Register Allocation: What does the NP-completeness Proof of Chaitin et al. Really Prove?[*] [†]

Florent Bouchez Alain Darte Fabrice Rastello

Laboratoire de l'Informatique du Parallélisme
UMR CNRS-ENS Lyon-UCB Lyon-INRIA 5668, France
Firstname.Lastname@ens-lyon.fr

ABSTRACT

Register allocation is one of the most studied problems in compilation. It is considered as an NP-complete problem since Chaitin et al., in 1981, modeled the problem of assigning temporary variables to k machine registers as the problem of coloring, with k colors, the interference graph associated to the variables. The fact that the interference graph can be arbitrary proves the NP-completeness of this formulation. However, this original proof does not really show where the complexity of register allocation 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 form to perform register allocation in polynomial time, without contradicting Chaitin et al's NP-completeness result? To address such a question and, more generally, the complexity of register allocation, we revisit Chaitin et al's proof to better identify the interactions between spilling (load/store insertion), coalescing/splitting (removal/insertion of moves between registers), critical edges (a property of the control-flow graph), and coloring (assignment to registers). In particular, we show that, in general (we will make clear when), it is *easy* to decide if temporary variables can be assigned to k registers or if some spilling is necessary. In other words, the real complexity does not come from the coloring itself (as a wrong interpretation of the proof of Chaitin et al. may suggest) but comes from the presence of critical edges and from the optimizations of spilling and coalescing.

Keywords

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

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 usually 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 transferred ("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 [17], a modified version of previous developments by Chaitin et al. [9, 8], and Briggs et al. [4]. 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 optimizing compilers. 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 et al. developed their first heuristics (in 1981-82). 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 on most architectures 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 it is worth clarifying.

This last point is particularly interesting to note. In 1981, Chaitin et al. [9] modeled the problem of allocating variables of a program to k registers as the problem of coloring, with k colors, the corresponding interference graph (two variables interfere if they are simultaneously live at some program point). As one can build a code corresponding to an arbitrary interference graph and because graph coloring is NP-complete [16, Problem GT4], heuristics have been used for spilling, coalescing, splitting, coloring, etc. The previous argument (register allocation *is* graph coloring, therefore it is NP-complete) is one of the first statements of many papers on register allocation. This is true that most problems related to register allocation are NP-complete but this simplifying statement can make us forget what Chaitin et al's proof actually shows. In particular, it is in the common belief that, when no instruction rescheduling is allowed, deciding if some spilling is necessary to allocate variables to k registers is NP-complete, even if live-range splitting is allowed. This is *not* what Chaitin et al. proved. We will even show that this particular problem is not NP-complete except for a few particular cases (we will make clear which ones). This is maybe a

[*]This work was supported by a contract with STMicroelectronics, Grenoble, France.

[†]WDDD'06, June 18, 2006, Boston, Massachussetts, USA.

folk theorem too but, to our knowledge, it has never been clearly stated. 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 (live-range 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. [22, 14], 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 variables 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] and for several variants of register coalescing problems in [3].

Today, most compilers go through an intermediate code representation, the (strict) SSA form (static single assignment) [11], 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-flow graph. Some so-called ϕ functions are used to transfer values along the control flow not covered by the dominance tree. The consequence is that, with an adequate interpretation of ϕ functions, 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 if 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 decides which values are spilled and where, so as to get to a code with Maxlive $\leq k$. This phase is called *allocation* in [22] as it decides which variables are allocated in memory and which variables are allocated in registers. The second phase (called *register assignment* in [22]) maps variables to registers, possibly removing (i.e., coalescing) or introducing (i.e., splitting) move instructions (also called shuffle code in [23]). Considering that loads and stores are more expensive than moves, such an approach is worth exploring. This is the approach experimented by Appel and George [1] and also advocated in [20, 2, 19].

The fact that interference graphs of strict SSA programs are chordal is not a new result if one makes the connection between graph theory and SSA form. Indeed, a theorem of Walter (1972), Gavril (1974), and Buneman (1974) (see [18, 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 when trying to better understand the interplay of register allocation and coalescing for out-of-SSA conversion [13]. Independently, Brisk et al. [6], Pereira and Palsberg [25], and Hack et al. [19] made the same observation on the chordality of SSA interference graphs. A direct proof of the chordality property for strict SSA programs can be given, see for example [2, 19].

Many papers [11, 5, 21, 28, 7, 27] address the problem of how to go out of SSA, i.e., how to generate a code, after SSA optimizations, that does not contain ϕ functions (which are not machine code) anymore. The difficulties are how to handle renaming constraints (due to specific requirements of the architecture), critical edges (a fundamental property of the control-flow graph that will be discussed hereafter), and how to reduce the number of moves that need to be introduced. However, in these papers, register allocation is performed *after* out-of-SSA conversion: in other words, the number of registers is not a constraint when going out of SSA and, conversely, the SSA form is not exploited to perform register allocation. In [19] on the other hand, the SSA form is used to do register allocation. Spilling and coloring (i.e., register assignment) are done in SSA and some permutations of colors are placed on new predecessor blocks of the ϕ points to emulate the semantics of ϕ functions. Such permutations can always be performed with register-to-register moves (and possibly register swaps or XOR functions) [1]. All these new results related to SSA form, combined with the idea of spilling before coloring so that Maxlive $\leq k$, has led Pereira and Palsberg [26] to wonder where the NP-completeness of Chaitin et al'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, if register swaps are not available, the answer is no unless P=NP.

