *Given a clique partition, a clique block is a block of the clique graph corresponding to the clique partition.*

The algorithm runs in four phases. Figure 5.1 presents an example of the algorithm's application. These main phases are :

1. The algorithm begins by merging all precolored nodes in K nodes (one for each color), as were doing Appel and George [14]. Notice that this transformation destroy the pecularities of the graph but preserves chordality. Indeed, the split interference graph remains chordal since a merge of clique does not destroy chordality [26].
2. Afterwards we search for separation cliques of the graph. We do not search for all of them, since there can be intersections, but for a set of disjoint separation cliques. This search is realized by finding a clique partition of the graph and searching for separation vertices of the corresponding clique-graph. Notice that the search of separation vertices of the clique graph also provides the clique-blocks [26].
3. Now we construct a last graph, called *separation graph*, which gives a fine representation of the algorithm statements. Its vertices are the clique-blocks of the graph and two vertices are adjacent iff their corresponding clique-blocks intersect.
4. Finally, we use the fact that the separation graph is a tree to color clique-blocks using a DFS order.

This algorithm relies on the two following theorem :

Theorem 5. *The separation graph of a connected graph is a tree.* □

Proof. See appendix C. □

Theorem 6. *The coloring algorithm returns a proper coloring of the graph. Moreover, this coloring is optimal if we can solve the problem optimally for each clique-block.*

Proof. See appendix C. □

5.2 Interest of the Decomposition

In split interference graphs, this decomposition can be done in linear time, rather than in quadratic time. Indeed, the hardest task is to find interference clique separators. This can easily been done in split interference graphs since all interference cliques are disjoint. Hence, to know if a SB-clique is a separator clique, we create a graph where a vertex represents a SB-clique and two vertices are

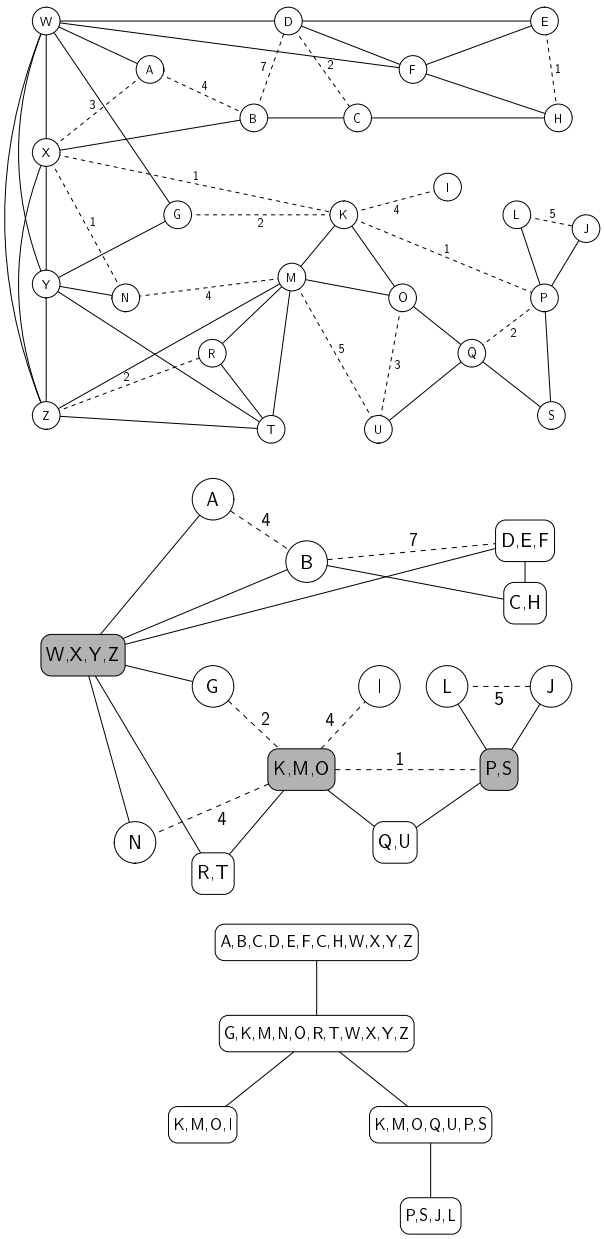

Fig. 6. From top to bottom : a graph G, a clique graph of G and the associated separation graph. Separable-cliques are colored in grey on the clique graph.

adjacent if there exists an edge between the two cliques that these vertices represent. Then, we compute separator vertices of this graph. A separator vertex of this graph corresponds to a separator clique of the split interference graph.

