Register allocation : what does Chaitin's NP-completeness proof really prove ?

Florent Bouchez, Alain Darte, Fabrice Rastello

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

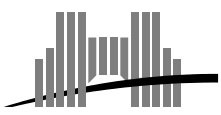
Register Allocation: What does Chaitin's NP-completeness Proof Really Prove?

Florent Bouchez
Alain Darte
Fabrice Rastello

March 2006

Research Report N° 2006-13

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`



Register Allocation: What does Chaitin's NP-completeness Proof Really Prove?

Florent Bouchez, Alain Darte, and Fabrice Rastello

March 2006

Abstract

Register allocation is one of the most studied problem in compilation. It is considered as an NP-complete problem since Chaitin, in 1981, showed that assigning temporary variables to k machine registers amounts to color, with k colors, the interference graph associated to variables and that this graph can be arbitrary, thereby proving the NP-completeness of the problem. However, this original proof does not really show where the complexity comes from. Recently, the re-discovery that interference graphs of SSA programs can be colored in polynomial time raised the question: Can we exploit SSA to perform register allocation in polynomial time, without contradicting Chaitin's NP-completeness result? To address such a question, we revisit Chaitin's proof to better identity the interactions between spilling (load/store insertion), coalescing/splitting (moves between registers), critical edges (a property of the control-flow graph), and coloring (assignment to registers). In particular, we show when it is easy to decide if temporary variables can be assigned to k registers or if some spilling is necessary. The real complexity comes from critical edges, spilling, and coalescing, which are addressed in our other reports.

Keywords: Register allocation, SSA form, chordal graph, NP-completeness, critical edge.

Résumé

L'allocation de registres est l'un des problèmes les plus étudiés en compilation. On le considère en général NP-complet depuis que Chaitin, en 1981, a montré qu'affecter des variables temporaires à k registres physiques revient à colorier avec k couleurs le graphe d'interférences associé aux variables et que ce graphe peut être quelconque. En revanche, cette démonstration ne révèle pas vraiment d'où vient la complexité. Récemment, la re-découverte que les graphes d'interférence des programmes SSA peuvent être coloriés en temps polynomial a conduit à la question : peut-on exploiter la forme SSA pour faire de l'allocation de registres en temps polynomial sans contredire la preuve de Chaitin ? Pour répondre à ce genre de questions, nous revisitons la démonstration de Chaitin pour mieux identifier les interactions entre le "spilling" (insertion de *store*/*load*), le "coalescing"/"splitting" (*moves* entre registres), la présence d'arcs critiques (une propriété du graphe de flot de contrôle) et le coloriage proprement dit (affectation aux registres). En particulier, nous montrons quand il est facile de décider si des variables temporaires peuvent être affectées à k registres ou si du "spilling" est nécessaire. La vraie complexité du problème d'allocation de registres provient de la présence d'arcs critiques, du "spilling" et du "coalescing", problèmes que nous considérons dans nos autres rapports.

Mots-clés: Allocation de registres, forme SSA, graphe triangulé, NP-complétude, arc critique.

Register Allocation: What does Chaitin's NP-completeness Proof Really Prove?

Florent Bouchez, Alain Darte, and Fabrice Rastello

13th March 2006

Abstract

Register allocation is one of the most studied problem in compilation. It is considered as an NP-complete problem since Chaitin, in 1981, showed that assigning temporary variables to k machine registers amounts to color, with k colors, the interference graph associated to variables and that this graph can be arbitrary, thereby proving the NP-completeness of the problem. However, this original proof does not really show where the complexity comes from. Recently, the re-discovery that interference graphs of SSA programs can be colored in polynomial time raised the question: Can we exploit SSA to perform register allocation in polynomial time, without contradicting Chaitin's NP-completeness result? To address such a question, we revisit Chaitin's proof to better identity the interactions between spilling (load/store insertion), coalescing/splitting (moves between registers), critical edges (a property of the control-flow graph), and coloring (assignment to registers). In particular, we show when it is easy to decide if temporary variables can be assigned to k registers or if some spilling is necessary. The real complexity comes from critical edges, spilling, and coalescing, which are addressed in our other reports.

1 Introduction

Register allocation is one of the most studied problem in compilation. Its goal is to find a way to map the temporary variables used in a program into physical memory locations (either main memory or machine registers). Accessing a register is much faster than accessing memory, therefore one tries to use registers as much as possible. Of course, this is not always possible, thus some variables must be transfer ("spilled") to and from memory. This has a cost, the cost of load and store operations, that should be avoided as much as possible.

Classical approaches are based on fast graph coloring algorithms (sometimes combined with techniques dedicated to basic blocks). A widely-used algorithm is iterated register coalescing proposed by Appel and George [12], a modified version of previous developments by Chaitin [6, 5], and Briggs et al. [3]. In these heuristics, *spilling*, *coalescing* (removing register-to-register moves), and *coloring* (assigning a variable to a register) are done in the same framework. Priorities among these transformations are done implicitly with cost functions. *Splitting* (adding register-to-register moves) can also be integrated in this framework.