The NP-completeness proof of Pereira and Palsberg is interesting, but we feel it does not completely explain why register allocation is difficult. Basically, it shows that if we decide *a priori* what the splitting points are, i.e., the program points where register-to-register moves can be placed (in their case, the splitting points are the ϕ points), then it is NP-complete to choose the right colors (they do not allow register swaps as in [19]). However, there is no reason to restrict to splitting points given by SSA. Actually, we show in this paper that, when we can choose the splitting points, when we are free to add program blocks so as to remove critical edges (a standard technique called *edge splitting*), then it is in general easy (we will make clear when) 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 et al's proof (and what it does not imply) and to make clearer the interactions between spilling, splitting, coalescing, critical edges, and coloring.

In Section 2, we first reproduce Chaitin et al'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 Pereira and Palsberg in [26]: we show that Chaitin et al's proof can be easily 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 if register swaps are not available. In Section 4, we show, again with a slight variation of Chaitin et al'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 swaps 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-level code representation) and register swaps are not available. Therefore, this study shows that the NP-completeness of register allocation (for a fixed schedule) is *not* due to the coloring phase (as a misinterpretation of Chaitin et al'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. In Section 5, we summarize our results and discuss how they can be used to improve previous approaches and to develop new register allocation schemes.

[1]However, minimizing the number of such permutations is NP-complete [19].

2. DIRECT CONSEQUENCES OF CHAITIN ET AL'S NP-COMPLETENESS PROOF

Let us look at Chaitin et al's NP-completeness proof again. The proof is by reduction from graph k-coloring [16, 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 $\{1, \dots, k\}$ such that $c(v) \neq c(u)$ for each edge $(u, v) \in E$? The problem is well-known to be NP-complete if G is arbitrary, even for a fixed $k \geq 3$. For the reduction, Chaitin et al. create a program with $|V| + 1$ variables, one for each vertex $u \in V$ and an additional variable x, and the following structure:

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

Consider the graph depicted in Figure 1, a cycle of length 4, with edges (a, b), (a, c), (b, d), and (c, d). This is also the example used in [26]. The corresponding program is given in Figure 2. 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 et al. proved: for such programs, deciding if one can assign the variables, *this way*, to $k \geq 4$ registers is thus NP-complete.

Chaitin et al's proof, at least in its original interpretation, does not address the possibility of splitting [10] the live-range of a variable (set of program points where the variable is live [2]). In other words, each vertex of the interference graph represents the complete live-range as an atomic object, and 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 has led to many different register allocation heuristics but with no corresponding complexity study even though their situations are not covered by the previous NP-completeness proof.

This raises the question: What if we allow to split live-ranges? Consider Figure 2 again and one of the variables, for example a.

[2]Actually, Chaitin et al'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 static 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 all the programs we manipulate in our NP-completeness proofs.

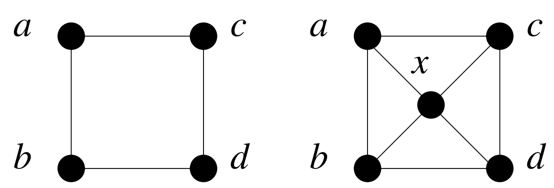

Figure 1: Cycle of length 4 and interference graph for Figure 2.

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 $B_{a,c}$. If we split the live-range of a in block B_a before it is used, some register must still 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 and each copy must remain in the same register. Therefore, the problem remains NP-complete even if we allow live-range splitting. Splitting live-ranges does not help here because 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. In Chaitin et al's model, each vertex is atomic and must be assigned a unique color. Live-range splitting redefines these objects. In general, defining precisely *what* is colored is indeed important as the subtle title of Cytron and Ferrante's paper "What's in a name?" pointed out [12]. However, here, because of critical edges, whatever the splitting, there remains atomic objects hard to color, defined by the copies live on the edges.

To conclude this section, we can interpret Chaitin et al'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 split (i.e., we cannot change the structure of the control flow graph and add new blocks).