Furthermore, our first optimization based on cliques merging (see algorithm 1) makes cliques more likely to be separators. Indeed, if a union of two cliques is a separable set, then the clique obtained by merging these two cliques is a separation clique.

Finally, another strength of our decomposition is that it gets rid of solutions that are permutations of previous solutions. For a coloring problem, the huge number of such permutations makes this problem hard to deal with. For instance, if D_1 and D_2 are two components of the decomposition that intersect, then coloring D_1 affects part of D_2. Thus, later, when D_2 must be colored, all solutions that are not compatible with the coloring for D_1 can be removed, including many permutations. Since ILP solvers are very sensitive to permutations, deleting some of them may lead to much faster computations.

6 Theoritical Impact of Reductions on the Cutting-plane Algorithm for Coalescing

Even if any algorithm can be used after our optimizations to solve coalescing, this section focuses on the most efficient optimal algorithm, the cutting-plane algorithm of Grund and Hack[16]. More precisely, we discusses of the theoritical impact our approach has on this algorithm.

First, at each iteration where a dominated clique of size s is found, the size of the graph decreases of s vertices, s^2 interference edges and at least s affinity edges. On the ILP formulation of Grund and Hack, it involves a reduction of at least $ks+s$ variables (ks for vertices and at least s for affinity edges) and at least $s^2 + k\frac{s(s-1)}{2} + s$ constraints (s^2 for interference constraints, $k\frac{s(s-1)}{2}$ for affinity constraints and s for coloring constraints). Such a reduction is quite significant, especially when applied many times as the reduction does.

Moreover, the number of cut inequalities generated for the cutting-plane algorithm and the number of variables involved in them decrease with the size of the graph. The more cut inequalities are generated, the more the solver takes time to find efficient ones for each iteration of the simplex algorithm (on which solvers rely). Following the same idea, the more variables are involved in a cut inequality, the more it is difficult to find values for these variables. For instance, a path cut [16] is more efficient if it concerns a path of three affinity edges than if it concerns a path of ten affinity edges. For these reasons, the computation of cut inequalities and the solution are speeded up when using our optimizations.

7 Preprocessing Rules

Finally, our last optimization consists in two preprocessing rules designed to reduce the graph size. The first one is an extension of a well-known reduction

of the coloring problem that is much used for coalescing and the second gives a means of dominating preference. As the second optimization, this one does not require the graph to be a split interference graph.

Lemma 1. *Let G be a graph, K a positive integer and C an preference connected component of G with no intern interference edge. If $\sum_{i \in V(C)} \delta_I(i) < K$ (where $\delta_I(i)$ denotes the interference degree of the vertice i) then C is colored with only one color in every optimal preference K-coloring.*

Proof. See appendix D. □

Lemma 2. *Let G be a graph and (x, y) a preference edge of G. If $w(x, y) \geq \sum_{z \neq y} w(x, z)$ and if $N_I(x) \subseteq N_I(z)$, where $N_I(x)$ denotes the interference neighbourhood of x, then there exists an optimal solution where x and z are colored with the same color.*

Proof. See appendix D. □

These two rules help to reduce the size of the graph. They are not specific to split interference graph and may further be used to design new heuristics, as were used vertices of low degree before [10] [8] [14].

