Live-range Unsplitting for Faster Optimal Coalescing

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 paper focuses on optimal coalescing in the context of extreme live-range splitting. It presents 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. Any coalescing technique can be applied after these 2 optimizations. Our 2 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 2 optimizations. Moreover, we provide all the solutions of the optimal coalescing challenge, including the 3 instances that were previously unsolved.

1 Introduction

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, 15, 7]. 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 [23, 6].

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. Given a number k of available registers, register allocation consists in finding (if any) a k-coloring of the interference graph. When

[⋆] 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.

there is no k-coloring, some variables are spilled to memory. When there is a k-coloring, coalescing consists in choosing a k-coloring that removes most of the `move` instructions.

ILP-based approaches have 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 that enables a more precise register allocation (e.g. avoiding to spill a variable everywhere). 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.

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, 8, 17, 18, 4, 12, 19]. 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. Thus, interference graphs become so huge that coalescing heuristics often fail on big graphs. 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 have formulated coalescing as an ILP [15]. They introduce a cutting-plane algorithm that reduces the search space (i.e. the space of potential solutions). Thus, their ILP formulation needs less time to compute optimal solutions. As a result, they provide the first optimal and efficient solutions of the optimal coalescing challenge. These solutions are 50% better than the solutions computed by the best coalescing heuristics.

Grund and Hack conclude in [15] that their cutting-plane algorithm (and more generally their reduction techniques) fails when applied to the biggest graphs of the optimal coalescing challenge. Because of extreme live-range splitting (that was required for optimal spilling), optimal solutions for coalescing can not be computed on these graphs. Owing to the extreme amount of copies, coalescing is hard to solve optimally. In this paper, we answer to the last question raised in [15]. We study the impact of extreme live-range splitting on coalescing and provide two optimizations for reducing interference graphs before coalescing.

The first optimization identifies most of the splitting points that are not useful for coalescing, and updates the graph consequently. After that, a second optimization finds clique separators in the updated graph and thus decomposes it into several subgraphs. Coalescing on each subgraph is then solved separately. As the subgraphs are much smaller than the original graph, ILP solutions are much easier and faster to find. Moreover, both optimizations do not break the optimality of coalescing.

The remainder of this paper is organized as follows. Section 2 defines *split interference graphs*, that are the interference graphs resulting from extreme live-range splitting. Properties of these graphs are also given and reused in the next section. Section 3 details the main contribution of this paper. We present our two optimizations: a graph reduction followed by a graph decomposition.

Section 4 presents 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. Related work is discussed in section 5, followed by concluding remarks in section 6.

2 Foundation

This section defines split interference graphs as well as some concepts from graph theory.

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 (but they are ommitted in this paper).

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 on a shorter live-range. Figure 1 illustrates this problem on a small program consisting of a `switch` statement with 3 branches (see [20] 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. 1. 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.

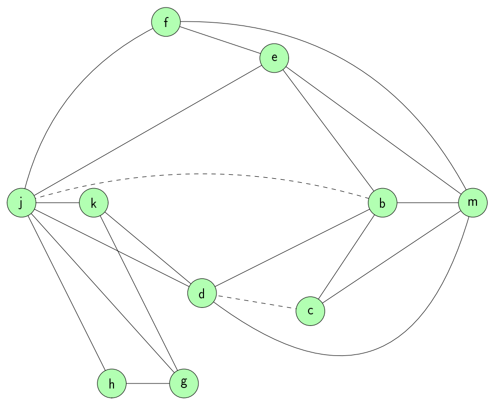
Moreover, the number of affinity edges blows up during extreme live-range splitting since there is an affinity edge between any two vertices which represent the same variable in two consecutive statements. Figure 2 shows the split interference graph of a small program given in [1]. In the initial interference 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 2 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 $jj0$. In the initial graph, the interference edge jk corresponds to two interference edges in the split interference graph, because there are two program points where j and k interfere.

```
Live-in : k j
g := mem[j+12]
h := k-1
f := g*h
e := mem[j+8]
m := mem[j+16]
b := mem[f]
c := e+8
d := c
k := m+4
j := b
Live-out : d k
```



Live-in : k j
$k0 := k \parallel j0 := j \parallel g := mem[j0 +$
$j1 := j0 \parallel g0 := g \parallel h := k0 - 1$
$j2 := j1 \parallel f := g0 * h$
$f0 := f \parallel j3 := j2 \parallel e := mem[j3 -$
$e0 := e \parallel f1 := f0 \parallel m := mem[j +$
$\dots$

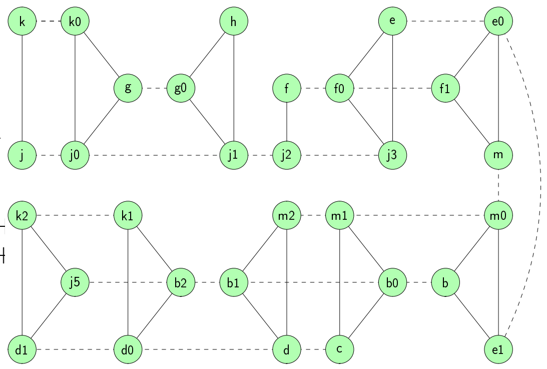


Fig. 2. A small program and its interference graph (top). The same program after extreme live-range splitting and its split interference graph (bottom). The end of the second program is omitted in the figure.

