Register Allocation: What Does the NP-completeness Proof of Chaitin et al. Really Prove? Or Revisiting Register Allocation: Why and How.

Florent Bouchez[1], Alain Darte[1], Christophe Guillon[2], and Fabrice Rastello[1]

[1] LIP, UMR CNRS-ENS Lyon-UCB Lyon-INRIA 5668, France.
[2] Compiler Group, ST/HPC/STS Grenoble, France.

Abstract. Register allocation is one of the most studied problems in compilation. It is considered NP-complete 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 this 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 use SSA to do register allocation in polynomial time, without contradicting Chaitin et al's NP-completeness result? To address this question and, more generally, the complexity of register allocation, we revisit Chaitin et al's proof to identify the interactions between spilling (load/store insertion), coalescing/splitting (removal/insertion of register moves), critical edges (property of the control flow), and coloring (assignment to registers). In particular, we show that, in general, 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 misinterpretation Chaitin et al's proof may suggest) but comes from critical edges and from the optimizations of spilling and coalescing.

1 Introduction

The goal of register allocation is to map the variables of a program into physical memory locations (main memory or machine registers). Accessing a register is usually faster than accessing memory, thus one tries to use registers as much as possible. When this is not possible, some variables must be transferred ("spilled") to and from memory. This has a cost, the cost of load/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 variables to registers) 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 several reasons to revisit these

approaches. First, some algorithms not considered in the past, because they were too time-consuming, can be good candidates today: processors used for compilation are now much faster and, for critical applications, industrial compilers are also ready to accept longer compilation times. Second, the increasing cost on most architectures of a memory access compared to a register access suggests to focus on heuristics that give more importance to spilling cost minimization, possibly at the price of additional register-to-register moves. Finally, there are many pitfalls and folk theorems concerning the complexity of the register allocation problem that are 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 in which two vertices/variables interfere if they are simultaneously live. As any graph is the interference graph of some program and because graph k-colorability is NP-complete [15, Problem GT4], heuristics have been used for spilling, coalescing, splitting, coloring, etc. The argument "register allocation *is* graph coloring, therefore it is NP-complete" is one of the first statements of many papers on register allocation. It 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 commonly believed that, in the absence of instruction rescheduling, it is NP-complete to decide if the program variables can be allocated to k registers with no spilling, even if live-range splitting is allowed. This is *not* what Chaitin et al. proved. We show that this problem is not NP-complete, except for a few particular cases that depend on the target architecture. This may be a 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, 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 (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 addressed 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 basic blocks. In this case, the coloring phase is of course easy because, after some variable renaming, the interference graph is an interval graph, but deciding *which* variables to spill and *where* is in general difficult. We completed this study for various spill cost models, and not just for basic blocks [2], and for several variants of register coalescing [3].

Today, most compilers go through an intermediate code representation, the (strict) SSA form (static single assignment) [12], 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 an SSA code is not arbitrary: it is chordal [18], thus easy to color. Furthermore, it can be colored with k colors iff (if and only if) Maxlive $\leq k$ where Maxlive is the maximal number of variables simultaneously live. What does this property imply? One can try to decompose register allocation into two phases. The first phase decides which values are spilled and where, to get a code with Maxlive $\leq k$. This phase is called

allocation in [22] as it decides the variables allocated in memory and the variables allocated in registers. The second phase, called *register assignment* in [22], maps variables to registers, possibly removing move instructions by *coalescing*, or introducing move instructions by *splitting*. These moves are also called *shuffle code* in [23]. When loads and stores are more expensive than moves, such an approach is worth exploring. It was 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, an interference graph is chordal iff it is the interference graph of a family of subtrees (here the live-ranges of variables) of a tree (here the dominance tree), see [18, Theorem 4.8]. 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. A direct proof of the chordality property for strict SSA programs can be given, see for example [2, 19].

Many papers [12, 5, 21, 28, 7, 27] address the problem of how to go out of SSA, in particular how to replace efficiently ϕ functions by move instructions. The issues addressed in these papers are how to handle renaming constraints, due to specific requirements of the architecture, how to deal with the so-called critical edges of the control-flow graph, and how to reduce the number of moves. 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 blocks, predecessors 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. However, minimizing the number of such permutations is NP-complete [19].