Such techniques are well-established and used in any optimizing compiler. However, there are at least four reasons to revisit these approaches.

1. Today's processors are now much faster than in the past, especially faster than when Chaitin developed his first heuristic (in 1981). Some algorithms not considered in the past, because they were too time-consuming, can be good candidates today.

2. For some critical applications, especially in embedded computing, industrial compilers are ready to accept longer compilation times if the final code gets improved.

3. The increasing cost of a memory access compared to a register access suggests that it is maybe better now to focus on heuristics that give more importance to spilling cost minimization, possibly at the price of additional register-to-register moves, in other words, heuristics that consider the trade-off spilling/coalescing as unbalanced.

4. There are many pitfalls and folk theorems, concerning the complexity of the register allocation problem, that need to be clarified.

This last point is particularly interesting to note. In 1981, Chaitin [6] showed that allocating variables of a program to k registers amounts to color with k colors the corresponding interference graph (two variables interfere if they are simultaneously live). As he was able to produce a code corresponding to an arbitrary interference graph and because graph coloring is NP-complete [11, Problem GT4], heuristics have been used for everything: spilling, coalescing, splitting, coloring, etc. Except in a few papers where authors are more careful, the previous argument (register allocation *is* graph coloring, therefore it is NP-complete) is one of the first sentences of any paper on register allocation. This way of presenting Chaitin's proof can make the reader (researcher or student) believe more than what this proof actually shows. In particular, it is in the common belief that, when no instruction scheduling is allowed, deciding if some spilling is necessary to allocate variables to k registers is NP-complete. This is *not* what Chaitin proves. We will even show that this particular problem is *not* NP-complete except for a few particular cases (we will make clear which one), which is maybe a folk theorem too. Actually, going from register allocation to graph coloring is just a way of modeling the problem, but it is not an equivalence. In particular, this model does not take into account the fact that a variable can be moved from a register to another one (splitting), of course at some cost, but only the cost of a move instruction (which is often better than a spill).

Until very recently, only a few authors tried to address the complexity of register allocation in more details. Maybe the most interesting complexity results are those of Liberatore et al. [16, 9], who analyze the reasons why optimal spilling is hard for local register allocation (i.e., register allocation for basic blocks). In brief, for basic blocks, the coloring phase is of course easy (the interference graph is an interval graph) but deciding which variable to spill and where is difficult (when stores and loads have nonzero costs). We completed this study for various models of spill cost in [2].

Today, most compilers go through an intermediate code representation, the (strict) SSA form (static single assignment) [7], which makes many code optimizations simpler. In such a code, each variable is defined textually only once and is alive only along the dominance tree associated to the control-fbw graph. Some so-called ϕ functions are used to transfer values along the control fbw not covered by the dominance tree. The consequence is that the interference graph of such a code is, again, not arbitrary: it is a chordal graph, therefore easy to color. Furthermore, it can be colored with k colors if and only Maxlive $\leq k$ where Maxlive is the maximal number of variables simultaneously live. What does this property imply? One can imagine to decompose the register allocation problem into two phases. The first phase (also

called allocation in [16]) decides what values are spilled and where, so as to get to a code where Maxlive $\leq k$. A second phase of coloring (called register assignment in [16]) maps variables to registers, possibly removing (i.e., coalescing) or introducing (i.e., splitting) move instructions (also called shuffle code in [17]). Considering that loads and stores are more expensive than moves, such an approach is worth exploring. This is the approach advocated by Appel and George [1] and, more recently, in [4, 2, 14].

The fact that interference graphs of strict SSA programs are chordal is well-known (if one makes the connection between graph theory and SSA form). Indeed, a theorem of Walter (1972), Gavril (1974), and Buneman (1974) (see [13, Theorem 4.8]) shows that an interference graph is chordal if and only if it is the interference graph of a family of subtrees (here the live ranges of variables) of a tree (here the dominance tree). Furthermore, maximal cliques correspond to program points. We re-discovered this property in 2002 when teaching to students that register allocation is indeed in general NP-complete but certainly not *just* because graph coloring is NP-complete. Independently, Brisk et al. [4], Perreira and Palsberg [18], and Hack et al. [14] made the same observation. A direct proof of the chordality property for strict SSA programs can be given, see for example [2, 14].

Recent work has been done on how to go out of SSA [15, 21, 20] and remove ϕ functions, which are not machine code. How to avoid permutations of colors at ϕ points is also addressed in [14]. These work combined with the idea of spilling before coloring so that Maxlive $\leq k$ has led Perreira and Palsberg [19] to wonder where the NP-completeness of Chaitin's proof (apparently) disappeared: "Can we do polynomial-time register allocation by first transforming the program to SSA form, then doing linear-time register allocation for the SSA form, and finally doing SSA elimination while maintaining the mapping from temporaries to registers?" (all this when Maxlive $\leq k$ of course, otherwise some spilling needs to be done). They show that the answer is no, the problem is NP-complete.

