The Complexity of Register Allocation

Philipp Klaus Krause
philipp@informatik.uni-frankfurt.de
Goethe-Universität
Institut für Informatik
Robert-Mayer-Straße 11-15
60325 Frankfurt am Main

Abstract

In compilers, register allocation is one of the most important stages with respect to optimization for typical goals, such as code size, code speed, or energy efficiency. Graph theoretically, optimal register allocation is the problem of finding a maximum weight \( r \)-colorable induced subgraph in the conflict graph of a given program. The parameter \( r \) is the number of registers.

Large classes of programs are structured, i.e. their control-flow graphs have bounded tree-width [1, 2, 3]. The decision problem of deciding if a conflict graph of a structured program is \( r \)-colorable is known to be fixed-parameter tractable [4]. Optimal register allocation for structured programs is known to be in XP [5].

We complement these results by showing that optimal register allocation parametrized by \( r \) is \( W[\text{SAT}] \)-hard. This even holds for programs using only if/else and while as control structures; these programs form a subclass of the structured programs.

Keywords: Register allocation, parametrized complexity, structured program

1. Introduction

Register allocation is a compiler stage that tries to assign variables in a computer program to hardware registers in a processor. Variables that are alive at the same time (conflicting variables) cannot be assigned to the same register, since this would result in values that are still needed being overwritten. Variables that are not assigned to registers are stored in main memory instead, which typically is slower by several orders of magnitude, and takes more or longer instructions to access. Register allocation is one of the most important

\footnote{This research was supported by DFG-Projekt GalA, grant number AD 411/1-1. It was partially conducted at NII, Tokyo, supported by a fellowship within the FIT-Programme of the German Academic Exchange Service (DAAD)}
stages in a compiler with respect to optimization for typical goals, such as code size, code speed or energy efficiency.

Register allocation can be seen as coloring the conflict graph of the variables of the program, with colors being the available registers. For \( r \) registers, finding an \( r \)-colorable induced subgraph of maximum weight in the conflict graph is a simplification of the the register allocation problem.

Large classes of programs are structured [1, 2, 3]. The problem of deciding if a conflict graph of a structured program is \( r \)-colorable is known to be fixed-parameter tractable [4]. Optimal register allocation for structured programs is known to be in XP [5].

We prove that finding an \( r \)-colorable induced subgraph of maximum weight in the conflict graph is \( W[\text{SAT}] \)-hard, even for a subclass of structured programs. This is a negative result complementing the earlier positive results.

The following Section 2 introduces the basic concepts necessary for the discussion of the related work in Section 3 and our results in Section 4. Section 5 concludes and states some questions that are still open.

2. Preliminaries

**Definition 1 (Graph).** A graph \( G \) is a pair of two sets, the node set and the edge set. For an undirected graph the edge set contains sets of two distinct nodes. For a directed graph, the edge set contains pairs of nodes.

**Definition 2 (Program).** A program consists of a directed graph \( G \), called the control-flow graph (CFG) of the program, a set of variables \( V \) and a weight function \( c : V \to [0, \infty] \). Each node of \( G \) is marked by a subset of \( V \). The set of nodes of \( G \) that are marked by a variable \( v \in V \) are the live-range of \( v \); \( v \) is said to be alive there. A live-range induces a connected subgraph of \( G \).

This representation can be easily generated from other representations, such as pseudo-code. The nodes of the CFG are the program’s instructions; there is an edge from \( i \) to \( j \) if there is some execution of the program where instruction \( j \) is executed directly after instruction \( i \). Typically there is a cost (code size, runtime, energy consumption) associated with not placing variables in registers. The cost depends on how often the variable is accessed in the program. This is represented in the weight function.

Figure 1 shows code and CFG of a program and corresponding live-ranges.