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., 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 that, when we can choose the splitting points, when we are free to add program blocks to remove critical edges (the standard *edge splitting* technique), then it is easy, except for a few particular cases, to decide if and how we can assign variables to registers without spilling.

More generally, the goal of this paper is to revisit Chaitin et al's proof to clarify the interactions between spilling, splitting, coalescing, critical edges, and coloring. Our study analyzes the complexity of the problem "are k registers enough to allocate variables without spill?" but, unlike Chaitin et al., we take into account live-range splitting. In Section 2, we analyze more carefully Chaitin et al's proof to show that the NP-completeness of the problem comes from critical edges. We then address the cases where critical edges can be split. In Section 3, we show how to extend Chaitin et al's proof to address the same problem as Pereira and Palsberg in [26]: if live-range splitting points are fixed at entry/exit of basic blocks and if register swaps are not available, the problem is NP-complete. In Section 4, we discuss the register swap constraint and show that, for most architecture configurations, if we can split variables wherever we want, the problem is polynomial. Section 5 summarizes our results and discusses how they can be used to improve previous approaches and to develop new allocation schemes.

2 Direct Consequences of Chaitin et al's NP-completeness Proof

Let us examine Chaitin et al's NP-completeness proof. The proof is by reduction from graph k-coloring [15, 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 well-known to be NP-complete if G is arbitrary, even for a fixed $k \geq 3$.

For the reduction, Chaitin et al. build a program with $|V| + 1$ variables, one for each vertex $u \in V$ and an additional variable x, as follows. 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 B_u and B_v. An entry block switches to all blocks $B_{u,v}$. Fig. 1 illustrates this construction when G is a cycle of length 4, the example used in [26]. The program is given on the right; its interference graph (upper-left corner) is the graph G (lower-left corner) plus a vertex for the variable x, connected to any other vertex. Thus x must use an extra color. Therefore G is k-colorable iff 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 [3]). In other words, each vertex of the interference graph represents the complete live-range as an atomic object and it is assumed that this live-range must always reside in the same register. Furthermore, 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.

[3] Actually, for Chaitin et al., two variables interfere only if one is live at the definition of the other. The two definitions coincide for *strict* programs, i.e., programs where any static control-flow path from the program start to a given use of a variable goes through a definition of this variable. This is the case for all the programs we manipulate in our NP-completeness proofs.

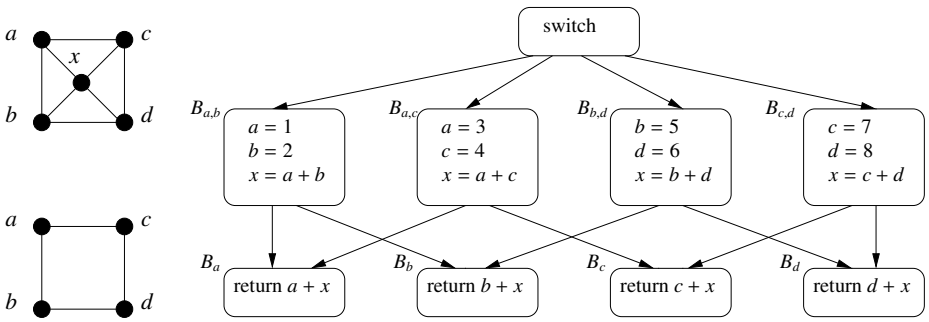

Fig. 1. Program built from a cycle of length 4 (bottom left) and its interference graph (top left).

