The Complexity of Register Allocation

Philipp Klaus Krause
Goethe-Universität, Frankfurt am Main

Abstract. Register allocation can be seen as coloring the conflict graph of the program’s variables. The problem of optimal register allocation can be seen as that of finding an \( r \)-colorable induced subgraph of maximum weight in the conflict graph. The parameter \( r \) is the number of registers. Large classes of programs are structured, i.e. the tree-width of their control-flow is bounded \[8][6][2]. Optimal register allocation for structured programs is known to be in XP \[7]. The decision problem of deciding if a conflict graph of a structured program is \( r \)-colorable is known to be fixed-parameter tractable \[1]. Here, a proof that optimal register allocation for structured programs is \( W[\text{SAT}] \)-hard is presented.

1 Introduction

In register allocation one 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 they would overwrite each other’s values.

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 thus the most important stage in a compiler with respect to optimization for typical goals, such as code size, code speed or energy efficiency.

The following section 2 introduced the basic concepts necessary for the discussion of the related work in section 3 and our results in section 4.

2 Preliminaries

Definition 1 (CFG). The control-flow graph of a computer program is a directed graph. The nodes of the CFG are the program’s instructions; there is an edge from \( i \) to \( j \), iff instruction \( j \) can be executed directly after instruction \( i \).

Definition 2 (Program). A program consists of a control-flow graph \( G \), a set of variables \( V \) and a weight function \( c: V \rightarrow [0, \infty] \). Each node of \( G \) is marked by a subset of \( V \) indicating the variables alive there. The nodes of \( G \) that are marked by a variable \( v \in V \) induce a connected subgraph of \( G \), the live-range of \( v \).
This representation can be easily generated from other representations, such as pseudo-code. 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.

```c
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);
}
```

![Diagram](image)

(a) Code and CFG
(b) Variable live-ranges

**Fig. 1.** Some program

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

**Definition 3 (Conflict Graph).** Let $V$ be the set of variables of a program. The conflict graph $(V, E)$ is the intersection graph of the live ranges.

We use $G$ for the control-flow graph, $V$ for the variables and $(V, E)$ for the conflict graph throughout.

**Definition 4 (Structured Program).** Let $t \in \mathbb{N}$ be fixed. A program is called $t$-structured, iff its control-flow graph $G$ has tree-width at most $t$.

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

**Definition 5 (ORA).** Given an input program and parameter $r$, the number of registers, the problem of register allocation 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 5 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 following definition makes register allocation a decision problem in place of the optimization problem from above.

**Definition 6 (DRA).** Given an input program and parameter $r$, the number of registers and a number $g$, the problem of register allocation 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 [4]:

**Definition 7 (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, 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 8 (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 9 (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 $n$ being the size of the input, $f$ being some computable function and $p$ being some polynomial.
- For a positive integer $t W[t]$ is the class of all parametrized problems that can be fpt-reduced to the WCS problem on weft $t$, depth $d$ SAT-circuits, where $d \geq 1$ is a constant.
\(W[\text{SAT}]\) is the class of all parametrized problems that are fixed-parameter reducible to the WCS problem on SAT-circuits.

\(W[P]\) is the class of all parametrized problems that are fixed-parameter reducible to the WCS problem.

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

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

### 3 Previous results

Any graph can occur as the conflict graph of some program [3]. Together with the NP-hardness of graph-coloring [7] this proves the NP-hardness of register allocation when the input is not restricted to structured programs.

However large classes of programs are structured. Programs written in Algol or Pascal are 2-structured, Modula-2 programs are 5-structured, programs written in C are \((6 + g)\)-structured if the number of gotos per function does not exceed \(g\) [8]. Similarly, Java programs are \((6 + g)\)-structured if the number of labeled breaks and labeled continues per function does not exceed \(g\) [6]. Ada programs are \((6 + g)\)-structured if the number of gotos and labeled loops per function does not exceed \(g\) [2].

Gary et alii [5] 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 2-structured programs.

Based on earlier work by Thorup [8], Bodlaender et alii [1] obtained the following result:

**Theorem 1.** For 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 [7] has proven that the register allocation problem ORA for structured programs 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

**Theorem 2.** The register allocation problem, when parametrized by the number of registers \(r\) is \(W[\text{SAT}]\)-hard, even for \(\text{tw}(G) \leq 2\).