**Definition 3 (Intersection Graph).** Let \( S_I \) be a family of sets for some index set \( I \). The intersection graph of \( S_I \) is the undirected graph that has a node for each \( i \in I \), and two nodes are connected by an edge, if the intersection of the corresponding sets is nonempty:

\[
\{(v_i | i \in I), \{\{v_i, v_j\} | S_i \cap S_j \neq \emptyset\}\}.
\]

**Definition 4 (Conflict Graph).** Let \( V \) be the set of variables of a program. The conflict graph of the program is the intersection graph of their live-ranges.
int binomial_coefficient(int n, int k) {
    int i, delta, max, c;
    if (n < k)
        return(0);
    if (n == k)
        return(1);
    if (k < n - k) {
        delta = n - k;
        max = k;
    } else {
        delta = k;
        max = n - k;
    }
    c = delta + 1;
    for (i = 2; i <= max; i++)
        c = (c * (delta + i)) / i;
    return(c);
}

We use $G$ for the control-flow graph and $V$ for the variables throughout.

**Definition 5 (Series-Parallel Graph).** A two-terminal graph (TTG) is a graph with two distinguished nodes, source $s$ and sink $t$. The parallel composition of two TTGs can be obtained by taking their disjoint union, and then merging the two sources into the new source and merging the two sinks into the new sink. The series composition of two TTGs $X$ and $Y$ can be obtained by taking their disjoint union, and then merging the sink of $X$ with the source of $Y$. The source for $X$ and the sink of $Y$ become source and sink of the new graph. The graphs that can be obtained from the TTGs that have a single edge connecting source and sink by doing parallel and series compositions, and forgetting about the distinction of source and sink are the *series-parallel* graphs.

Tree decompositions [6] have commonly been used to find polynomial algorithms on restricted graph classes for many problems that are hard on general graphs, since their rediscovery by Robertson and Seymour [7]. We need them to be able to discuss some related work.

**Definition 6 (Tree decomposition).** Given a graph $G = (\Pi, K)$ a *tree decomposition* of $G$ is a pair $(T, \mathcal{X})$ consisting of a tree $T$ and a family $\mathcal{X} = \{ X_i \mid i \text{ node of } T \}$ of subsets of $\Pi$ with the following properties:
• $\bigcup_{i \text{ node of } T} X_i = \Pi$,
• For each $\{x, y\}$ edge in $K$, there is a node $i$ of $T$, such that $x, y \in X_i$,
• The subgraph of $T$ induced by $\{i \text{ node of } T \mid x \in X_i\}$ is connected for each $x \in \Pi$.

The width of a tree decomposition $(T, \mathcal{X})$ is $\max\{|X_i| \mid i \text{ node of } T\} - 1$. The tree-width $tw(G)$ of a graph $G$ is the minimum width of all tree decompositions of $G$.

Series-parallel graphs have tree-width at most 2.

**Definition 7 (Structured Program).** Let $k \in \mathbb{N}$ be fixed. A program is called $k$-structured, iff its control-flow graph $G$ has tree-width at most $k$.

Different notions of “structured program” exist. Many of them, including famous ones, are rather informal or not directly related to the tree-width of control-flow graphs [8]. We now take a closer look at some formal definitions of “structured program”. Under an early one, it meant programs using only if/else and while as control structures [9]. These are a subclass of the programs that have a series-parallel control-flow graph, and later “structured program” has been used to denote all programs that have a series-parallel control flow graph [10]. Some define “structured program” as a program that does not contain goto statements [4]. More recently, it denotes programs that are $k$-structured for a $k$ fixed in advance [5]. Programs that have a series-parallel control flow graph are 2-structured, and programs that use only if/else and while as control structures have series-parallel control-flow graphs. A discussion of how not containing goto statements relates to being $k$-structured for various programming languages can be found in Section 3. Our hardness result holds even for the early, most restrictive notion of “structured program”.

In $k$-structured programs the conflict graph is the intersection graph of connected subgraphs of a graph of tree-width at most $k$.

**Definition 8 (ORA).** Given an input program and parameter $r$, the number of registers, the optimization problem of register allocation (ORA) is to find an $r$-colorable induced subgraph $S$ in the conflict graph, such that the sum $\sum_{v \in V \setminus V(S)} c(v)$ of the costs of the variables outside this subgraph is minimized.

The subgraph $S$ is induced by the variables, that will be placed in registers by optimal allocation.

This Definition 8 is a simplification of the problem encountered in real-world register allocation. It does not capture aspects such as different register classes, register preferences, register aliasing, coalescing, rematerialization or rescheduling. However, since we present a hardness result it is sufficient.