8 Experimental Results

As mentioned previously, we use the optimal coalescing challenge (OCC) as benchmark. OCC is a set of 474 large interference graphs that result from a spilling phase. Our two optimizations are performed on the OCC graphs and generate simplified graphs that are given as input to the ILP formulation (and the associated cutting-plane algorithm) defined by Grund and Hack in [16]. We use the AMPL/CPLEX 9.0 solver (as in [16]) on a PENTIUM 4 2.26Ghz. The first part of this section measures the efficiency of our reduction. Then, the section details respectively optimal and near-optimal solutions. Notice that we only use the two first optimizations for experimental results. Indeed, the last optimization has no sensitive influence on these results.

8.1 Reduction and decomposition

The first measure is the ratio between the sizes of the OCC graph and the biggest subgraph on which coalescing has to be solved (i.e. resulting from our decomposition). We focus on this subgraph because its solution requires almost the whole computation time. These results are detailed in figure 7.

The average reduction is quite significant since the vertex (resp. edge) number is divided by 6 (resp. 4.5). Let us note that the precolored vertices are always kept (because they model the calling conventions of the processor), thus involving a smaller reduction ratio for small graphs. 90% of the reduction arises from the first optimization, i.e. from the deletion of a set of splitting points. Moreover, the reduction runs very fast since it only takes 6 seconds when applied to all the instances of OCC.

Initial number of vertices	Number of instances	Vertex number ratio	Edge number ratio
0-499	292	18,37%	32,59%
500-999	97	13,76%	26,71%
1000-2999	63	12,72%	26,73%
over 3000	22	12,64%	7,64%

Fig. 7. Size reductions for OCC graphs. The vertex (resp. edge) number ratio is the ratio between the number of vertices (resp. edges) of the graph after reduction and the one before reduction.

8.2 Optimal Solutions

We compute optimal solutions for each component of the decomposition using the cutting-plane algorithm of Grund and Hack [16]. For each interference edge, we only compute the path cut corresponding to the shortest path of preference edges linking its extremities. Figure 8 shows a fall of computation times between the solution with and without our optimizations. Indeed, the cutting-plane algorithm finds only 430 optimal solutions within 5 minutes when applied to the OCC graph. When used after our optimization, the cutting-plane algorithm finds 436 instances within one second (including the time spent for our 2 optimizations). On average, the cutting-plane algorithm runs 300 times faster when combined with our 2 optimizations. Only 6 instances are solved in more than one minute, and only 3 of them are solved in more than 150 seconds.

Moreover, we are the first to solve the whole OCC instances optimally. Indeed, in [16] 3 solutions are far too slow and thus their optimality was not certain. We have found a strictly better solution for one instance and proved that the two other solutions are optimal.

8.3 Near-Optimal Solutions

Many problems are solved within a few seconds. We adapt our approach to the other problems in order to avoid combinatorial explosion. Thus, we tune the ILP solver for the 6 instances that take more than one minute to be solved. Numerical results are presented figure 8.3.

A first way of tuning the solver is to give it a time limit. Finding the optimal solution (or a near optimal one) often takes less than 10% of the computation time. The ILP formulation can call the solver a lot, even if the solver has a time limit. Thus, the computation can take more than the time limit. However, it never exceeds this limit too much since there is empirically only one call to the solver that reaches the time limit. In addition, this method can fail if no integer solution is found within the time limit.

A better way to tune the solver is to limit the gap between the expected solution and the optimum. Indeed, the solver can give at any time the gap between the current best solution and the best potential one using a bound

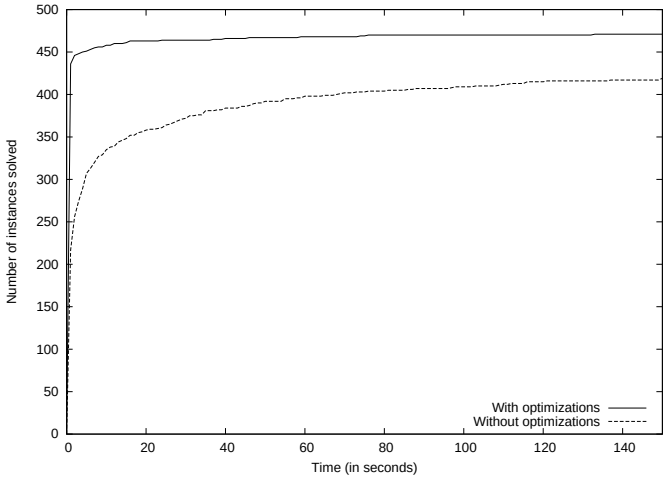

Fig. 8. Number of instances solved within a short time limit : comparison of the cutting-plane algorithm efficiency when using (or not) our optimizations.