The NP-completeness proof of Perreira and Palsberg is interesting, but it does not completely answer the question. It shows that if we choose the splitting points *a priori* (in particular as ϕ points), then it is NP-complete to choose the right colors. However, there is no reason to fix these particular split points. We show in this paper that, when we can choose the split points, when we are free to add program blocks so as to remove critical edges (as this is often done), when Maxlive $\leq k$, then it is in general easy to decide if and how we can assign variables to registers without spilling. More generally, the goal of this paper is to discuss the implications of Chaitin's proof (and what it does not imply) concerning the interactions between spilling, splitting, coalescing, critical edges, and coloring.

In Section 2, we first reproduce Chaitin's proof and analyze it more carefully. The proof shows that when the control-flow graph has critical edges, which we are not allowed to remove with additional blocks, then it is NP-complete to decide whether k registers are enough, even if splitting variables is allowed. In Section 3, we address the same question as Perreira and Palsberg in [19]: we show that Chaitin's proof can easily be extended to show that, when the graph has no critical edge but if splitting points are fixed (at entry and exit of basic blocks), the problem remains NP-complete. In Section 4, we show, again with a slight variation of Chaitin's proof, that even if we can split variables wherever we want, the problem remains NP-complete, *but only when there are machine instructions that can create two new variables at a time*. However, in this case, it is more likely that the architecture can also perform register swap and then k registers are enough if and only if Maxlive $\leq k$. Finally, we show that it is also easy to decide if k registers are enough when only one variable can be created at a given

time (as in traditional assembly code representation). Therefore, this study shows that the NP-completeness of register allocation is *not* due to the coloring phase (as a misinterpretation of Chaitin's proof may suggest), but is due to the presence of critical edges or not, and to the optimization of spilling costs and coalescing costs.

2 Direct consequences of Chaitin's proof

Let us look at Chaitin's NP-completeness proof again. The proof is by reduction from graph coloring [11, Problem GT4]: Given an undirected graph $G = (V, E)$ and an integer k, can we color the graph with k colors, i.e., can we define, for each vertex $v \in V$, a color $c(v)$ in $\{1, \ldots, k\}$ such that $c(v) \neq c(u)$ for each edge $(u, v) \in E$? The problem is NP-complete if G is arbitrary, even for a fixed $k \geq 3$.

For the reduction, Chaitin creates a program with $|V|+1$ variables, one for each vertex $u \in V$ and an additional variable x, and the following structure:

- For each $(u, v) \in E$, there is a block $B_{u,v}$ that defines u, v, and x.
- For each $u \in V$, there is a block B_u that reads u and x, and returns a new value.
- Each block $B_{u,v}$ is a direct predecessor in the control-fbw graph of the blocks B_u and B_v.
- An entry block switches to all blocks $B_{u,v}$.

Figure 1 shows the program associated to a cycle of length 4, with edges (a, b), (a, c), (b, d), and (c, d). This is also the example used in [19].

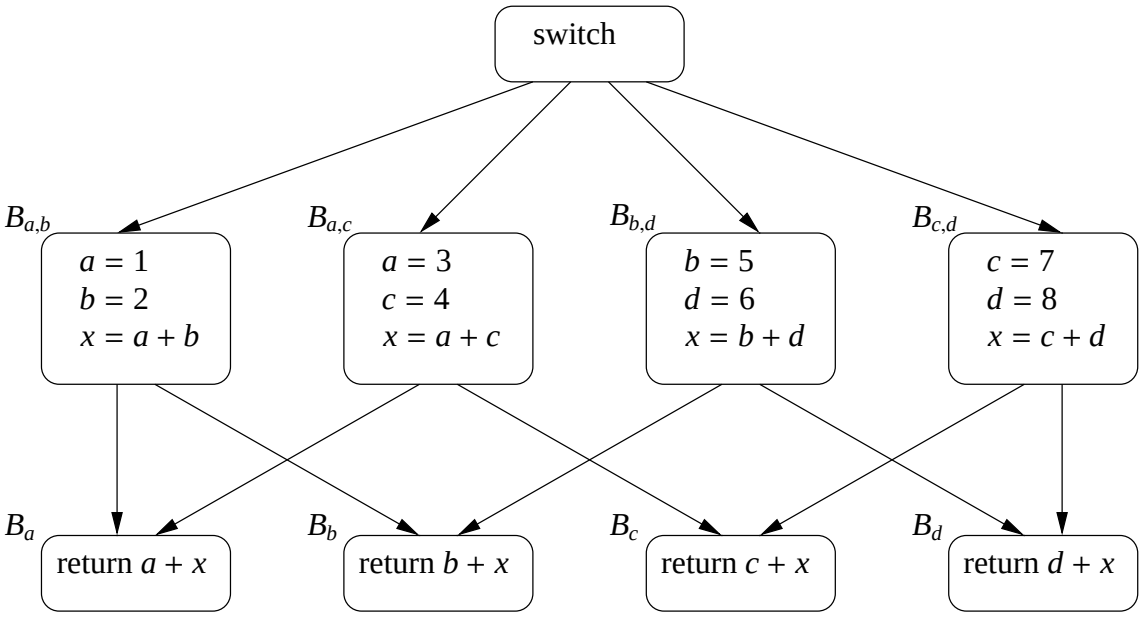


Figure 1: The program associated to a cycle of length 4.