3. SPLIT POINTS ON ENTRY & EXIT OF BLOCKS AND TREE-LIKE PROGRAMS

In [26], Pereira and Palsberg pointed out that the construction of Chaitin et al. (as done in Figure 2) 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 3 for a possible allocation of the program of Figure 2). Indeed, we can place (color) the two variables of each basic block of type $B_{u,v}$ in two registers (independently of other blocks), for example always using r_1 for u, r_2 for v, and r_3 for x, and then "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 general a permutation) 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 the colors chosen for

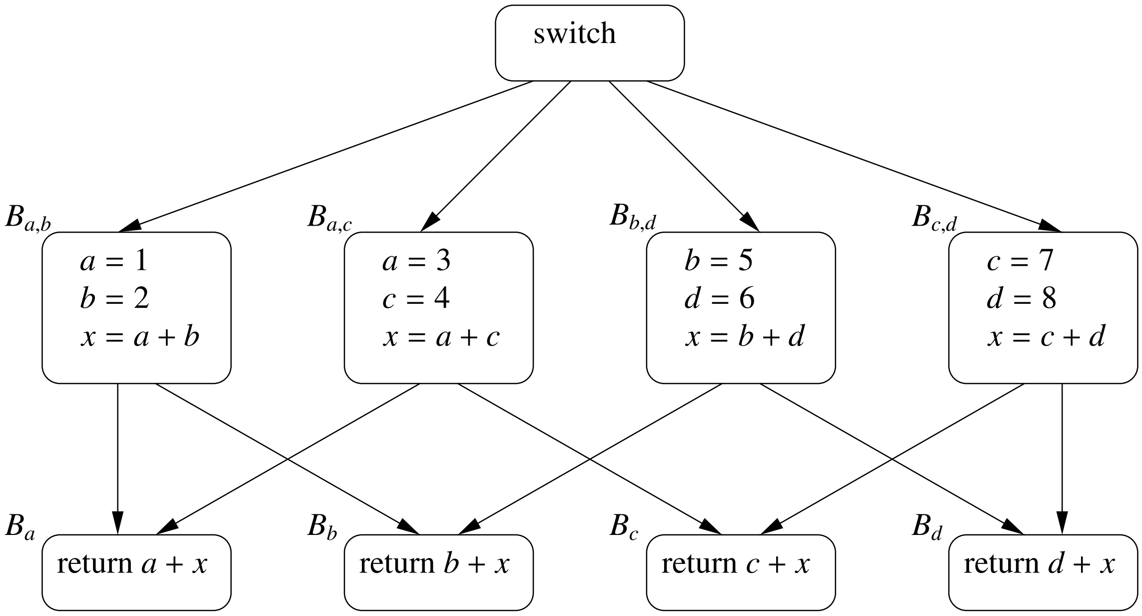

Figure 2: The program associated to a cycle of length 4.

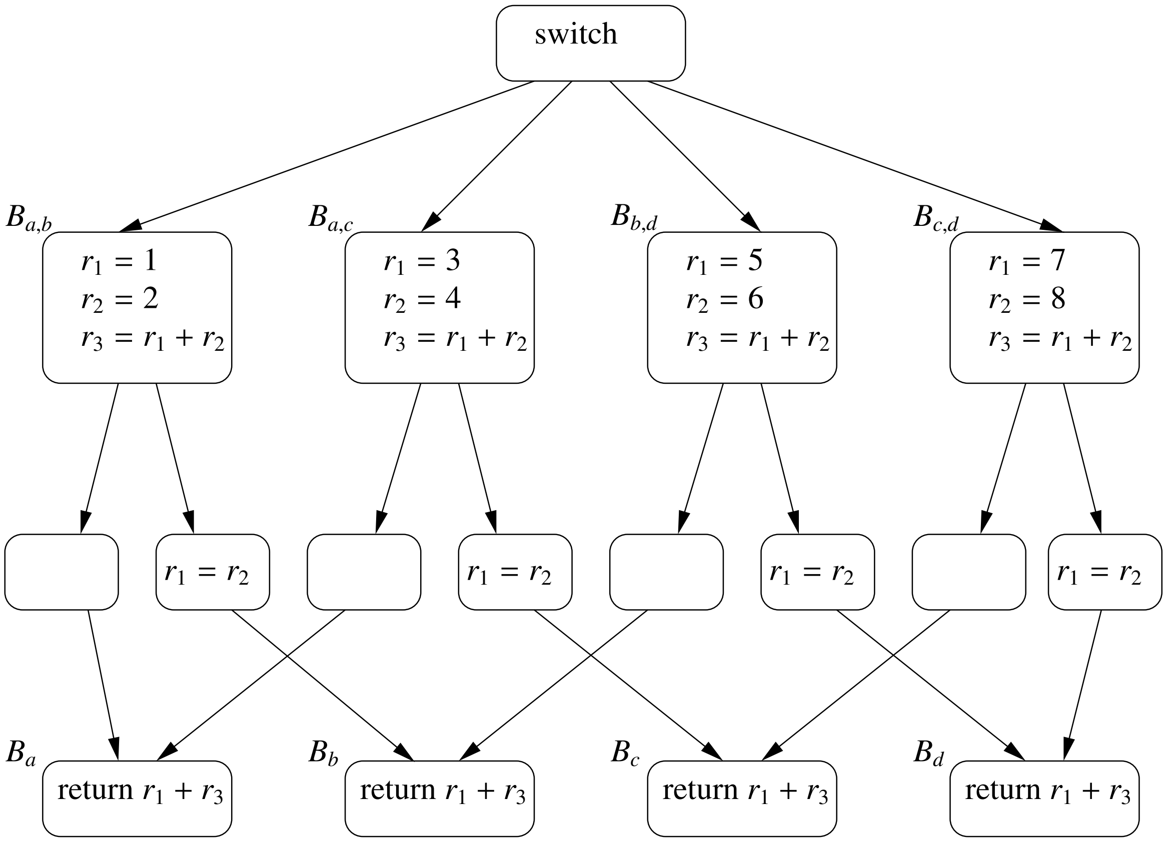

Figure 3: The program of Figure 2 assigned to 3 registers, with additional basic blocks.