The decision problem corresponding to ORA is the following:
Definition 9 (DRA). Given an input program and parameter \( r \), the number of registers and a number \( g \), the decision problem of register allocation (DRA) is to decide if there is an \( r \)-colorable induced subgraph \( S \) in the conflict graph, such that the sum \( \sum_{v \in V \setminus V(S)} c(v) \) of the costs of the variables outside this subgraph is at most \( g \).

Note that computationally, the optimization problem ORA is at least as hard as the decision problem DRA.

We use standard terminology from parametrized complexity theory [11, 12]:

Definition 10 (Parametrized Problem). A parametrized problem over a finite alphabet \( \Sigma \) is a set of pairs \( (x, k) \) with \( x \in \Sigma^* \) and \( k \in \mathbb{N} \).

DRA above is an example of a parametrized problem under this definition.

Definition 11 (Parametrized Reduction). Let \( L_1, L_2 \subseteq \Sigma^* \times \mathbb{N} \) be two parametrized problems. We say that \( L_1 \) reduces to \( L_2 \) by a parametrized reduction, if there are functions \( k \mapsto k' \) and \( k \mapsto k'' \) from \( \mathbb{N} \) to \( \mathbb{N} \) and a function \( (x, k) \mapsto x' \) from \( \Sigma^* \times \mathbb{N} \) to \( \Sigma^* \) such that

- \( (x, k) \mapsto x' \) is computable in time \( k''|\langle x, k \rangle|^c \) for some constant \( c \),
- \( (x, k) \in L_1 \iff (x', k') \in L_2 \).

Definition 12 (Circuit). A circuit is a directed, acyclic graph, with nodes representing logical gates (or, and, not). Not-gates have in-degree one, or- and and-gates have in-degree two or more. There is exactly one node of out-degree zero, called the output, and there are nodes of in-degree zero, called inputs.

The circuit computes a Boolean function in the natural way.

The weft of a circuit is the maximum number of gates of in-degree more than two on a path from input to output. The depth of a circuit is the maximum number of gates on a path from input to output.

A circuit is called a SAT-circuit, iff the undirected subgraph induced by all nodes that are not input nodes is a tree.

Except for the definition of \( W[t] \) below, weft and depth are irrelevant in this work, so we can assume that all or- and and-gates have in-degree two by replacing the gates of in-degree more than two by trees of gates of in-degree two.

Definition 13 (WCS). Given a circuit \( C \) (input) and an integer \( k \) (parameter) the weighted circuit satisfiability problem asks if there is a satisfying assignment for \( C \) of weight exactly \( k \) (i.e. exactly \( k \) of the inputs are set to “true” in the assignment).

Definition 14 (Parametrized Complexity Classes).

- FPT is the class of all parametrized problems parametrized by \( k \) that can be solved in time \( f(k)p(n) \) with input size \( n \), with a computable function \( f \) and a polynomial \( p \).
Let \( t \) be a positive integer. \( W[t] \) is the class of all parametrized problems that can be reduced to the WCS problem on weft \( t \), depth \( d \) SAT-circuits by a parametrized reduction, where \( d \geq 1 \) is a constant.

\( W[SAT] \) is the class of all parametrized problems that can be reduced to the WCS problem on SAT-circuits by a parametrized reduction.

\( W[P] \) is the class of all parametrized problems that can be reduced to the WCS problem by a parametrized reduction.

\( XP \) is the class of all problems parametrized by \( k \) that can be solved in time \( f(k, n) \), with input size \( n \) and \( f \) polynomial in \( n \) for fixed \( k \).

\[
\text{FPT} \subseteq W[1] \subseteq W[2] \subseteq \ldots \subseteq W[SAT] \subseteq W[P] \subseteq XP.
\]

3. Previous results

Any graph can occur as the conflict graph of some program [13]. Together with the NP-hardness of graph-coloring [14] this proves the NP-hardness of register allocation when the input is not restricted to \( \ell \)-structured programs for some \( \ell \). Garey et al. [15] have proven that when the number \( r \) of registers is part of the input (i.e. not a parameter) the register allocation problem DRA is NP-hard even for structured programs. Kannan and Proebsting [10] were able to approximate a simplified version of the register allocation problem within a factor of 2 for programs that have series-parallel control-flow graphs (a subclass of 2-structured programs).