This raises the question: What if we allow to split live-ranges? Consider Fig. 1 again and one of the variables, 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 $B_{a,c}$. 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 [11]. 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, we can interpret Chaitin et al's original proof as follows. It is NP-complete to decide if the program variables can be assigned to k registers, even if live-range splitting is allowed, but only when the program has critical edges that cannot be split, i.e., when we cannot change the control flow graph structure 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 Fig. 1) is not enough to prove anything about register allocation through SSA. Indeed, any program built this way can be allocated with only 3 registers by adding extra blocks, where out-of-SSA code is traditionally inserted, and performing some register-to-register moves in these blocks. We can place the variable definitions of each block of type $B_{u,v}$ in 3 registers (independently of other blocks), e.g., r_1 for u, r_2 for v, and r_3 for x, and decide that variables u and x in each block of type B_u are always expected in registers r_1 and r_3. Then, we can "repair" the coloring at each join point, when needed, thanks to an adequate re-mapping of registers (here a move from r_2 to r_1) in a new block along the edge from $B_{u,v}$ to B_u. We will use a similar block structure later, see Fig. 2.

More generally, when there are no critical edges, one can indeed go through SSA (or any live-ranges representation as subtrees of a tree), i.e., consider that different variable definitions belong to different live-ranges, and to color them with k colors, if possible. This can be done in linear time, in a greedy fashion, because the corresponding interference graph is chordal. At this stage, it is easy to decide if k registers are enough. This is possible iff 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 maximal size of a clique in the interference graph (in general, it is only a lower bound); b) the maximal size of a clique and the chromatic number of the graph are equal (the graph is chordal). Moreover, if k registers are not enough, additional splitting will not help as this leaves Maxlive unchanged.

If k colors are enough, it is still possible that the colors selected under SSA do not match at join points where live-ranges were split. Some "shuffle" [23], i.e., registers permutation, is needed along the edge where colors do not match. Because the edge is not critical, the shuffle will not propagate along other control flow paths. If some register is available at this point, any remapping can be performed as a sequence of register-to-register moves, possibly using the free register as temporary storage. Otherwise, an additional register is needed unless one can perform register swaps, either with a special instruction or with arithmetic operations such as XOR (but maybe only for integer registers). This view of coloring through permutations insertion is the base of any approach that optimizes spilling first [20, 1, 2, 19]. Some spilling and splitting are done, optimally or not, to reduce 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 solve spilling optimally, and there is a potential permutation between any two program points. Then, live-ranges are merged back, as much as possible, thanks to coalescing.

Thus, it seems that if we go through SSA (for example but not only), deciding if k registers are enough becomes easy. The only possible remaining difficult case is if register swaps are not available. Indeed, in this case, no permutation except the identity can be performed at a point with k live variables. 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 is needed? The answer is no, the problem is NP-complete. Pereira and Palsberg use a reduction from the k-colorability problem for circular-arc graphs, which is NP-complete if k is a problem input [16]. Basically, the idea is to start from a circular-arc graph, to cut all arcs at some point to get an interval graph, to view 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 k variables are live 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 technique used in [16] 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, it is difficult to choose the right coloring of the SSA representation and thus decide if k registers are enough. It is NP-complete even for a simple loop and a single split point. However, if k is fixed, this specific problem is polynomial as is the k-coloring problem of circular-arc graphs, by propagating possible

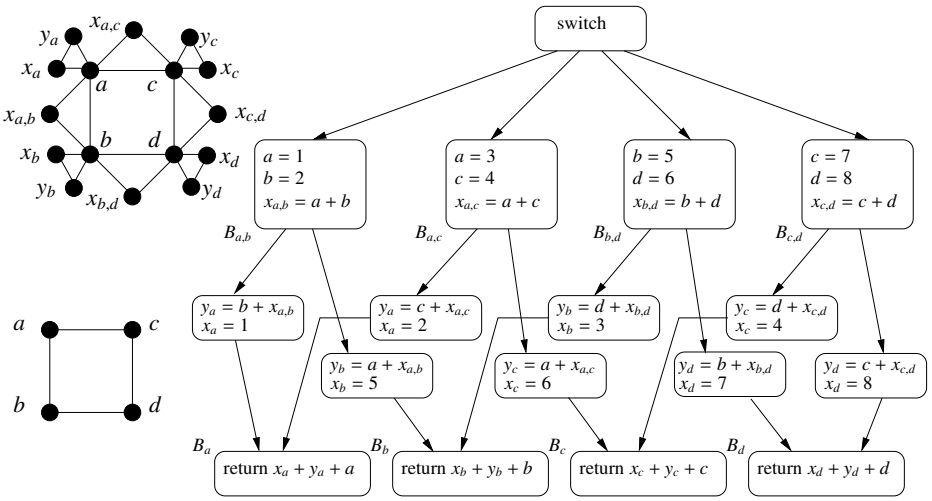

Fig. 2. Program built from a cycle of length 4 (bottom left) and its interference graph (top left).

permutations. We now show that, with a simple variation of Chaitin et al's proof, we can get a similar NP-completeness result, even for a fixed k, but for an arbitrary program.