It is clear that the interference graph associated to such a program is the graph G plus a vertex for variable x with an edge (u, x) for each $u \in V$ (thus this new vertex must use an extra color). If one interprets a register as a color then G is k-colorable if and only if each variable

can be assigned to a unique register for a total of at most $k + 1$ registers. This is what Chaitin proved, nothing less, nothing more: for such programs, deciding if one can assign the variables, *this way*, to $k \geq 4$ registers is thus NP-complete.

Chaitin's proof, at least in its original interpretation, does not address the possibility of splitting the live range of a variable (set of program points where the variable is live [1]). Each vertex of the interference graph represents the complete live range as an atomic object, in other words, it is assumed that one variable must always reside in the same register. The fact that the register allocation problem is modeled through the interference graph loses information on the program itself and the exact location of interferences (this is a well-known fact, which led to many different heuristics).

This raises the question: What if we allow to split live ranges? Consider Figure 1 again and a particular variable, for example a. In block B_a, variable a is needed for the instruction "return $a + x$", and this value can come from blocks $B_{a,b}$ and $B_{a,c}$. Therefore, even if we split the live range of a in block B_a before it is used, some register must contain the value of a both at the exit of blocks $B_{a,b}$ and $B_{a,c}$. The same is true for all other variables. In other words, if we consider the possible copies live at exit of blocks of type $B_{u,v}$ and at entry of blocks of type B_v, we get the same interference graph G for the copies. Therefore, the problem remains NP-complete even if we allow live range splitting.

Splitting live ranges does not help here because, in the general case, the control-flow edges from $B_{u,v}$ to B_u are *critical* edges, i.e., they go from a block with more than one successor to a block with more than one predecessor. This forces the live range of a copy to span more than one edge, leading to the well-known notion of *web*. All copies of a given variable a are part of the same web and must be assigned the same color. In general, defining precisely *what* is colored is important as the subtle title of Cytron and Ferrante's paper "What's in a name?" pointed out [8].

To conclude this section, we can interpret Chaitin's original proof as follows. It shows that it is NP-complete to decide if the variables of an arbitrary program can be assigned to k registers, even if live range splitting is allowed, but only when the program has critical edges that we are not allowed to remove (i.e., we cannot change the structure of the control flow graph and add new blocks). In the following section, we consider the case of programs with *no* critical edges. The case of programs with *some* critical edges with a particular structure will be addressed in another report.

3 SSA-like programs and fixed splitting points

In [19], Ferreira and Palsberg pointed out that the construction of Chaitin (as done in Figure 1) is not enough to prove anything about register allocation through SSA. Indeed, to assign variables to registers for programs built as in Section 2, one just have to add extra blocks (where out-of-SSA code is traditionally inserted) and to perform some register-to-register moves in these blocks. Any such program can now be allocated with only 3 registers (see Figure 2 for a possible allocation of the program of Figure 1). Indeed, as there are no critical edges anymore,

[1] Actually, Chaitin's definition of interference is slightly different: Two variables interfere only if one is live at the definition of the other one. However, the two definitions coincide for programs where any control-flow path from the beginning of the program to a given use of a variable goes through a definition of this variable. Such programs are called *strict*. This is the case for the programs we manipulate in our NP-completeness proofs.