Large classes of programs are \( \ell \)-structured for some \( \ell \). Programs written in Algol or Pascal are \((2 + g)\)-structured, if the number of labels targeted by goto statements per function does not exceed \( g \). Modula-2 programs are 5-structured [1]. Programs written in C are \((7 + g)\)-structured if the number of labels targeted by gotos per function does not exceed \( g \) [1, 16]. Similarly, Java programs are \((6 + g)\)-structured if the number of labels targeted by labeled breaks and labeled continues per function does not exceed \( g \) [2]. Ada programs are \((6 + g)\)-structured if the number of labels targeted by gotos and labeled loops per function does not exceed \( g \) [3]. Coding standards tend to place further restrictions, resulting e.g. in C programs being 5-structured when adhering to the widely adopted MISRA-C:2004 [17] standard [16].

Based on earlier work by Thorup [1], Bodlaender et al. [4] obtained the following result:

**Theorem 1.** Let \( \ell \) be fixed. For \( \ell \)-structured programs, deciding if the register allocation problem has a solution such that \( \sum_{v \in V \setminus V(S)} c(v) = 0 \), (i.e. all variables can be placed in registers) is in FPT when parametrized by the number \( r \) of registers. I.e. DRA is in FPT for \( g = 0 \).

Krause [5] obtained the following result for the optimization problem:
Theorem 2. Let $t$ be fixed. For $t$-structured programs, the register allocation problem ORA can be solved in time $O(|G||V|^2(\text{tw}(G)+1)r)$, and thus is in XP.

This result even holds when taking into account further aspects of register allocation such as different register classes, register preferences, register aliasing, coalescing and rematerialization.

4. $W[\text{SAT}]$-hardness of register allocation

We prove the hardness by constructing a program from the circuit in a rather natural way. The following Lemmata 1 and 2 and their proofs allow us to construct program parts from the individual gates in the circuit. Lemma 3 and its proof allow us to combine them into a program for the whole circuit.

Lemma 1. For every $r \geq 3$, there is a structured program, such that there are variables $A, \overline{A}, B, \overline{B}, Y, \overline{Y}$, and when doing optimal register allocation for the program, exactly one of $X, \overline{X}$ is placed in registers for $X = A, B, Y$. Furthermore, $Y$ is placed in a register exactly if $A$ placed in a register or $B$ is placed in a register.

Proof. To abbreviate, and guide intuition we will say that a variable is “true”, if it is placed in a register. Note that this notion of “true” is completely unrelated to any value the variable might hold in the program at run-time.

We construct a conflict graph with cliques as in Figure 2(a). Each clique (represented by a grey shape in the figure) is filled by additional, unnamed nodes, so that it has size exactly $r + 1$. Thus e.g. the middle clique containing $A, B$ and $\overline{Y}$ has $r - 2$ other nodes. These other nodes, each of which belongs to exactly one such added clique, are given a huge weight. Thus in each optimal assignment of variables to registers all unnamed variables will be placed in registers. Since each clique has size $r + 1$ this means that at least one of the named nodes in each clique will not be placed in a register. For the clique containing $A$ and $\overline{A}$ it means that at most one of them goes into a register. The
same hold for the clique containing $B$ and $\overline{B}$ and the clique containing $Y$ and $\overline{Y}$. The clique containing $A$ and $\overline{Y}$ and the one containing $B$ and $\overline{Y}$ work the same way. The middle clique ensures that the value “true” can be assigned to at most two out of $A, B$ and $Y$. Since positive weights are assigned to all named variables, an optimal assignment will place as many of the variables in registers, as possible. It is possible to place half of them in registers (since $r \geq 3$), thus, for an optimal assignment, the following hold:

- At most one of $A$ and $A$ is “true”.
- At most one of $B$ and $B$ is “true”.
- At most one of $Y$ and $Y$ is “true”.
- At most one of $A$ and $Y$ is “true”.
- At most one of $B$ and $Y$ is “true”.
- At most two of $A, B, Y$ are “true”.
- At least three of $A, A, B, B, Y, Y$ are “true”.

As one can easily verify, this corresponds to $Y$ being “true” exactly if $A$ is “true” or $B$ is “true”.