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. The proofs of these properties are omitted in this paper. They are given in a full technical report [5].

Clique. A *clique* of a graph G is a subgraph of G having an edge between every pair of vertices. A clique is said to be *maximal* if there is no clique containing it.

Interference connected component. Given a graph, an interference connected component is a maximal set of vertices such that there exists a path (of interference edges) between any pair of its vertices.

Interference clique. An *interference clique* is a clique containing only interference edges.

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 interference clique, that we call a statement clique.*

Matching. A *matching* is a set of non-adjacent edges. A matching is *maximum* if there is no matching having a strictly higher cardinal.

Affinity matching. An *affinity matching* is a matching consisting only of affinity edges.

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.*

Figure 3 shows a subgraph of he split interference graph given in figure 2. 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.

Theorem 2. *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.*

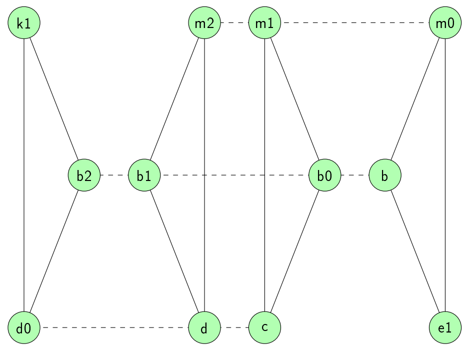


Fig. 3. Some parallel cliques. The cliques {m1, b0, c} and {m2, b1, d} are iso-parallel.

3 Simplifying split interference graphs

In this section, we detail two optimizations for simplifying significantly the split interference graphs resulting from spilling, and thus improving optimal coalescing. These optimizations do not affect the global quality of coalescing. The first optimization reduces split interference graphs. The second optimization is performed after the first one. It uses clique separators to decompose the graph into several small subgraphs.

3.1 Size reduction using parallel cliques

The first optimization removes 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 2. Furthermore, dominated cliques can be found in polynomial time. The algorithm 1 does it in $\mathrm{O}(km_{C_2})$, where m_{C_2} is the number of affinity edges having an extremity in C_2.

Induced graph. Let G be a graph and S a set of its vertices. The subgraph *induced* by S is the graph such that its set of vertices is S and its edges are the edges of G having both extremities in S.

Bipartite graph. A graph G is bipartite if there exists a partition (V_1, V_2) of its vertices such that every edge has an extremity in V_1 and the other in V_2.

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

Algorithm 1 parallel cliques (C_1,C_2)

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

```
1: $E := \{$affinity edges having an extremity in $C_1$ and the other in $C_2\}$
2: delete every edge having extremities precolored with different colors from $E$
3: for all color $c$ do
4:     if there exist $v_1 \in C_1$ and $v_2 \in C_2$ both colored with $c$ then
5:         if $v_1$ and $v_2$ are linked with an affinity edge then
6:             delete from $E$ every affinity edge reaching $v_1$ or $v_2$ except $(v_1, v_2)$
7:         else
8:             return NULL
9:         end if
10:    end if
11: end for
12: for all $v \in C_2$ do
13:    $Pref\_weight(v) = \sum_{x \in Pref\_Neighbors(v)} weight_{(v,x)}$
14: end for
15: for all $v \in C_2$ do
16:    for all $v'$ such that $(v, v') \in E$ or $(v', v) \in E$ do
17:        if $weight_{(v,v')} < \frac{1}{2} Pref\_weight(v)$ and $v_1$ and $v_2$ are not precolored with
           the same color then
18:            delete $(v, v')$ from $E$
19:        end if
20:    end for
21: end for
22: $M :=$ maximum matching included in $E$
23: if cardinal(M) = number of vertices of $C_2$ then
24:    return $M$
25: else
26:    return NULL
27: end if
28: add back deleted edges
```

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 [22]. 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 *SB-cliques* (for split-blocks cliques by analogy to basic blocks), as cliques of the interference graph. Initially, all SB-cliques are statement 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. Indeed, there exists an optimal solution that assigns the same color to both vertices. 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 2, 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.