the split live-ranges do not match at join points where live-ranges were split. Some "shuffle" [23], i.e., permutation 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. 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 swaps (arithmetic operations such as XOR 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 [20, 1, 2, 19]. Some spilling and splitting are 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 but not only) 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 Pereira and Palsberg in [26]: 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, Pereira and Palsberg use a reduction from the problem of coloring circular-arc graphs, proved NP-complete by Garey et al. [15]. Basically, the idea is to start from a circular-arc graph, to cut all arcs at some point to get an interval graph, to represent this interval graph as the interference graph of a basic block, to add a back edge to form a loop, and to make sure that Maxlive $= k$ on the back edge. Then, coloring the basic block so that no permutation is needed on the back edge is equivalent to coloring the original circular-arc graph. This is the same proof technique used in [15] to reduce the coloring of circular-arc graphs from a permutation problem.

This proof shows that if we restrict to the split points defined by SSA, 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, by propagating possible permutations. We now show that, with a simple variation of Chaitin et al's proof, a similar result can be proved even for a fixed k, but for an arbitrary program.

Consider the control-flow graph as Chaitin et al. do, but after critical edges have been split, as shown in Figure 3. 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 $(u, v) \in E$, a block $B_{u,v}$ defines u, v, and $x_{u,v}$.
- For each $u \in V$, a block B_u 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, a block 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. Consider again a cycle of length 4, with edges (a, b), (a, c), (b, d), and (c, d), as in Figure 4 (on the left). The corresponding program is given in Figure 5 and its interference graph in Figure 4 (on the right).

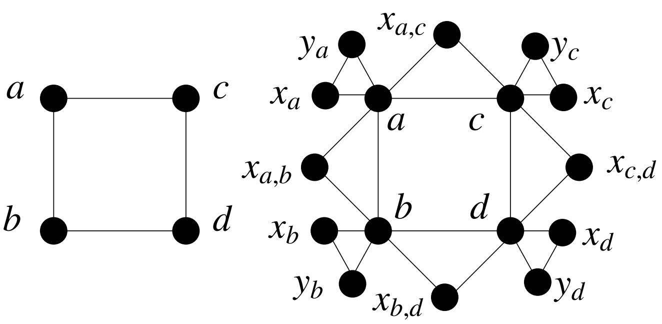

Figure 4: Cycle of length 4 and interference graph for Figure 5.

Assume that permutations can be placed only along the edges, or equivalently on entry or exit of the intermediate blocks, 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, for each u and v, exactly 3 variables are live on exit of $B_{u,v}$ and on entry of B_u and B_v. Thus, 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*, i.e., each variable must be assigned to a unique color. Using the same color for the corresponding vertex in G gives a 3-coloring of 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 colors for u and v, so $x_{u,v}$ can be assigned the remaining color. Finally assign x_u and y_u to two colors different than the color of u (see Figure 4 again to visualize the cliques of size 3). This gives a valid register assignment.

To get a similar proof for any fixed $k \geq 3$, add $(k - 3)$ variables in the switch block and make their live-ranges traverse all blocks. To conclude, this slight variation of Chaitin et al's proof shows that if we cannot split inside basic blocks but are allowed to split only on entries and exits of blocks (as this is traditionally done 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 for a program with no critical edge.

4. IF SPLIT POINTS CAN BE ANYWHERE

Does the study of Section 3 completely answer the question? Not quite. Indeed, who said that split points need to be on entry and exit of blocks? Why can't we shuffle registers at any program point, in particular in the middle of a block if this allows us to perform a permutation? Consider Figure 5 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. (To follow the discussion, consider Figure 5 again.) Indeed, one can color independently the top part (including the variables y_u) and the bottom part (including the variables x_v), then place permutations between the definitions of variables y_u and x_u. More precisely, for each block $B_{u,v}$ independently, color the definitions of u, v, and $x_{u,v}$ with three different colors, arbitrarily. For each block B_u, do the same for u, x_u, and y_u (i.e., define arbitrarily three registers where u, x_u, and y_u are supposed to be on block entry). In the block between $B_{u,v}$ and B_u, keep u in the same register as $B_{u,v}$, give to x_u the same color it has in B_u and store y_u in a register not used by u in $B_{u,v}$. So far, all variables are correctly colored except that there may be a need of register moves for the values u and y_u, after the definition of y_u, and before their uses in B_u, if the colors do not match. But, between the definitions of y_u and x_u, only two registers contain a live value: one containing u defined in $B_{u,v}$ and one containing y_u. These two values can thus be moved to the registers where they are supposed to be in B_u, with at most three moves in case of a swap, using the available register in which x_u is going to be placed just after this shuffle.