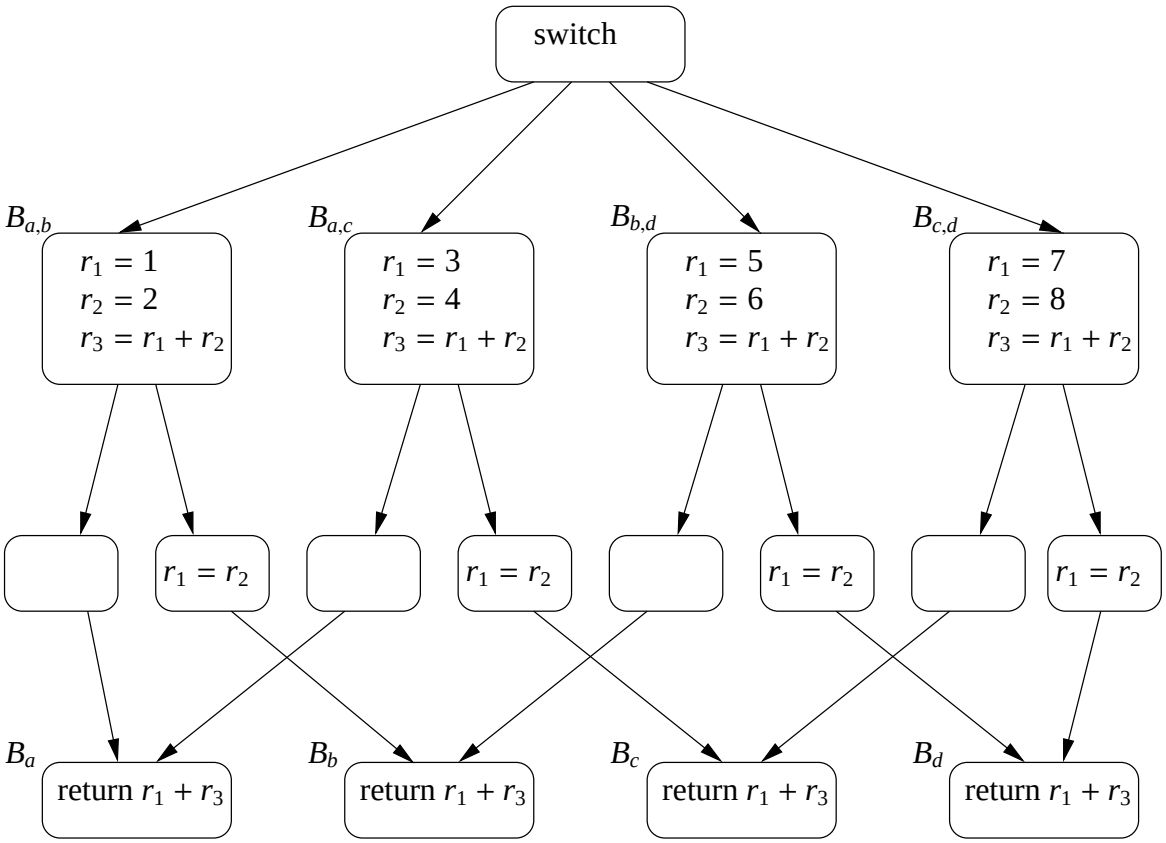


Figure 2: The program of Figure 1 assigned to 3 registers, with additional basic blocks .

we can color the two variables of each basic block of type $B_{u,v}$ independently and "repair", when needed, the coloring to match the colors at each join, i.e., each basic block of type B_u. This is done by introducing an adequate re-mapping of registers (here a single move) in the new block along the edge from $B_{u,v}$ to B_u.

When there are no critical edges, one can indeed go through SSA (or any representation of live ranges as subtrees of a tree), i.e., consider that all definitions of a given variable belong to different live ranges, and to color them with k colors, if possible, in linear time (because the corresponding interference graph is chordal) in a greedy fashion.

At this stage, it is of course easy to decide if k registers are enough. This is possible if and only if Maxlive, the maximal number of values live at any program point, is less than k. Indeed, Maxlive is obviously a lower bound for the minimal number of registers needed, as all variables live at a given point interfere (at least for strict programs). Furthermore, this lower bound can be achieved by coloring because of a double property of such live ranges: a) Maxlive is equal to the size of a maximal clique in the interference graph (in general, it is only a lower bound); b) the size of a maximal clique and the chromatic number of the graph are equal (as the graph is chordal). Furthermore, if k registers are not enough, additional splitting will not help as splitting does not change Maxlive.

If k colors are enough, it is still possible that colors do not match at join points where live ranges were split. Some "shuffle" of registers is needed in the block along the edge where colors do not match. The fact that the edge is not critical guarantees that the shuffle will not propagate along other control flow paths. A shuffle is a permutation of the registers. If some register is

available at this program point, i.e., if Maxlive $< k$, then any remapping can be performed as a sequence of register-to-register moves, possibly using the free register as temporary storage. Otherwise, one additional register is needed unless one can perform register swap (arithmetic operations are also possible but maybe only for integer registers).

This view of coloring through the insertion of permutations is the base of any approach that optimizes spilling first. Some spilling is done (optimally or not) so as to reduce the register pressure (Maxlive) to at most k. In [1], this approach is even used in the most extreme form: live ranges are split at each program point in order to address the problem of optimal spilling. After the first spilling phase, there is a potential permutation between any two program points. Then, live ranges are merged back, as most as possible, thanks to coalescing.

In other words, it seems that going through SSA (for example) makes easy the problem of deciding if k registers are enough. The only possible remaining case is if we do not allow any register swap. If colors do not match at a joint point where Maxlive $= k$, then the permutation cannot be performed. This is the question addressed by Ferreira and Palsberg in [19]: Can we easily choose an adequate coloring of the SSA representation so that no permutation (different than identity) is needed? The answer is no, the problem is NP-complete.

To show this result, Ferreira and Palsberg use a reduction from the problem of coloring circular-arc graphs [10]. Basically, the idea is to start from a circular-arc graph, to choose a particular split point of the arcs to get an interval graph, to represent this interval graph as the interference graph of some basic block, to add a back edge to form a loop, and to make sure that Maxlive $= k$ on the back edge. In this case, coloring the basic block so that no permutation is needed on the back edge is equivalent to coloring the original circular-arc graph. Actually, this is the same proof technique used in [10] to reduce the coloring of circular-arc graphs from a permutation problem.