3.2 Decomposition by clique separators

Our second optimization is a decomposition based on clique separators, inspired from [21, 16]. Its main idea is to use SB-cliques as separation sets. A *separation set* is a set of vertices whose removal partitions a connected component into several ones. Then, the coalescing will not be solved on the whole graph, but on each component resulting from 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 adjacent if there exists an edge between the two cliques that these vertices

Algorithm 2 graph reduction (G)

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

```
 1: remove vertices that do not belong to any affinity edge
 2: compute statement cliques
 3: for all SB-clique $i$ do
 4:     N(i) = {statement cliques linked to $i$ with an affinity edge}
 5: end for
 6: $red = 1$
 7: while $red \neq 0$ do
 8:     red = 0
 9:     for all statement clique $i$ do
10:         for all $j \in N(j)$ such that $|j| \geq |i|$ do
11:             $M = dominated\ parallel\ cliques(i, j)$
12:             if $M \neq NULL$ then
13:                 merge each pair of $M$ and compute new weights
14:                 red = red+1
15:                 $N(ij) := N(i) \cup N(j)$
16:                 for all $k \in N(i) \cup N(j)$ do
17:                     $N(k) := N(k) \cup ij \backslash \{i, j\}$
18:                 end for
19:             end if
20:         end for
21:     end for
22: end while
```

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.

3.3 Impact 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[15]. 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 [15] 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.

4 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 [15]. We use the AMPL/CPLEX 9.0 solver (as in [15]) 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.

4.1 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 4.

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. 4. 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.

4.2 Optimal solutions

We compute optimal solutions for each component of the decomposition using the cutting-plane algorithm of Grund and Hack [15]. For each interference edge, we only compute the path cut corresponding to the shortest path of preference edges linking its extremities. Figure 5 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.

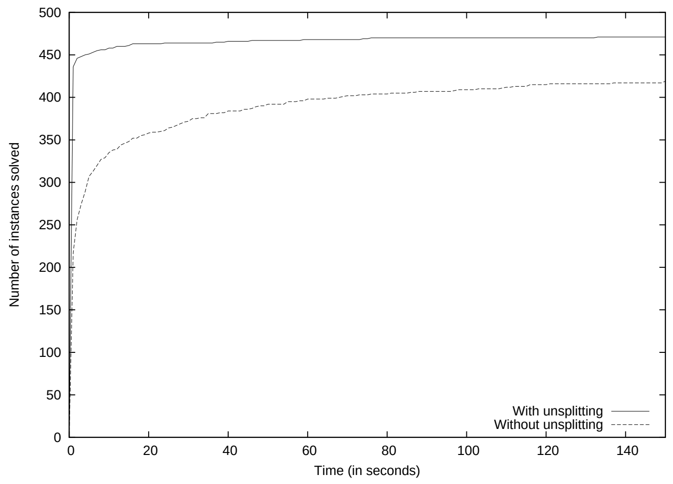


Fig. 5. Number of instances solved within a short time limit : comparison of the cutting-plane algorithm efficiency when using (or not) our optimizations.

Moreover, we are the first to solve the whole OCC instances optimally. Indeed, in [15] 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.

4.3 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 4.3.

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 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.

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. 6. 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.

Results of figure 4.3 give a flavor of the quality of coalescing on split interference graphs. First, optimistic coalescing (i.e. the best known heuristics for coalescing [19]), 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%.

5 Related work

Goodwin and Wilken were the first using ILP to solve register allocation [14]. 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 [15]. 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 [15]

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.

6 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 2 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 [15], all the optimizations must go hand in hand to achieve top performance. Whe 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 paper.

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. Sandrine Blazy and Benot Robillard. Live-range unsplitting for faster optimal coalescing (extended version). Technical report, CEDRIC, oct 2008.
6. Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. In *CGO'07*, mar 2007.
7. Florent Bouchez, Alain Darte, and Fabrice Rastello. Advanced conservative and optimistic coalescing. In *CASES'08*, Atlanta, USA, oct 2008.
8. Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, april 1992.
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.

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

13. Changqing Fu and Kent Wilken. A faster optimal register allocator. In *MICRO 35*, pages 245–256, 2002.
14. 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.
15. Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. In *CC'07*, volume 4420 of *LNCS*, pages 111–125, 2007.
16. Rajiv Gupta, Mary Lou Soffa, and Denise Ombres. Efficient register allocation via coloring using clique separators. *ACM TOPLAS.*, 16(3):370–386, 1994.
17. Priyadarshan Kolte and Mary Jean Harrold. Load/store range analysis for global register allocation. In *PLDI'93*, pages 268–277, 1993.
18. Guei-Yuan Lueh and Thomas Gross. Fusion-based register allocation. *ACM Transactions on Programming Languages and Systems*, 22:2000, 1997.
19. Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. In *PACT '98*, page 196, 1998.
20. 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.
21. Robert Endre Tarjan. Decomposition by clique separators. *Discrete Mathematics*, 55(2):221–232, 1985.
22. Douglas B. West. *Introduction to Graph Theory (2nd Edition).* Prentice Hall, August 2000.
23. Mihalis Yannakakis and Fanica Gavril. The maximum k-colorable subgraph problem for chordal graphs. *Inf. Process. Lett.*, 24(2):133–137, 1987.