4.1 Simultaneous definitions

So, is it really NP-complete to decide if k registers are enough when splitting can be done anywhere and swaps are not available? The problem with the previous construction is that there is no way, with simple statements, to avoid a program point with a low register pressure while keeping the reduction with graph 3-coloring. This is illustrated in Figure 6: on the left, the previous situation with Maxlive = 2, in the middle, a situation with Maxlive = 3 but that does not keep the equivalence with the 3-colorability of the graph. However, if we are considering the complexity of register allocation for an instruction set architecture where instructions can define more than one value, it is easy to modify the proof. In a block between $B_{u,v}$ and B_u, use a statement that consumes v and $x_{u,v}$ and produces y_u and x_u simultaneously, for example something like $(x_u, y_u) = (b + x_{u,v}, b - x_{u,v})$ as illustrated on the right of Figure 6. 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. (Also such an instruction should consume at least two values, otherwise, the register pressure drops to Maxlive − 1 just before and a permutation can be placed.) Notice the similarity with circular-arc graphs: as noticed in [15], the problem of coloring circular-arc graphs remains NP-complete even if at most two circular arcs can start at any point (but not if only one can start, as we show below).

Besides, if such machine instructions exist, it is likely that a register swap is also provided in the architecture (we discuss such architectural subtleties at the end of this section). In this 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. Thus, 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-level code representation and register swaps are not available?

4.2 Only one definition at a time

If blocks can be introduced to split critical edges and live-range splitting can be done anywhere, we claim that it is polynomial to decide if k registers are enough, in the case of a strict program. We proceed as follows.

Consider the program after edge splitting and compute Maxlive, the maximal number of values live at any program point. As we already discussed, if Maxlive < k, it is always possible to assign variables to k registers by splitting live-ranges because adequate permutations can always be performed, thanks to a remaining available register at any point. If Maxlive > k, this is not possible [3], 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 component, start from an arbitrary program point and an arbitrary coloring of the k variables live at this point. Propagate this coloring in a greedy fashion, backwards and forwards along the control flow until all points are reached. In this process, there is no ambiguity to choose a color: at any program point, the number of live variables remains equal to k, one variable (and only one) is created, thus exactly one must become dead, and the new variable must be assigned the same color as the dead one. Therefore, for each connected component, going backwards and forwards defines a *unique* solution (up to the initial permutation of colors), if it exists. In other words, if there is a solution, we can define it, for each connected component, by propagation. Furthermore, 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 (where Maxlive < k) 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 summarize, to decide if k registers suffice when Maxlive ≤ k, 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, possibly using the remaining available register.

4.3 Subtleties of the architectures

To conclude this section, let us illustrate the impact of architectural subtleties with respect to the complexity of the previously analyzed cases, when edge splitting is allowed. We use the example of the ST200 core family from STMicroelectronics, which was the target of this study.

As for many other processors, some variables need to be assigned to specific registers: they are *precolored*. Let us show that such precoloring constraints do not change the complexity of deciding if some spilling is necessary. First consider the case where swaps are available. Without precoloring constraints, Maxlive ≤ k was the condition to be able to color with k registers. This is still true, since we can insert adequate permutations, possibly using swaps, when colors do not match the precolored constraints. Reducing these mismatches is a coalescing problem, as in a regular Chaitin-like approach. Now, consider the second case that was polynomial, i.e., when no register swap is available and instructions create at most one variable. Even with precolored constraints, a similar greedy approach can be used to decide, in polynomial time, if k registers are enough. It just propagates colors from precolored variables, along program points with exactly k live variables, i.e., with no freedom, so it is easy to check if colors match.

A similar situation occurs when trying to exploit auto-increments

[3] This is true for a strict program. For a non-strict program, two variables interfere only if one is live at the definition of the other, which makes possible to use fewer than Maxlive registers. We do not address non-strict programs here. Some open questions remain for such programs.