of the latter. This method is the opposite of choosing a time limit: it sets the quality of the expected solution and evaluates the time spent to find it, instead of setting a time limit and evaluating the quality of the solution.

Results of figure 8.3 give a flavor of the quality of coalescing on split interference graphs. First, optimistic coalescing (i.e. the best known heuristics for coalescing [21]), is clearly overpassed by limited ILP. Indeed, a short time limit of 20 seconds is already better when it does not fail. Second, a time limit of 30 seconds leads to near-optimal coalescing. The gap between the corresponding solutions and the optimum is never greater than 20%, while the gap for the optimistic coalescing is often about 50%. The failure that occurs for some instances is quite prohibitive but the time limit gives a good idea of the difficulty for solving an instance.

Last, using a gap limit seems very powerful, especially when it is large enough to avoid combinatorial explosion. Here, a limit of 10% leads to solutions of very good quality (under 5% of gap with the optimum) and within a quite short time (less than 2 minutes). Giving a too restricted limit (such as 5% or less) leads to good solutions too but these solutions may be quite slower, as for the instance 387 that goes from 115 to 1187 seconds when the gap goes from 10% to 5%.

Instance	144	304	371	387	390	400
Optimum value	129332	6109	1087	3450	339	1263
Optimistic value	188903	9602	1616	8788	677	2936
20s limited value	129333	no	no	no	417	1388
30s limited value	129333	no	1285	no	365	1263
10% gap limited value	132040	6448	1094	3550	339	1263
5% gap limited value	129342	6273	1094	3450	339	1263
Optimum time	11026	1058	132	29543	102	75
10% gap limited time	15	36	62	115	86	21
5% gap limited time	17	64	62	1187	92	21

Fig. 9. Comparison between different approaches for solving the hardest instances of OCC. *no* means that no solution is computed within the time limit. Times are in seconds.

9 Related Work

Goodwin and Wilken were the first using ILP to solve register allocation [15]. Their model was quite difficult to handle since they tackled the problem with a hardware point of view. Since then, some improvements were added, in particular by Fu and Wilken [13], Appel and George [3], or Grund and Hack [16]. Appel and George optimally solved spilling by ILP and empirically showed that separating spilling and coalescing does not significantly worsen the quality of register allocation. Because of their ILP formulation, they perform extreme live range splitting. For that reason, they were not able to solve coalescing optimally. More recently, Grund and Hack proposed a cutting-plane algorithm to solve coalescing and were the first to solve the optimal coalescing challenge [16]

Our study reuses this previous work and focuses on properties of split interference graphs. Concerning coalescing, our optimizations divide the size of the interference graphs (by ten when measured on the OCC graphs), thus enabling us to find in a faster way more solutions that are optimal and efficient. Moreover, our reduction can explain why optimistic coalescing is quite efficient for split interference graphs. Indeed, our reduction is close to optimistic coalescing: the vertices that are coalesced with this heuristics often correspond to the edges of dominant matchings. Thus, moves corresponding to these edges can be removed while conserving optimality.

When a program is in SSA form, each variable is defined only once. A program modified by extreme live-range splitting can be considered as a generalization of a SSA form. There is a lof of work on register coalescing for programs in SSA forms. This work relies on the chordality of interference graphs resulting from SSA forms and is different from our work.