Now that we have a conflict graph that gives us the desired properties, we need to show that this conflict graph can occur in a structured program. Figure 2(b) shows live ranges, that result in the cliques just discussed in the conflict graph; the narrow, short lines represent the unnamed variables. As can be seen in Figure 2(c), we get a star-shaped control-flow graph. The dashed lines represent paths where the additional, unnamed variables of the cliques are alive.

C-like pseudo-code for this or-gate could look like Figure 3. The cases of the switch statement correspond to the indices in Figure 2(c). The macro CLIQUE1 expands to code that has $r - 1$ additional variables that, together with the other variables alive, form a clique of size $r + 1$ in the conflict graph, and accesses them as often as necessary to give them the desired weight. CLIQUE2 does the same for $r - 2$ additional variables to form a clique of size $r + 1$. Since the only control construct used is a switch statement (which could be easily replaced by if/else), the program is structured. The assignments to the variables $A, \overline{A}, B, \overline{B}$ in the cases 0 to 2 of the switch statement only serve to overwrite these variables to ensure that they are not alive after the CLIQUE1/CLIQUE2 macro. Cases 3 and 4 are where other code is inserted later in the proof of Lemma 3. q. e. d.

Lemma 2. For every $r \geq 3$, there is a structured program, such that there are variables $A, \overline{A}, B, \overline{B}, Y, \overline{Y}$, and when doing optimal register allocation for the program, exactly one of $X, \overline{X}$ is placed in registers for $X = A, B, Y$. Furthermore, $Y$ is placed in a register exactly if $A$ and $B$ are placed in a register.

Proof. The proof is similar to the one of Lemma 1; see also Figure 4. q. e. d.
signal_y:
CLIQUE:
$A = f(), \overline{A} = f(), B = f(), \overline{B} = f();$
switch ($f() \mod 5$)
{
case 0:
    CLIQUE2;
g(Y, A, B);
    $A = f(), \overline{A} = f(), B = f(), \overline{B} = f();$
    break;
case 1:
    CLIQUE1;
g(Y, A);
    $A = f(), \overline{A} = f(), B = f(), \overline{B} = f();$
    break;
case 2:
    CLIQUE1;
g(Y, B);
    $A = f(), \overline{A} = f(), B = f(), \overline{B} = f();$
    break;
case 3:
    CLIQUE1;
    break; // Place goto signal_a; or nest here.
case 4:
    CLIQUE1;
    break; // Place goto signal_b; or nest here.
}

Figure 3: Pseudocode for OR from Figure 2

Figure 4: AND
Lemma 3. For every circuit, we can construct a program, such that the solution to the register allocation problem DRA corresponding to the program gives a solution to the WCS for the circuit. The tree-width of the control-flow graph of the constructed program is at most one bigger than the tree-width of the underlying graph of the circuit without input nodes. For SAT-circuits, a series-parallel control-flow graph can be constructed. The number of registers in the register allocation problem is the maximum of 3 and the parameter $k$ from the WCS instance of size $n$. The construction can be done in time $O(nk)$.

Proof. We assume $k \geq 3$ and set $r = k$.

We represent each gate output from the circuit by a variable (and thus by a node in the conflict graph). We do the same for the complement of the output. Placing a variable in a register represents assigning the value "true" to the output. By adding cliques to the conflict graph and choosing suitable weights on the nodes of the conflict graph we can ensure that exactly one of these two variables will be placed in a register. The gates in the circuit are represented in a similar way. Lemmata 1 and 2 show that this can be done, and their proofs give the details.

We then combine the individual gates as in the circuit and add one node to the control flow graph, at which all input variables of the circuit are alive. See Figure 5 for an example. The control-flow graph now is basically the graph from the circuit with input nodes replaced by a single node $n$, a few nodes of degree one added and some paths subdivided. The weight of the one variable representing the output from the circuit is increased slightly. The weight of the inputs is slightly increased as well, but much less than that of the output.

There are basically two ways to do the combining: A general one that works for any circuit and gives us a $t$-structured program for a circuit of tree-width $t-1$, and a specialized one that works for SAT-circuits and only uses switch/break
(or, alternatively if/else) as control structure. Both essentially go bottom-up through the circuit while constructing the program.