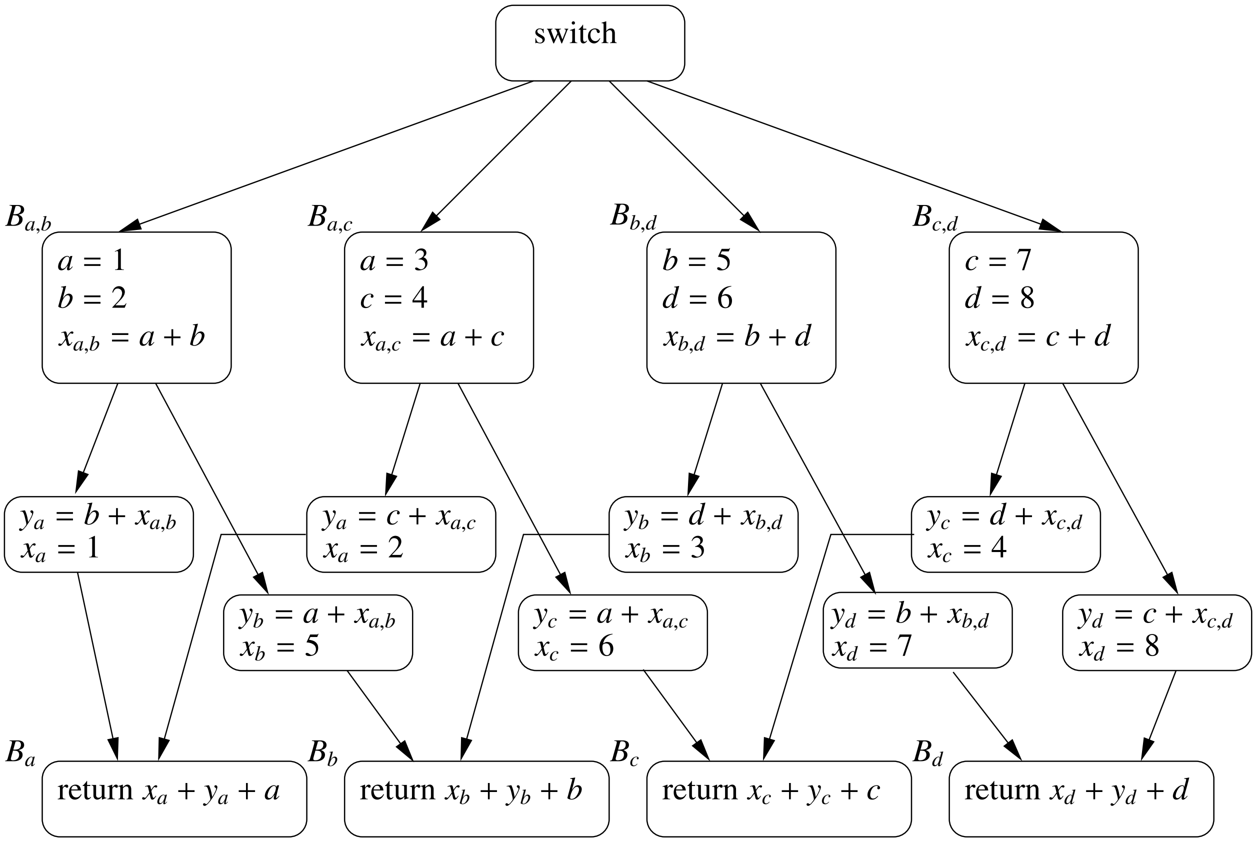

Figure 5: The program associated to a cycle of length 4.

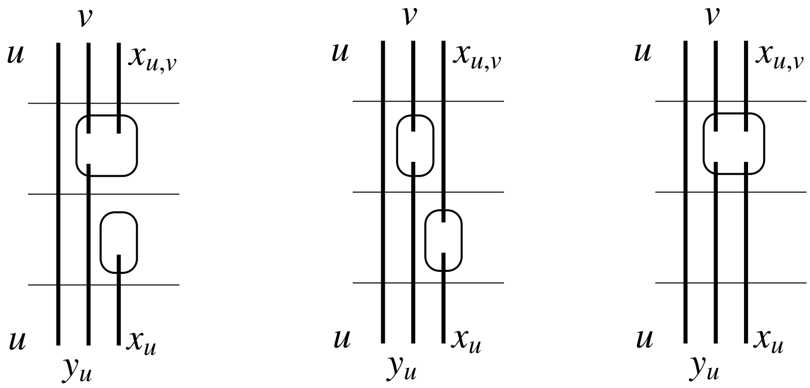

Figure 6: Three situations with Maxlive $= 2$ (on the left) or Maxlive $= 3$ (middle and right).

r_x++, i.e., $r_x = r_x + 1$. An instruction x++ apparently prevents the coloring under SSA as x is redefined, unless it is first rewritten as $y = x + 1$. A coalescer may then succeed in coloring y and x the same way. To enforce such an auto-increment, one can also simply ignore the definition of x in the auto-increment and consider that the live-range of x go through the instruction, as a larger SSA-like live-range.

More annoying are the NP-complete cases due to the fact that register swaps are not available and that some machine instructions can define more than one variable. In the ST200 core family, two types of instructions can create more than one variable: the function calls that can have up to 8 results and the 64 bits load operations that loads a 64 bits value into two 32 bits registers.

- For a function call, the constraint Maxlive $\leq k$ needs to be refined. Spilling must be done so that the number of live variables at the call, excluding parameters and results, is less than the number of callee-save registers. In the ST200 core family, the set of registers used for the results is strictly included in the set of caller-save registers. Therefore, just after the call and before the possible reloads of caller-saved registers, there is at least one free caller-save register that can be used to permute colors if needed. Therefore, function calls do not make the problem NP-complete, even if they can have more than one result. Also, if no caller-save register was available, as results of function calls are in general precolored, this situation could also be solved as previously explained.