This proof shows that if the split points are chosen *a priori*, then it is difficult to choose the right coloring of the SSA representation (and thus decide if k registers are enough) even for a simple loop and a single split point. However, for a fixed k, this specific problem is polynomial as it is the case for the k-coloring problem of circular-arc graphs. We now show that, with a simple variation of Chaitin's proof, a similar result can be proved even for a fixed k, but for an arbitrary program.

Consider the same program structure as Chaitin does, but after critical edges have been removed, thus a program structure such as in Figure 2. Given an arbitrary graph $G = (V, E)$, the program has three variables u, x_u, y_u for each vertex $u \in V$ and a variable $x_{u,v}$ for each edge $(u, v) \in E$. It has the following structure:

- For each $(u, v) \in E$, there is a block $B_{u,v}$ that defines u, v, and $x_{u,v}$.
- For each $u \in V$, there is a block B_u that reads u, y_u, and x_u, and returns a new value.
- For each block $B_{u,v}$, there is a path to the blocks B_u and B_v. Along the path from $B_{u,v}$ to B_u, there is a block that reads v and $x_{u,v}$ to define y_u, and then defines x_u.
- An entry block switches to all blocks $B_{u,v}$.

The interference graph restricted to variables u (those that correspond to vertices of G) is still exactly G. Figure 3 shows the program associated to a cycle of length 4, with edges (a, b), (a, c), (b, d), and (c, d). It has no critical edge.

Assume that permutations can be placed only along the edges, or equivalently on entry or exit of the intermediate blocks that are between blocks of type $B_{u,v}$ and type B_u. We claim that the program can be assigned to 3 registers if and only if G is 3-colorable. Indeed, it is easy to see that on each control-fbw edge, exactly 3 variables are live, therefore if only 3 registers are used, no permutation different than identity can be performed. As a consequence, the live range of any variable $u \in V$ cannot be split, each variable is therefore assigned to a unique color. Using the same color for the corresponding vertex in G gives a 3-coloring of the G. Conversely, if G is 3-colorable, assign to each variable u the same color as the vertex u. It remains to color the variables $x_{u,v}$, x_u, and y_u. This is easy: in block $B_{u,v}$, only two colors are used so far, the color for u and the color for v, so $x_{u,v}$ can be assigned the remaining color. Finally assigned x_u to a color different than u, and y_u to the remaining color. This gives a valid register assignment of the program.