10 Conclusion

Our main motivation was to improve register coalescing using ILP techniques. Solving an ILP problem is exponential in time and thus reducing the size of the formulation can drastically speed up the solution. Rather than reasoning on the ILP model, we have studied the impact of extreme live-range splitting on register coalescing. We have reused 2 properties of interference graphs resulting from extreme live-range splitting, that are useful for simplifying these graphs. We have defined 3 optimizations for reducing significantly the size of the ILP formulations for coalescing. They are general enough and they can be combined with well-known heuristics for register coalescing.

As said in [16], all the optimizations must go hand in hand to achieve top performance. When our optimizations are combined with a cutting-plane algorithm, we solve the whole optimal coalescing challenge optimally and more efficiently than previously.

Moreover, this work on extreme live-range splitting raises many questions. Indeed, it can be interesting to relax some constraints on split-blocks merging in order to design new heuristics, or to wonder if unsplitting could be done before spilling. Finally, since finding optimal solutions for spilling and coalescing separately is not elusive anymore, one could expect to solve both simultaneously and to evaluate the real gap arising from the separation.

This work is part of an on-going project called CompCert [2], that investigates the formal verification of a realistic C compiler usable for critical embedded software. Future work concern the formal verification of the optimizations described in this report.

References

1. A. W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.
2. Andrew W. Appel and Lal George. Optimal coalescing challenge, 2000. `http://www.cs.princeton.edu/~appel/coalesce`.
3. Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *PLDI'01*, pages 243–253, 2001.
4. Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill code minimization via interference region spilling. In *PLDI '97*, pages 287–295, 1997.
5. Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. In *CGO'07*, mar 2007.
6. Florent Bouchez, Alain Darte, and Fabrice Rastello. Advanced conservative and optimistic coalescing. In *CASES'08*, Atlanta, USA, oct 2008.
7. Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, april 1992.
8. Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428 – 455, 1994.

[2] `http://compcert.inria.fr`

9. Philip Brisk, F.Dabiri, J.Macbeth, and M.Sarrafzadeh. Polynomial time graph coloring register allocation. In *14th Int. Workshop on Logic and Synthesis*, 2005.
10. G J Chaitin. Register allocation and spilling via graph coloring. *Symposium on Compiler Construction*, 17(6):98 – 105, 1982.
11. Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, 1990.
12. Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *CC '98*, pages 174–187, 1998.
13. Changqing Fu and Kent Wilken. A faster optimal register allocator. In *MICRO 35*, pages 245–256, 2002.
14. Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.
15. David Goodwin and Kent Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *Softw. Pract. Exper.*, 26(8):929–965, 1996.
16. Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. In *CC'07*, volume 4420 of *LNCS*, pages 111–125, 2007.
17. Rajiv Gupta, Mary Lou Soffa, and Denise Ombres. Efficient register allocation via coloring using clique separators. *ACM TOPLAS.*, 16(3):370–386, 1994.
18. Priyadarshan Kolte and Mary Jean Harrold. Load/store range analysis for global register allocation. In *PLDI'93*, pages 268–277, 1993.
19. Guei-Yuan Lueh and Thomas Gross. Fusion-based register allocation. *ACM Transactions on Programming Languages and Systems*, 22:2000, 1997.
20. Jan Palsberg and Fernando Pereira. 2008.
21. Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. In *PACT '98*, page 196, 1998.
22. Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *3rd Asian Symp., APLAS 2005, Japan, November, 2005, Proc.*, volume 3780 of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2005.
23. Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999.
24. Vivek Sarkar and Rajkishore Barik. Extended linear scan: An alternate foundation for global register allocation. In *CC'07*, volume 4420 of *LNCS*, pages 141–155, 2007.
25. Robert Endre Tarjan. Decomposition by clique separators. *Discrete Mathematics*, 55(2):221–232, 1985.
26. Douglas B. West. *Introduction to Graph Theory (2nd Edition)*. Prentice Hall, August 2000.
27. Mihalis Yannakakis and Fanica Gavril. The maximum k-colorable subgraph problem for chordal graphs. *Inf. Process. Lett.*, 24(2):133–137, 1987.