Given an arbitrary graph $G = (V, E)$, we build a program with the same control-flow structure as for Chaitin et al's construction, but we split critical edges as shown in Fig. 2. 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 $(u, v) \in E$. 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 is 3-colorable iff G itself is 3-colorable. Its restriction to variables u (those that correspond to vertices of G) is exactly G. Fig. 2 illustrates this construction, again when G is a cycle of length 4. The interference graph is in the upper-left corner.

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 iff 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, except the identity, can be done. Thus 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 Fig. 2 again to visualize the cliques of size 3). This gives a valid register assignment.

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 in traditional out-of-SSA translation), it is NP-complete to decide if k registers are enough. This is true 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, for example in the middle of a block if this allows us to perform a permutation? Consider Fig. 2 again. The register pressure is 3 on any control-flow edge, this was the key for the proof of Section 3. But it is not 3 everywhere; between the definitions of each y_u and each x_u, it drops to 2. Here, some register-to-register moves can be inserted to permute 2 colors and, thanks to this, 3 registers are always enough for such a program. One can color independently the top (including the variables y_u) and the bottom (including the variables x_v), then place permutations between the definitions of 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 3 different colors, arbitrarily. For each block B_u, do the same for u, x_u, and y_u (i.e., define 3 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 for $B_{u,v}$, 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 some moves may be needed for the values u and y_u, after the definition of y_u, and before their uses in B_u, if colors do not match. But, between the definitions of y_u and x_u, only 2 registers contain a live value: one contains u defined in $B_{u,v}$ and one contains y_u. These 2 values can thus be moved to the registers where they are supposed to be in B_u, with at most 3 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 Fig. 3: on the left, the previous situation with a register pressure drop to 2, in the middle, a situation with a constant register pressure equal to 3, but that does not keep the equivalence with graph 3-coloring. However, if instructions can define more than one value, it is easy to modify the proof. To define x_u and y_u, use a statement $(x_u, y_u) = f(v, x_{u,v})$ that consumes v and $x_{u,v}$ and produces y_u and x_u *simultaneously*, as depicted on the right of Fig. 3. Now, the register pressure is 3 everywhere in the program and thus G is 3-colorable iff the program can be mapped to 3 registers (and, in this case, with no live-range splitting). Thus, it is NP-complete to decide if k registers are enough *if two variables can be created simultaneously by a machine instruction* but swaps are not available, even if there is no critical edge and if we can split wherever we want. Such an instruction should consume at least 2 values, otherwise, the register pressure is lower just before and a permutation can be placed. Notice the similarity with circular-arc graphs: as mentioned in [16], coloring circular-arc graphs remains NP-complete even if at most 2 circular arcs start at any point (but not if only one can start).

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

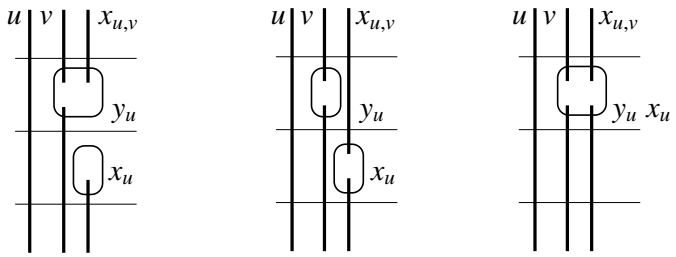

Fig. 3. Three cases: register pressure drops to 2 (on the left), is constant to 3 (middle and right).