- The second type of instruction that can define more than one variable are the 64 bits loads, such as $r_x, r_y = \text{load}(r_z)$, with the additional constraint that $y = x + 1$, i.e., the results are two consecutive registers. Such a load instruction has only one variable argument. So, if the number of live variables is k after defining r_x and r_y, it is at most $k - 1$ just before, so a permutation can be done and there is no problem for coloring with k registers. The fact that r_x and r_y must be consecutive can be addressed by placing a permutation just before the load so as to make two successive registers available. Thus, again, despite the fact that such an instruction has two results, it does not make the problem NP-complete because it has only one variable argument.

Finally, even if no swap operation is available in the instruction set, a swap can be simulated thanks to the parallelism available in the ST200 core family. In the compiler infrastructure, one needs to work with a pseudo-operation swap $R_x, R_y = \text{swap}(R_i, R_j)$, which will then be replaced by two parallel operations scheduled in the same cycle: $R_x = \text{move}(R_i)$ and $R_y = \text{move}(Rj)$. Also, for integer registers, another possibility to swap without an additional register is to use the instruction XOR.

In conclusion, when edge splitting is allowed, even if one needs to pay attention to the subtleties of the architecture, the cases where deciding if some spilling is necessary is NP-complete seem to be quite artificial. In practice, swaps can usually be done and one just has to check that Maxlive $\leq k$. If not, one can rely on a greedy coloring, propagating only along program points where Maxlive $= k$. Instructions with more than one result could make this greedy coloring non deterministic (and the problem NP-complete) but, fortunately, at least for the ST200, these instructions have a neighbor point (either just before or just after) where the number of live variables is strictly less than k. Thus, it is in general easy to decide if some spilling is necessary or if, at the price of additional register-to-register moves, the code can be assigned to k registers.

5. CONCLUSION

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

5.1 Summary of results

The original proof of Chaitin et al. [9] proves that this problem is NP-complete when live-range splitting is not allowed, i.e., with the constraint that each variable can be assigned to only one register. We showed that Chaitin et al's construction also proves that the problem remains NP-complete when live-range splitting is allowed but not (critical) edge splitting.

Recently, Pereira and Palsberg [26] proved that, if k is arbitrary and the program is a simple loop, then the problem is still NP-complete with the constraint that live-range splitting is only allowed on a block of the back edge and register swaps are not available. This is a particular form of register allocation through SSA. We showed that Chaitin et al's proof can be extended to show a bit more. When register swaps are not available, the problem remains NP-complete for a fixed $k \geq 3$ (but for a general control-flow graph), even if the program has no critical edge and if live-range splitting can be done on any control-flow edge, i.e., on entry and exit of blocks, but not inside basic blocks.

These results do not address the general case where live-range splitting can be done anywhere, including *inside* basic blocks. We showed that the problem remains NP-complete only if some instructions can define two variables at the same time but register swaps are not available. Such a situation might not be so common in practice. For a strict program, we can answer the remaining cases in polynomial time. If Maxlive $= k$ and register swaps are available, or if Maxlive $< k$, then k registers are enough. If register swaps are not available and only one variable can be defined at a given program point, then a simple 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 may suggest a misinterpretation of Chaitin et al's proof, which uses a reduction from graph k-coloring. If live-range splitting is taken into account, deciding if k registers are enough or if some spilling is necessary is not as hard as one might think. The NP-completeness of register allocation is due to three factors: the presence of critical edges or not, the optimization of spilling costs, i.e., how to reduce Maxlive to k, and the optimization of coalescing costs, i.e., which live-ranges should be fused while keeping the graph k-colorable.

5.2 Research directions

What does such a study imply for the developments of register allocation strategies? Most approaches decide to spill because their coloring technique fails to color the live-ranges of the program. But, for coloring, a heuristic is used and this may generate some useless spills. Our study shows that, instead of using an *approximation heuristic* to decide when to spill, we can use an *exact algorithm* to spill only when necessary. Such a test is fundamental to develop register allocation schemes where the spilling phase is decoupled from the coloring/coalescing phase. However, we point out that this “test”, which is, roughly speaking, to make sure that Maxlive $\leq k$, does not indicate which variables should be spilled and where the store and load operations should be placed, so as to get an optimal code in terms of spill cost (if not execution time). Optimal spilling is a much more complex problem, even for basic blocks [22, 14], for which several heuristics, as well as an exact integer linear programming approach [1], have been proposed.

Existing register allocators give satisfying results, when measured on average for a large set of benchmarks. But many benchmarks do not need any spill for current architectures. However, for some applications, when the register pressure is high, we noticed some possible improvements in terms of spill cost. For example, in Chaitin's first approach, when all variables interfere with at least k other variables, one of them is selected to be spilled and the process iterates. We measured, with a benchmark suite from STMicroelectronics, that such a strategy produces many *useless* spills: we say that a spill is useless if after spilling all chosen variables except this one, Maxlive is still less than k. A similar fact was noticed by Briggs et al. [4] who decided to delay the spill decision to the coloring phase. If a potential spill does not get a color during the coloring phase, it is marked as an *actual spill*, else it is useless. This strategy significantly reduces the number of useless spills compared to Chaitin's initial approach. Other improvements include biased coloring, which in general reduces the number of actual spills, or conservative coalescing and iterated register coalescing [17], as coalescing can reduce the number of neighbors of some vertex of the interference graph.