A Coalescing's Proofs of Complexity in Split Interference Graphs

Theorem 7. *The coalescing is NP-hard, even if each connected component of interference edges is a 2-clique and with two registers.*

Proof. The proof is a reduction from 3SAT, which is known as NP-hard. Let us first define the 3SAT problem. A clause is defined as a disjunction of litterals. A formula is a conjunction of clauses. The 3SAT problem consists of finding values

of each litteral of a formula where each clause contains at most three litterals, such that the formula is satisfied, i.e. each clause is satisfied.

Here, we consider a 3SAT instance containing n clauses. We now describe how to construct the instance of coalescing. We begin by creating two vertices T and F (to represent $true$ and $false$) linked by an interference edge. The idea is to use the two colors as values $true$ and $false$ and assign to each litteral a color. The second important point is that any satisfied clique must raise the same gain for the objective function, in order to be able to know if all the cliques are satisfied, and that the gain must be strictly lower otherwise. More precisely, any assignment that satisfies a clause must raise the same gain. To do that, we handle each clause in function of its size. Figure 10 presents the construction of the graph corresponding to a clause, in function of its size.

Yet, the graph is almost constructed. The last point is that, for the moment, two vertices (l_i, C) and (l_i, C') are seen as different litterals though they must have the same value. To be sure of that necessity, we defined σ as one plus the sum of all weight of affinity edges. Then, for each litteral l, we create a path of affinity edges, all weighted byσ, containing all occurences l. We further call these specific edges consistancy edges.

Clearly, the size of the graph is polynomial in the size of the SAT problem. Let us now show that solving coalescing on this graph is equivalent to decide if the original SAT problem has a solution.

We first prove that any assignment that satisfy a clause raises a gain of 60 and raises stricly less otherwise. If the clause contains exactly one litteral, then it is obvious. For the other cases, we reason on the number of neighbours of T which are colored with the same color than it. So, consider a clause with two litterals. If no vertex representing these litterals is colored with the same color than T then it raises 0. Otherwise, there are exactly 4 affinity edges that are satisfied and then it raises 60. The hardest case is the one of a clause with 3 litterals. We first remark that the vertex that is linked to F with an affinity edge is of the color of T iff the 3 neighbours of T are colored with the color of T. Indeed, if one is colored with the color of F then the best choice for the neighbour of F becomes the color of F. So, if all the neighbours of T have the same color than T then it raises 60 (the six edges weighted with 10 are satisfied). If the number of vertices like that is one or two, then it also raises 60 (the edge weighted with 22, two edges weighted with 4, and three edges weighted with 10). Finally, if there is no vertex like that then it raises 52 (the edge weighted with 22 and three edges weighted with 10).

We now prove that any optimal solution of this coalescing problem assigns the same color to each vertex corresponding to a litteral. So, suppose that there exists an optimal solution S such that two vertices (l_i, C) and (l_i, C') are not colored with the same color. Thus, along the path of consistancy edges corresponding to l_i, there is at least one edge which extremities have not the same color. Hence, if w designs the total sum of all affinity edges, then the value of S is at most $w - \sigma$. Moreover, it is clear that there exist solutions that assign a unique color