In the general one, for each program fragment generated for a gate, we replace the “break”-statements in the cases 0 to 3 by “return”-statements. We replace the “break”-statements in case 3 by “goto”-statements: If the input \( A \) of the gate is connected to the output of another gate, the “goto” targets the corresponding label. If the input of the gate is an input of the circuit, the “goto” targets the new node \( n \). We handle case 4 in a similar way depending on input \( B \). The control-flow graph is basically the graph from the circuit, subdivided, and with the addition of one node \( n \), and paths that connect \( n \) to various parts of the graph. Thus the tree-width of the control-flow graph is at most one greater than the tree-width \( t - 1 \) of the circuit, and thus the program is \( t \)-structured. Figure 6 shows code generated for the circuit in Figure 5 this way (excluding some code we would add to increase the weights of the output and inputs).

In the specialized way of combining, we generate the program fragment for the gate that has the circuit’s output as its output. If input \( A \) of our gate is the output of another gate, we replace the “break” statement in case 3 by the program fragment generated for that gate. We do this recursively, and also for case 4 with input \( B \). This results in a program that uses only “switch”-statements (if/else could be used instead) as control structure. This makes the program structured even under early, most restrictive notions of “structured program”. Figure 7 shows code generated for the circuit in Figure 5 this way (excluding some code we would add to increase the weights of the output and inputs).

The construction can be done in linear time for fixed \( k \): For each node in the input graph a fixed amount of nodes in the CFG (or a fixed amount of pseudocode) plus whatever is needed for the fixed amount of cliques, is generated. The size of the cliques is linear in the parameter \( k \). We thus get a total runtime and output size of \( O(kn) \subseteq O(n^2) \).

The one node in the control-flow graph where all the live ranges of inputs meet ensures that at most \( r = k \) of the inputs are assigned the value “true”. Thus an optimal assignment will find a valid configuration of the circuit, with exactly \( k \) of the inputs set to “true”. If it is possible to set the output to “true” under these conditions it will happen, since it has slightly higher weight than the other variables.

The sum of the weights of all the variables that would be placed in registers by such an assignment with the output set to “true” can easily be calculated. We then use this sum as \( g \) for the DRA.

\[ \text{Theorem 3.} \quad \text{The register allocation problem DRA, when parametrized by the number of registers } r, \text{ is } W[\text{SAT}]-\text{hard, even for structured programs.} \]

\[ \text{Proof.} \quad \text{Lemma 3 gives a reduction from WCS to DRA. The functions in the reduction are } k \mapsto k' = \max\{3,k\} \text{ and } k \mapsto k'' = k, \text{ and } (x,k) \mapsto x' \text{ with constructed program } x'. \text{ For SAT-circuits, our construction only uses one very basic control structure (switch/break or, alternatively, if/else) which is available in virtually all programming languages. Thus our hardness result is not} \]

11
void h(void)
{
    Y = f();
    Y = f();
    A = f(), \( \mathcal{N} = f() \), B = f(), \( \mathcal{M} = f() \);
    switch (f() % 5)
    {
    case 0:
        CLIQUE2;
        g(Y, \( \mathcal{N}, \mathcal{M} \));
        return;
    case 1:
        CLIQUE1;
        g(Y, A);
        return;
    case 2:
        CLIQUE1;
        g(\( \mathcal{N}, B \));
        A = f(), \( \mathcal{N} = f() \), B = f(), \( \mathcal{M} = f() \);
        return;
    case 3:
        CLIQUE1;
        goto signal_a;
        break;
    case 4:
        CLIQUE1;
        goto signal_b;
    }

    signal_a:
    CLIQUE;
    C = f(), \( \mathcal{N} = f() \), D = f(), \( \mathcal{M} = f() \);
    switch (f() % 5)
    {
    case 0:
        CLIQUE2;
        g(A, C, D);
        return;
    case 1:
        CLIQUE1;
        g(\( \mathcal{N}, C \));
        return;
    case 2:
        CLIQUE1;
        g(\( \mathcal{N}, D \));
        return;
    case 3:
        CLIQUE1;
        D = f(), \( \mathcal{N} = f() \);
        goto n;
    case 4:
        CLIQUE1;
        C = f(); \( \mathcal{N} = f() \);
        goto n;
    }