and k registers are enough iff Maxlive $\leq k$. Thus, it remains to consider only one case: what if *at most 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 At Most One Definition at a Time

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

The *register pressure* at a point is the number of variables live at this point; Maxlive is the maximal register pressure in the program. If Maxlive $> k$, it is not possible to assign the variables of a strict program to k registers without spilling, as two simultaneously live variables interfere.[4] If Maxlive $< k$, it is always possible to assign variables to k registers by splitting live-ranges and performing adequate permutations. The same occurs for a point with register pressure $< k$, as explained in Section 3: a color mismatch can always be repaired by an adequate permutation, thanks to an available register. Thus, for a strict program, the only problem may come from the sequences of program points where the register pressure remains equal to k. But, unlike Section 4.1 where the degree of freedom (at least 2) to choose colors leads to NP-completeness, here, the fact that at most *one* variable can be defined at a time simplifies the problem. It does not mean that k registers are always enough, but it is easy to decide if this is the case. To explain this situation precisely, we need to define more formally what we mean by *color propagation*.

Liveness analysis defines, for each instruction s, $live_in(s)$ and $live_out(s)$, the set of variables live just before and just after s. We color these sets locally, propagating the colors from instruction to instruction, i.e., coloring variables in neighbor sets with the same color, following the control-flow, possibly backwards or forwards, i.e., considering the control-flow as undirected. More formally, coloring a statement s means defining two injective maps $col_in(s)$ (resp. $col_out(s)$) from $live_in(s)$ (resp. $live_out(s)$) to $[1..k]$. When we propagate colors from a statement s_1 to a statement s_2, forwards, we define $col_in(s_2)$ so that $col_in(s_2)(x) = col_out(s_1)(x)$ for all $x \in live_in(s_2) \cap live_out(s_1)$ and we pick different colors for the other variables, arbitrarily. Then, we do the same to define $col_out(s_2)$ from $col_in(s_2)$. When we propagate backwards, the situation is

[4] Notice that it is not true for a *non-strict* program. We leave this case open.

symmetric; we define $col_out(s_2)$ from $col_in(s_1)$, then $col_in(s_2)$ from $col_out(s_2)$. Below, when we explain the effect of propagation, we assume the propagation is forwards; otherwise one just needs to exchange the suffixes "in" and "out".

We first restrict to the subgraph of the control-flow graph defined by the program points where the register pressure is equal to k, i.e., we only propagate between two instructions s_1 and s_2 such that both $live_out(s_1)$ and $live_in(s_2)$ have k elements. 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, except possibly for the sets $live_out(s_2)$ where the propagation stops. 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, as defined above, backwards and forwards along the control flow until all points are reached. In this process, there is no ambiguity to choose a color. First, there is no choice for defining $col_in(s_2)$ from $col_out(s_1)$ as $live_out(s_1) = live_in(s_2)$: indeed they both have k elements and, since there is no critical edge, either $live_out(s_1) \subseteq live_in(s_2)$ or the converse. Second, if $live_out(s_2)$ has k elements, then either $live_out(s_2) = live_in(s_2)$ or, as s_2 defines at most one variable, there is a unique variable in $live_out(s_2) \setminus live_in(s_2)$ and a unique variable in $live_in(s_2) \setminus live_out(s_2)$: they must have the same color. Thus, there is no choice for defining $col_out(s_2)$ from $col_in(s_2)$ either. Therefore, for each connected component, going backwards and forwards defines, if it exists, a *unique* solution up to the initial permutation of colors. In other words, if there is a solution, we can define it, for each connected component, by propagation. Moreover, 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 color propagation on each connected component is possible, then k registers are enough for the whole program. Indeed, we can color the rest (with register pressure $< k$) in a greedy (but not unique) fashion. When we reach a point already assigned, we can repair a possible color mismatch: we place an adequate permutation of colors between s_1 and s_2, in the same basic block as s_1 if s_2 is the only successor (resp. predecessor for backwards propagation) of s_1 or in the same basic block as s_2 if s_1 is the only predecessor (resp. successor) of s_2. This is always possible because there is no critical edge and there are at most $(k-1)$ live variables at this point. To summarize, to decide if k registers suffice when Maxlive $\leq k$ (and color when possible), we first propagate colors, following the control flow, along program points where the register pressure is exactly 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 propagation phase, along all remaining program points. If we reach a program point already colored and the colors do not match, we solve the problem with a permutation of at most $(k-1)$ registers, using an extra available register. We point out that we can also do the propagation in a unique phase, as long as we propagate in priority along points where the register pressure is exactly k. A work list can be used for this purpose.