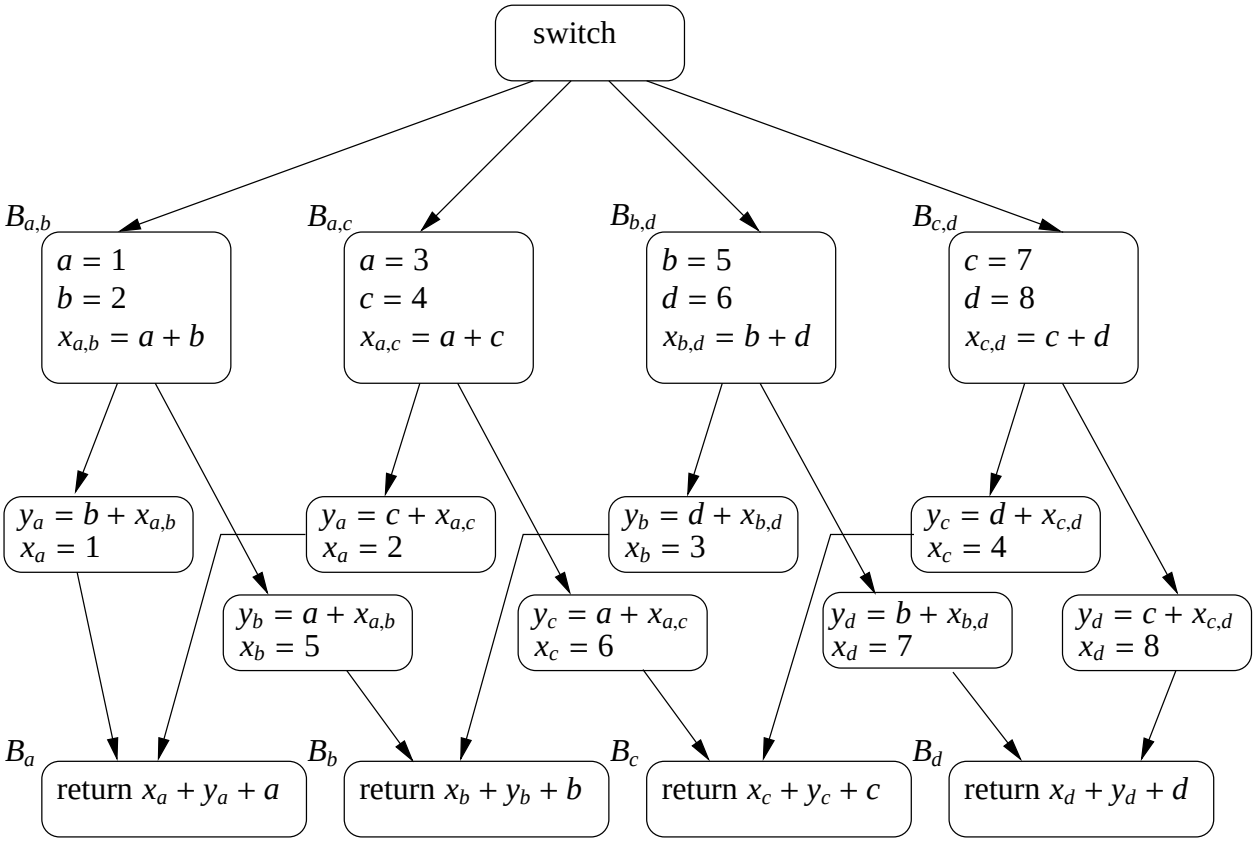


Figure 3: The program associated to a cycle of length 4.

To get a similar proof for any fixed $k > 3$, just add $k - 3$ variables in the switch block and make their live ranges traverse all other blocks. What we just proved, with this slight variation of Chaitin's proof, is that if split points are fixed (as this is traditionally the case when going out of SSA), then it is NP-complete to decide if k registers are sufficient, even for a fixed $k \geq 3$ and even if the program has no critical edge.

4 When split points can be anywhere

Does the study of Section 3 completely answer the question? Not quite. Indeed, who said that split points are fixed? Why can't we shuffle registers at any program point? Consider Figure 3 again. The register pressure is 3 on any control-flow edge, but it is not 3 everywhere. In particular, between the definitions of each y_u and each x_u, the register pressure drops to 2. At this point, some register-to-register moves could be inserted to permute two colors.

Actually, if we allow to split wherever we want then, for such a program, 3 registers are always enough. Indeed, for each block $B_{u,v}$, color u, v, and $x_{u,v}$ with 3 different colors, arbitrarily. For each block B_u, do the same for u, x_u, and y_u. In the block between $B_{u,v}$ and B_u, give to x_u the same color it has in B_u and give to y_u a color different than the color given to u in $B_{u,v}$. Now, between the definitions of y_u and x_u, only two registers contain a live value: the register that contains u defined in $B_{u,v}$ and the register that contains y_u. These two values can be moved to the registers where there are supposed to be in B_u, with one move, two moves, or three moves in case of a swap, using the available register in which x_u is going to be defined just after this shuffle.

So, is it really NP-complete to decide if k registers are enough when splitting can be done anywhere? The problem with the previous construction is that there is no way to not leave a program point with a low register pressure with simple statements while keeping NP-completeness. But, if we are considering the register allocation problem for an architecture with instructions that can define more than one value, it is easy to modify the proof. In a block where y_u and x_u are defined, use a parallel statement that uses v and $x_{u,v}$ and defines y_u and x_u simultaneously, for example something like $(x_u, y_u) = (b + x_{u,v}, b - x_{u,v})$. Now, Maxlive = 3 everywhere in the program and, even if splitting is allowed anywhere, the program can be mapped to 3 registers if and only if G is 3-colorable. Therefore, it is NP-complete to decide if k registers are enough *if two variables can be created simultaneously by a machine instruction*, even if there is no critical edge and if we can split wherever we want. Notice the similarity with circular-arc graphs: as noticed in [10], the problem of coloring circular-arc graphs remains NP-complete even if at most 2 circular arcs can start at any point (but not if at most 1 can start, as we show below).

However, if such instructions exist, it is more likely that a register swap is also provided in the architecture, in which case we are back to the easy case where any permutation can be done and k registers are enough if and only if Maxlive = k. It remains to consider one case: what if only one variable can be created at a given time as it is in traditional sequential assembly code representation? We claim this is polynomial to decide if k registers are enough, in the case of a strict program and if we are allowed to introduce blocks to remove critical edges. This can be done as follows.

Consider the program after edge splitting and compute Maxlive, the maximal number of values live at any program point. If Maxlive $< k$, it is always possible to assign variables to k registers by splitting live ranges as we already discussed because adequate permutations can always be performed. If Maxlive $> k$, this is not possible [2], more spilling has to be done. The remaining case is thus when Maxlive = k.

If Maxlive = k, restrict to the control-flow graph defined by program points where exactly k variables are live. We claim that, in each connected component of this graph, if k registers are enough, there is a unique solution, up to a permutation of colors. Indeed, for each connected

[2] This is true for a strict program. For a non-strict program, one needs to consider another definition of Maxlive. We do not address non-strict programs in this report.

component, start from a particular program point and a particular coloring of the k variables live at this point. Propagate this coloring in a greedy fashion, backwards or forwards along the control flow. In this process, there is no ambiguity because the number of live variables remains equal to k: At any program point, since one variable (and only one) is created, exactly one must become dead, and the new variable must be assigned the same color as the dead one. Therefore, going backwards or forwards defines a *unique* solution (up to the initial permutation of colors). In other words, if there is a solution, we can define it, in each connected component, by propagation. If, during this traversal, we reach a program point already assigned and if the colors do not match, this *proves* that k registers are not enough.