**Proof.** The theorem directly follows from the following lemma, since the underlying graph of SAT-circuits (without the input nodes) is a tree, and thus has tree-width at most 1.
Lemma 1. 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. 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 its complement. 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:
For an or-gate we add 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 $Y$ has $r - 2$ other nodes. These other nodes, each of which belongs to exactly one 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 (and thus at most one of them is assigned the value “true”). The same holds 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 $\overline{A}$ is “true”.
- At most one of $B$ and $\overline{B}$ is “true”.
- At most one of $Y$ and $\overline{Y}$ is “true”.
- At most one of $A$ and $\overline{Y}$ is “true”.
- At most one of $B$ and $\overline{Y}$ is “true”.
- At most two of $A$, $\overline{A}$, $B$, $\overline{B}$, $Y$, $\overline{Y}$ are “true”.
- At least three of $A$, $\overline{A}$, $B$, $\overline{B}$, $Y$, $\overline{Y}$ are “true”.

As one can easily verify, this corresponds to an or-gate.

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 5. The cases of the switch statement correspond to the indices in Figure 2(c).

And-gates can be represented in a similar way (see Figure 3).

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 4 for an example. The control-flow graph now is basically the graph from the circuit with input nodes replaced by a single node, a few nodes of degree one added and some paths subdivided, and thus has at most the tree-width of the circuit with the input nodes removed plus one. 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.

This construction can be done in linear time: 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.

Thus finding an optimal assignment of variables to registers results in a “yes” or “no” answer for the WCS problem.

In general the process outlined in the proof above can lead to unnatural-seeming code containing a large number of goto statements. For SAT-circuits, such as the one from Figures 4 and 6 we can do without goto: switch statements (or, alternatively if/else) are sufficient.

References

A Pseudocode

CLIQUE is a macro that expands to generate additional variables as necessary to form the clique and accesses them as often as necessary to give them the desired weight.

Additional code necessary to adjust the weights only is not shown.

```plaintext
signal_y:
CLIQUE;
A = f(), \overline{A} = f(), B = f(), \overline{B} = f();
switch (f() \% 5)
{
  case 0:
    CLIQUE;
    g(Y, \overline{A}, \overline{B});
    return;
  case 1:
    CLIQUE;
    g(Y, A);
    return;
  case 2:
    CLIQUE;
    g(Y, B);
    return;
  case 3:
    CLIQUE;
    goto signal_a;
  case 4:
    CLIQUE;
    goto signal_b;
}
```

Fig. 5. Pseudocode for or-gate from Figure 2
void h(void)
{
    Y = f();
    x = f();
    CLIQUE;
    A = f(); x = f(); B = f();
    switch (f() % 5)
    {
        case 0:
            CLIQUE;
            g(Y, x, B);
            return;
        case 1:
            CLIQUE;
            g(x, A);
            return;
        case 2:
            CLIQUE;
            g(x, B);
            return;
        case 3:
            CLIQUE;
            C = f(); x = f(); D = f();
            switch (f() % 5)
            {
                case 0:
                    CLIQUE;
                    g(A, C, D);
                    return;
                case 1:
                    CLIQUE;
                    g(x, x);
                    return;
                case 2:
                    CLIQUE;
                    g(x, x);
                    return;
                case 3:
                    CLIQUE;
                    D = f(); E = f();
                    break;
                case 4:
                    CLIQUE;
                    C = f(); E = f();
            } break;
        case 4:
            CLIQUE;
            D = f(); x = f(); E = f();
            switch (f() % 5)
            {
                case 0:
                    CLIQUE;
                    g(x, x, E);
                    return;
                case 1:
                    CLIQUE;
                    g(x, D);
                    return;
                case 2:
                    CLIQUE;
                    g(x, E);
                    return;
                case 3:
                    CLIQUE;
                    C = f(); E = f();
                    break;
                case 4:
                    CLIQUE;
                    C = f(); D = f();
            } break;
    }
    g(C, D, E);
}

Fig. 6. Pseudocode for the example from Figure 4