4.3 Subtleties of the Architectures

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

As for many processors, some variables need to be assigned to specific registers: they are *precolored*. Such precoloring constraints do not change the complexity of deciding if some spilling is necessary. Indeed, when swaps are available, Maxlive $\leq k$ is still the condition to be able to color with k registers as 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 polynomial case, i.e., when no register swap is available and instructions define 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 r_x++, i.e., $r_x = r_x + 1$. An instruction x++ apparently prevents the coloring under SSA as x is redefined, unless first rewritten as $y = x + 1$. A coalescer may then succeed in coloring y and x the same way. To enforce this coloring, one can also simply ignore this redefinition of x and consider that the live-range of x goes through the instruction, as a larger SSA-like live-range.

The possible NP-complete cases are 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 instruction can create more than one variable: the function calls that can have up to 8 results and the 64 bits load operations that load 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, the set of registers used for the results is a *strict* subset of the set of caller-save registers. Thus, just after the call and before the possible reloads of caller-saved registers, at least one caller-save register is available to permute colors if needed. The situation is similar before the call, reasoning with parameters. 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. What about 64 bits loads, such as $r_x, r_y = \text{load}(r_z)$? Such a load instruction has only one argument on the ST200. So, if the number of live variables is k after defining r_x and r_y, it is $< k$ just before, so a permutation can be done if needed. We can ensure that r_x and r_y are consecutive thanks to a permutation just before the load 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 architectural subtleties, the NP-complete cases where deciding if some spilling is necessary seem to be quite artificial. In practice, swaps can usually be done and one just has to check Maxlive $\leq k$.

If not, one can rely on a greedy coloring, propagating only along program points where the register pressure is 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 (just before or just after) with $< k$ live variables. 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 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., if each variable can be assigned to only one register. We showed that the same construction proves more: the problem remains NP-complete when live-range splitting is allowed but not (critical) edge splitting.

Recently, Pereira and Palsberg [26] proved that, if the program is a simple loop, the problem is NP-complete if live-range splitting is allowed but only on a block on the back edge and register swaps are not available. This is a particular form of register allocation through SSA. The problem is NP-complete if k is a problem input. We showed that Chaitin et al's proof can be extended to show a bit more. When register swaps are not available, the problem is 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 at most 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 the reduction of Chaitin et al. 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 (if k registers are not enough) and 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, i.e., when one considers better to avoid spilling at the price of register-to-register moves. 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/load operations should be placed 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. However, for some applications, when the register pressure is high, we noticed some possible improvements in terms of spill cost. 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, i.e., a variable such that, 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/iterated register coalescing [17], as coalescing can reduce the number of neighbors of a vertex in 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 remove it from the set. We repeat this process 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. 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 spilling phases needed for a RISC machine: for a RISC machine, a spilled live-range leaves small live-ranges to perform the reloads, and a Chaitin-like approach needs to iterate. 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 at once.

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 ensure that coloring with Maxlive registers is possible. A possibility is to split all critical edges and to color in polynomial time with Maxlive colors, possibly inserting color permutations. The extreme way to do so 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]. Such a strategy leads to a valid k-coloring but of course with an unacceptable number

of moves. This can be improved thanks to coalescing, although it is in general NP-complete [3]. As there are many moves to remove, possibly with tricky structures, a conservative approach does not work well, and an optimistic coalescing seems preferable [24, 1]. Another way is to color independently each basic block, with Maxlive colors, 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 all moves along the dominance tree, but this may not be the best way because this can create split points with many moves. 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, severely limits the coalescing mechanism. 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.