We applied a very simple strategy in our preliminary experiments: in the set of variables selected for spilling, we choose the most expensive useless spill and we remove it from the set. This process is repeated until no useless spill remains. This simple additional check is enough to reduce the spill cost of Chaitin's initial approach to the same order of magnitude as a biased iterated register coalescing, although it remains a bit worse on average. Also, even for a biased iterated register coalescing, this strategy still detects some useless spills and can improve the spill cost. With this more precise view of necessary spills, one can also avoid the successive phases of spilling needed for a RISC machine: for a RISC machine, a spilled live-range leaves small live-ranges to perform the reloads (the same is true if the live-range is partially spilled). Because of this, a Chaitin-like approach needs to generate spill code and start again. With an exact criterion for spilling needs, we can take into account the new live-ranges to measure Maxlive and decide if more spilling is necessary.

Once spilling is done, variables still have to be assigned to registers. The test "Is some spilling necessary?" does not really give a coloring. For example, if swaps are available, the test is simply Maxlive $\leq k$. One still needs to make sure that coloring with Maxlive registers is possible. As the previous complexity study shows, a possibility is to split all critical edges and to color in polynomial time with Maxlive colors, possibly inserting color permutations. The most extreme possibility is to color independently each program point, whose corresponding interference graph is a clique of at most Maxlive variables, and to insert a permutation between any two points. This amounts to split live-ranges everywhere, as done in [1]. Of course, such a strategy leads to a valid k-coloring, but with an unacceptable number of moves. This can be improved by treating the optimization of moves as a coalescing problem, although it is in general NP-complete [3]. As there are many moves to remove, with tricky structures, a conservative approach does not work well, and an optimistic coalescing [24] seems preferable [1]. Another way is to color independently each basic block, with Maxlive colors, in linear time after renaming each variable so that it is defined only once. This will save all moves inside the blocks. Then permutations between blocks can be improved by coalescing. One can try to extend the basic blocks to larger regions, while keeping them easy to color. This is the approach of fusion-based register allocation [23], except that the spilling decision test is Chaitin's test, thus a heuristic that can generate useless spills. One can also go through SSA, color in polynomial time, and place adequate permutations. This will save for free all moves along the dominance tree. But this may not be the best way because this can create split points with many moves to implement the permutations. A better way seems a) to design a cost model for permutation placement and edge splitting, b) to choose low-frequency potential split points to place permutations, and c) to color the graph with a coalescing-coloring algorithm, splitting points – and thus live-ranges – on the fly when necessary. In the worse case, the splitting will lead to a tree (but not necessarily the dominance tree) for which one can be sure that coloring with Maxlive registers is possible. The cost of moves can be further reduced by coalescing and permutation motion.

Designing and implementing such a coloring mechanism has still to be done in details. How to spill remains also a fundamental issue. Finally, it is also possible that a too tight spilling, with many points with k live variables, constrains too much the coalescing. In this case, it is maybe better to spill a bit more so as to balance the spill cost and the move cost. Such tradeoffs need to be evaluated with experiments before concluding. The same is true for edge splitting versus spilling, but possibly with less importance, as splitting an edge does not always imply introducing a jump.

6. ACKNOWLEDGMENTS

We would like to thank Christophe Guillon from STMicroelectronics, Grenoble, for his explanations on the ST200 core family and for the preliminary experiments done within the ST compiler.

7. REFERENCES

[1] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, pages 243–253, Snowbird, Utah, USA, 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] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. On the complexity of register coalescing. Technical Report RR2006-15, LIP, ENS-Lyon, France, April 2006.

[4] 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.

[5] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software: Practice and Experience*, 28(8):859–881, 1998.

[6] 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.

[7] Zoran Budimlić, Keith Cooper, Tim Harvey, Ken Kennedy, Tim Oberg, and Steve Reeves. Fast copy coalescing and live range identification. In *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation (PLDI'02)*, pages 25–32, Berlin, Germany, 2002. ACM Press.

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

[9] 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.

[10] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 174–187. Springer Verlag, 1998.

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

[12] 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.

[13] Alain Darte and Fabrice Rastello. Personal communication with Benoît Dupont de Dinechin, Jeanne Ferrante, and Christophe Guillon, 2002.

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

[15] 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.

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

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

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

[19] 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.

[20] Kathleen Knobe and Kenneth Zadeck. Register allocation using control trees. Technical Report No. CS-92-13, Brown University, 1992.

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

[22] Vincenzo Liberatore, Martin Farach-Colton, and Ulrich Kremer. Evaluation of algorithms for local register allocation. In *8th International Conference on Compiler Construction (CC'99), held as part of ETAPS'99*, volume 1575 of *Lecture Notes in Computer Science*, pages 137–152, Amsterdam, The Netherlands, March 1999. Springer Verlag.

[23] 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.

[24] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT'98)*, pages 196–204. IEEE Press, 1998.

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

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

[27] Fabrice Rastello, Francois de Ferrire, 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.

[28] 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.