    signal_b:
    CLIQUE;
    D = f(), \( \mathcal{N} = f() \), E = f(), \( \mathcal{M} = f() \);
    switch (f() % 5)
    {
    case 0:
        CLIQUE2;
        g(D, \( \mathcal{N}, E \));
        return;
    case 1:
        CLIQUE1;
        g(\( \mathcal{N}, D \));
        return;
    case 2:
        CLIQUE1;
        g(\( \mathcal{N}, E \));
        return;
    case 3:
        CLIQUE1;
        C = f(); \( \mathcal{N} = f() \);
        goto n;
    case 4:
        CLIQUE1;
        C = f(); \( \mathcal{N} = f() \);
        goto n;
    }

    n:
    g(C, D, E);
    }

Figure 6: 2-structured pseudocode for the example from Figure 5
void h( void )
{
    Y = f();
    T = f();
    CLIQUE;
    A = f(), B = f(), \( \overline{T} = f() \);
switch ( f() \% 5 )
    case 0:
        CLIQUE2;
        g(\( V, \overline{T}, \overline{Y} \));
        break;
    case 1:
        CLIQUE1;
        g(\( \overline{T}, A \));
        break;
    case 2:
        CLIQUE1;
        g(\( \overline{T}, B \));
        A = f(), B = f(), \( \overline{T} = f() \);
        break;
    case 3:
        CLIQUE1;
        C = f(), \( \overline{T} = f() \), D = f(), \( \overline{T} = f() \);
switch ( f() \% 5 )
        case 0:
            CLIQUE2;
            g(\( A, C, D \));
            C = f(), D = f();
            break;
        case 1:
            CLIQUE1;
            g(\( \overline{T}, C \));
            C = f(), D = f();
            break;
        case 2:
            CLIQUE1;
            g(\( \overline{T}, D \));
            C = f(), D = f();
            break;
        case 3:
            CLIQUE1;
            D = f(), E = f();
            break;
        case 4:
            CLIQUE1;
            C = f(), E = f();
        break;
    case 4:
        CLIQUE1;
        D = f(), \( \overline{T} = f() \), E = f(), \( \overline{T} = f() \);
switch ( f() \% 5 )
        case 0:
            CLIQUE2;
            g(\( C, D, E \));
            D = f(), E = f();
            break;
        case 1:
            CLIQUE1;
            g(\( \overline{T}, D \));
            D = f(), E = f();
            break;
        case 2:
            CLIQUE1;
            g(\( \overline{T}, E \));
            D = f(), E = f();
            break;
        case 3:
            CLIQUE1;
            C = f(), E = f();
            break;
        case 4:
            CLIQUE1;
            C = f(), D = f();
        break;
    case 4:
        CLIQUE1;
        C = f();
    break;
    case 4:
        CLIQUE1;
        D = f();
    break;
switch ( f() \% 5 )
        case 0:
            CLIQUE2;
            g(\( B, \overline{T}, \overline{Y} \));
            B = f(), \( \overline{T} = f() \);
        break;
    case 1:
        CLIQUE1;
        g(\( \overline{T}, B \));
        B = f(), \( \overline{T} = f() \);
        break;
    case 2:
        CLIQUE1;
        g(\( \overline{T}, E \));
        \( \overline{T} = f() \), E = f();
        break;
    case 3:
        CLIQUE1;
        C = f();
        break;
    case 4:
        CLIQUE1;
        C = f();
        break;
    break;

    g(\( C, D, E \));
}

Figure 7: Pseudocode for the example from Figure 5 using only switch/break as control construct
restricted to a particular programming language and holds even for the early, most restrictive notion of “structured program”.

q. e. d.

5. Conclusion

We have proven that register allocation for structured programs is \( W[\text{SAT}] \)-hard. This complements a previous result that register allocation for structured programs is in \( XP \), even when taking into account further aspects, such as different register classes, register preferences, register aliasing, coalescing and rematerialization, which were not considered here [5]. Since \( W[\text{SAT}] \subseteq W[\text{P}] \subseteq XP \), open questions remain about which class exactly register allocation for structured programs falls into. The answer could depend on the further aspects of register allocation mentioned above.