Acknowledgments

The authors would like to thank Philip Brisk, Keith Cooper, Benoît Dupont de Dinechin, Jeanne Ferrante, Daniel Grund, Sebastian Hack, Jens Palsberg, and Fernando Pereira, for interesting discussions on SSA form and register allocation.

References

1. A. W. Appel and L. George. Optimal spilling for CISC machines with few registers. In *Proceedings of PLDI'01*, pages 243–253, Snowbird, Utah, USA, June 2001. ACM Press.
2. F. Bouchez, A. Darte, C. Guillon, and F. Rastello. Register allocation and spill complexity under SSA. Technical Report RR2005-33, LIP, ENS-Lyon, France, Aug. 2005.
3. F. Bouchez, A. Darte, C. Guillon, and F. Rastello. On the complexity of register coalescing. Technical Report RR2006-15, LIP, ENS-Lyon, France, Apr. 2006.
4. P. Briggs, K. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.
5. P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software: Practice and Experience*, 28(8):859–881, 1998.

6. P. Brisk, F. Dabiri, J. Macbeth, and M. Sarrafzadeh. Polynomial time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*, June 2005.
7. Z. Budimlić, K. Cooper, T. Harvey, K. Kennedy, T. Oberg, and S. Reeves. Fast copy coalescing and live range identification. In *Proc. of PLDI'02*, pages 25–32, 2002. ACM Press.
8. G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of Compiler Construction (CC'82)*, volume 17(6) of *SIGPLAN Notices*, pages 98–105, 1982.
9. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, Jan. 1981.
10. K. D. Cooper and L. T. Simpson. Live range splitting in a graph coloring register allocator. In *Proceedings of Compiler Construction (CC'98)*, volume 1383 of *LNCS*, pages 174–187. Springer Verlag, 1998.
11. R. Cytron and J. 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, Aug. 1987.
12. 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.
13. A. Darte and F. Rastello. Personal communication with Benoît Dupont de Dinechin, Jeanne Ferrante, and Christophe Guillon, 2002.
14. M. Farach-Colton and V. Liberatore. On local register allocation. *Journal of Algorithms*, 37(1):37–65, 2000.
15. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
16. M. R. Garey, D. S. Johnson, G. L. Miller, and C. H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM J. of Algebr. Discrete Methods*, 1(2):216–227, 1980.
17. L. George and A. W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.
18. M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.
19. S. Hack, D. Grund, and G. Goos. Register allocation for programs in SSA-form. In *Proceedings of Compiler Construction (CC'06)*, volume 3923 of *LNCS*. Springer Verlag, 2006.
20. K. Knobe and K. Zadeck. Register allocation using control trees. Technical Report No. CS-92-13, Brown University, 1992.
21. A. Leung and L. George. Static single assignment form for machine code. In *Proceedings of PLDI'99*, pages 204–214. ACM Press, 1999.
22. V. Liberatore, M. Farach-Colton, and U. Kremer. Evaluation of algorithms for local register allocation. In *Proceedings of Compiler Construction (CC'99)*, volume 1575 of *LNCS*, pages 137–152, Amsterdam, 1999. Springer Verlag.
23. G.-Y. Lueh, T. Gross, and A.-R. Adl-Tabatabai. Fusion-based register allocation. *ACM Transactions on Programming Languages and Systems*, 22(3):431–470, 2000.
24. J. Park and S.-M. Moon. Optimistic register coalescing. In *Proceedings of PACT'98*, pages 196–204. IEEE Press, 1998.
25. F. M. Q. Pereira and J. Palsberg. Register allocation via coloring of chordal graphs. In *Proceedings of APLAS'05*, pages 315–329, Tsukuba, Japan, Nov. 2005.
26. F. M. Q. Pereira and J. Palsberg. Register allocation after classical SSA elimination is NP-complete. In *Proceedings of FOSSACS'06*, Vienna, Austria, Mar. 2006.
27. F. Rastello, F. de Ferrière, and C. Guillon. Optimizing translation out of SSA using renaming constraints. In *Proceedings of 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 *LNCS*, pages 194–210. Springer Verlag, 1999.