Finally, if the propagation of colors on each connected component is possible, then k registers are enough for the whole program. Indeed, we can color the rest in a greedy (but not unique) fashion and, when we reach a point already assigned, we can resolve a possible register mismatch because at most $k - 1$ variables are live at this point.

To conclude, to decide if k registers are enough, one just need to propagate colors along the control flow. We first propagate along program points where Maxlive $= k$. If we reach a program point already colored and the colors do not match, more spilling needs to be done. Otherwise, we start a second phase of propagation, along all remaining program points. If we reach a program point already colored and the colors do not match, we resolve the problem with a permutation of at most $k - 1$ registers.

5 Conclusion

In this report, we tried to make clearer where the complexity of register allocation comes from. Our goal was to recall what exactly Chaitin's original proof proves and to extend this result. The main question addressed by Chaitin is of the following type: Can we decide if k registers are enough for a given program or if some spilling is necessary?

The original proof of Chaitin [6] proves that this problem is NP-complete if each variable can be assigned to only one register (i.e., no live range splitting). We showed that Chaitin's construction also proves the NP-completeness of the problem if live range splitting is allowed but if we are not allowed to remove critical edges (i.e., no edge splitting).

Recently, Ferreira and Palsberg [19] proves that, if k is arbitrary and the program is a loop, then the problem remains NP-complete if live range splitting is allowed but only on a block on the back edge and if register swaps are not available. This is a particular form of register allocation through SSA. We showed that Chaitin's proof can be extended to show a bit more. The problem remains NP-complete for a fixed $k \geq 3$, even if the program has no critical edge and if we can split live ranges along any control-flow edge (but not inside basic blocks). This again is if register swaps are not available.

These results do not address the general case where we are allowed to split wherever we want, including inside basic blocks. We show that the problem remains NP-complete but only if some instructions can define two variables simultaneously. For a strict program and/or if we consider that two variables interfere if and only if they are both live at some program point, we can answer the remaining cases. First k must be at least Maxlive, the maximal number of variables live at any program point. If Maxlive $< k$ or if register swaps are available (which is likely to be the case if some instructions can define two variables simultaneously) then k registers are enough. If register swaps are not available and if only one variable can be defined

at a given program point, then a simple polynomial-time greedy approach can be used to decide if k registers are enough.

This study shows that the NP-completeness of register allocation is *not* due to the coloring phase (as a misinterpretation of Chaitin's proof may suggest); deciding if k registers are enough or if spilling is necessary is not as hard as one might think. The NP-completeness of register allocation is due to the presence of critical edges or not, and to the optimization of spilling costs (which variables should be spilled and where so as to reduce Maxlive with a minimal cost?) and coalescing costs (which live ranges should be fused so as to minimize register-to-register moves while keeping the graph k-colorable?).

References

[1] Andrew W. Appel and Lal George. Optimal spilling for cisc machines with few registers. In *PLDI'01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 243–253, Snowbird, Utah, United States, June 2001. ACM Press.

[2] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation and spill complexity under SSA. Technical Report RR2005-33, LIP, ENS-Lyon, France, August 2005.

[3] Preston Briggs, Keith Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[4] Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*, June 2005.

[5] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the ACM SIGPLAN'82 Symposium on Compiler Construction*, volume 17(6) of *SIGPLAN Notices*, pages 98–105, 1982.

[6] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[7] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[8] Ron Cytron and Jeanne Ferrante. What's in a name? or the value of renaming for parallelism detection and storage allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27. IEEE Computer Society Press, August 1987.

[9] Martin Farach-Colton and Vincenzo Liberatore. On local register allocation. *Journal of Algorithms*, 37(1):37–65, 2000.

[10] M. R. Garey, D. S. Johnson, G. L. Miller, and C. H Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM Journal of Algebraic Discrete Methods*, 1(2):216–227, 1980.

[11] Michael R. Garey and Davis S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

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

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

[14] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *Compiler Construction 2006*, volume 3923 of *LNCS*. Springer Verlag, 2006.

[15] Allen Leung and Lal George. Static single assignment form for machine code. In *PLDI'99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 204–214. ACM Press, 1999.

[16] Vincenzo Liberatore, Martin Farach-Colton, and Ulrich Kremer. Evaluation of algorithms for local register allocation. In *Compiler Construction - 8th International Conference, CC'99, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99*, volume 1575 of *Lecture Notes in Computer Science*, pages 137–152, Amsterdam, The Netherlands, March 1999. Springer Verlag.

[17] Guei-Yuan Lueh, Thomas Gross, and Ali-Reza Adl-Tabatabai. Fusion-based register allocation. *ACM Transactions on Programming Languages and Systems*, 22(3):431–470, 2000.

[18] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *Proceedings of APLAS'05, Asian Symposium on Programming Languages and Systems*, pages 315–329, Tsukuba, Japan, November 2005.

[19] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation after classical SSA elimination is NP-complete. In *Proceedings of FOSSACS'06, Foundations of Software Science and Computation Structures*, Vienna, Austria, March 2006.

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

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