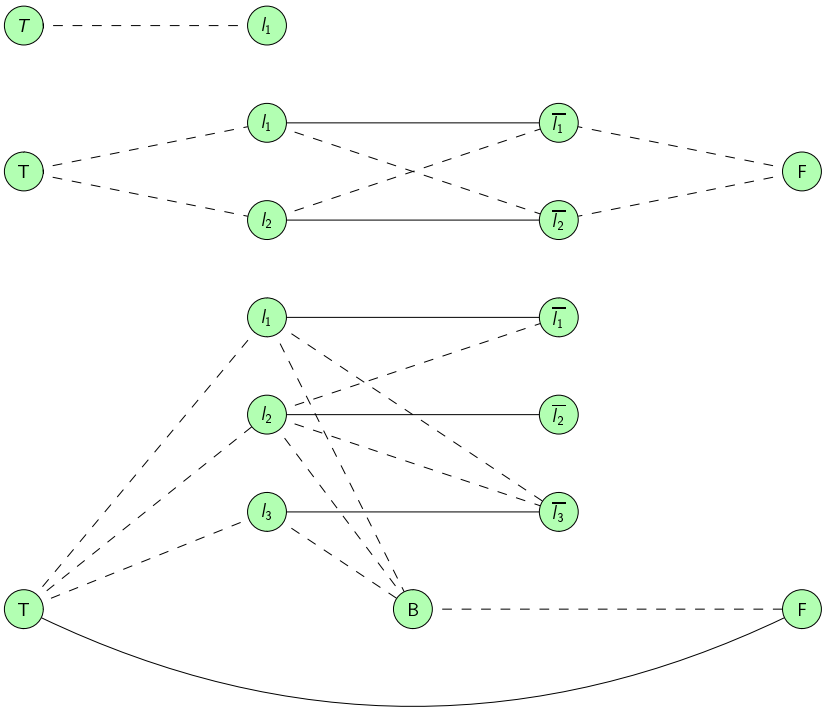

Fig. 10. Detailed construction of the graph for the proof of complexity, for clauses of 1, 2 or 3 litterals (from top to bottom). Weights of the affinity edges are 60 for a single litteral clause and 15 for two litterals clauses. For a clause of three litterals weights are 10 for edges linking B and T to litterals, 4 for edges linking litterals to their opposite and 22 for the edge linking B and F. We do not detail how weights are computed and the graph has been constructed since the reasonment and the proofs are quite long and not really whorthwhile for the result. Notice that these weights make that any clause satisfied, that is such that T and a litteral have the same color, ensures a gain equal to 60.

along all paths of consistancy edges. Any of these solutions has a value which is greater than $w - \sigma + 1$ *by definition of* σ*.*

Now we claim that, if q *designs the number of consistancy edges, then the 3SAT instance has a solution iff the coalescing instances has a solution of value* $q\sigma + 60n$*. Clearly, a solution cannot have a greater value than* $q\sigma + 60n$ *since each subgraph corresponding to a clause can increase the objective value of at most* 60*. Moreover, if we consider a solution of the coalescing problem of value* $q\sigma + 60n$*, we assign a litteral to true if vertices representing it are colored as* T *and to* $false$ *otherwise. By construction, it leads to a solution of the 3SAT instance.*

Finally, the theorem remains correct for unit weights. An edge weighted with w can be replaced by a path of w unit weighted edges. □

Corollary 3. *The coalescing is NP-hard for any K, even if connected component of interference edges is a clique of size lower than K.*

Proof. We prove this result by reduction of the above one. Indeed, we can replace any 2-clique by a clique of size lower than or equal to K, without adding any affinity edge. Suppose we have solution of the problem with two registers. We will show that $q\sigma + 24n$ is an upper bound of the objective function's value and that this bound cannot be reached if more than two colors are used for vertices which have incident affinity edges. To show that $q\sigma + 24n$ is an upper bound we only have to prove that 24 is an upper bound for each subgraph corresponding to a clause, without taking account of consistancy edges, and that no solution that reaches this bound uses strictly more than two colors. The simplest way is to solve it with an ILP solver and to verify the result. Indeed, the proof is not difficult but requires many cases. □

Corollary 4. *The coalescing is NP-hard, even if the interference edges form a bipartite interval graph and for unit weights.*

Proof. The reduction described above constructs an graph where each connected component of interference edges are 2-cliques. Such a graph is clearly either an interval graph and a bipartite graph. Hence, the coalescing is NP-hard in both of these classes. □

Theorem 8. *The coalescing is polynomial if the interference edges form a connected bipartite graph and for $K = 2$. More precisely, for $K = 2$, it can be solved in $O(m + 2^p n)$ where p is the number of interference connected components.*

Proof. The proof relies on a simple algorithm. We divide each connected components in two independant sets. All these decompositions are unique and can be done in $O(m)$. Then, for each connected component only two colorings are possible : one color is assigned to an independant set (two possibilities) and the other color must be assigned to the other independant set. So, there are only 2^p possible coloring of the graph and each of them takes $O(n)$ to be computed. □

B Size Reduction Properties

Theorem 9. *When using extreme live-range splitting, a statement corresponds to an interference connected component of the split interference graph. Moreover, such a component is an interference clique, that we further call statement clique.*

Proof. All the variables between each program points are defined in parallel, thus they all interfere together. It results in a interference clique. Moreover, no other interference concerns these variables since there is an unique statement where they are live. □

Theorem 10. *If C_1 and C_2 are two statement cliques such that there exists a clique of C_1 dominating C_2, then there exists an optimal coalescing coloring extremities of each edge of the dominant matching with the same color.*

Proof. We prove this result by induction on the size of C_2. If there is only one vertex then C_2 has no coloring constraint. Thus coloring this vertex with the same color than its image in the matching leads to an optimal solution. Let us suppose the property correct for a clique of size p. We decompose a clique of size $p+1$ into a clique of size p and an other vertex. The result holds for the clique and the alone vertex. Moreover, there is no constraints between the p vertices and the last one since $p+1$ is lower than or equal to K because spilling has already been done, and there is no other constraints since an statement clique is a connected component. Finally, the optimum is reached since minima of each problems are reached and that the function to optimize is linear. □

C Decomposition Algorithm Soundness Proofs

Theorem 11. *The separation graph of a connected graph is a tree.*

Proof. Let us suppose that the separation graph contains a cycle. Thus, vertices of this cycle are contained into a block. It contradicts the fact that each vertex of this cycle is a block. In addition, if a graph is connected then its separation graph is clearly connected too. Hence, the separation graph of a connected graph is a tree. □

Theorem 12. *The coloring algorithm returns a proper coloring of the graph. Moreover, this coloring is optimal if we can solve the problem optimally for each clique-block.*

Proof. At any moment where we have to color a clique-block C the set of its vertices that are already colored is an interference clique, by definition of clique-blocks, except the first clique-block which contains all the precolored nodes. So, we can solve coalescing on each clique-block. This approach clearly leads to an optimal solution iff the problem is optimally solved on each clique-block. □

D Preprocessing Rules Proofs

Lemma 3. *Let G be a graph and (x, y) a preference edge of G. If $w(x, y) \geq \sum_{z \neq y} w(x, z)$ and if $N_I(x) \subseteq N_I(z)$, where $N_I(x)$ denotes the interference neighbourhood of x, then there exists an optimal solution where x and z are colored with the same color.*

Proof. Let Col be an optimal coloring of G. We use *val_edge* and *val* respectively to denote the value of a preference edge in a coloring (i.e. its weight if its extremities do not have the same color and 0 otherwise) and the value of a coloring (i.e. the sum of its edges' values). If x and z have the same color then

the result holds. Otherwise, we consider Col' the coloring such that x is colored with the color of z in Col and each other vertex has the same color in Col and in Col'. So we have

$$val(Col) = \sum_{(a,b)\in A, a\neq x, b\neq x} val\_edge(Col,(a,b)) + \sum_{(x,z)\in A} val\_edge(Col,(x,z))$$

and

$$val(Col') = \sum_{(a,b)\in A, a\neq x, b\neq x} val\_edge(Col',(a,b)) + \sum_{(x,z)\in A} val\_edge(Col',(x,z))$$

Hence

$$val(Col) - val(Col') = \sum_{(x,z)\in A} val\_edge(Col,(x,z)) - \sum_{(x,z)\in A} val\_edge(Col',(x,z))$$

and then, using the hypothesis we obtain

$$val(Col) - val(Col') \leq \sum_{(x,z)\in A} val\_edge(Col,(x,z)) - w(x,y) \leq 0$$

Finally, we have $val(Col) \leq val(Col')$. Moreover, Col' is a proper coloring since Col is a proper coloring and $N_I(x) \subseteq N_I(z)$. □