# Faster Register Allocation via Grammatical Decompositions of Control-Flow Graphs

Xuran Cai, Amir Kafshdar Goharshady, S. Hitarth, Chun Kit Lam

# Faster Register Allocation via Grammatical Decompositions of Control-Flow Graphs

Xuran Cai
xcaiay@connect.ust.hk
Hong Kong University of Science and Technology
Hong Kong

Amir Kafshdar Goharshady
goharshady@cse.ust.hk
Hong Kong University of Science and Technology
Hong Kong

S. Hitarth
hitarth.singh@connect.ust.hk
Hong Kong University of Science and Technology
Hong Kong

Chun Kit Lam
cklamaq@connect.ust.hk
Hong Kong University of Science and Technology
Hong Kong

## Abstract

It is well-known that control-flow graphs (CFGs) of structured programs are sparse. This sparsity has been previously formalized in terms of graph parameters such as treewidth and pathwidth and used to design faster parameterized algorithms for numerous compiler optimization, model checking and program analysis tasks.

In this work, we observe that the known graph sparsity parameters fail to exactly capture the kind of sparsity exhibited by CFGs. For example, while all structured CFGs have a treewidth of at most 7, not every graph with a treewidth of 7 or less is realizable as a CFG. As a result, current parameterized algorithms are solving the underlying graph problems over a more general family of graphs than the CFGs.

To address this problem, we design a new but natural concept of graph decomposition based on a grammar that precisely captures the set of graphs that can be realized as CFGs of programs. We show that our notion of decomposition enables the same type of dynamic programming algorithms that are often used in treewidth/pathwidth-based methods. As two concrete applications, using our grammatical decomposition of CFGs, we provide asymptotically more efficient algorithms for two variants of the classical problem of register allocation. Our algorithms are asymptotically faster not only in comparison with the non-parameterized solutions for these problems, but also compared to the state-of-the-art treewidth/pathwidth-based approaches in the literature. For minimum-cost register allocation over a fixed number of registers, we provide an algorithm with a runtime of $O(|G| \cdot |\mathbb{V}|^{5 \cdot r})$ where $|G|$ is the size of the program, $\mathbb{V}$ is the set of program variables and $r$ is the number of registers. In contrast, the previous treewidth-based algorithm had a runtime of $O(|G| \cdot |\mathbb{V}|^{16 \cdot r})$. For the decision problem of spill-free register allocation, our algorithm's runtime is $O(|G| \cdot r^{5 \cdot r+5})$ whereas the previous works had a runtime of $O(|G| \cdot r^{16 \cdot r})$.

Finally, we provide extensive experimental results on spill-free register allocation, showcasing the scalability of our approach in comparison to previous state-of-the-art methods. Most notably, our approach can handle real-world instances with up to 20 registers, whereas previous works could only scale to 8. This is a significant improvement since most ubiquitous architectures, such as the x86 family, have 16 registers. For such architectures, our approach is the first-ever *exact* algorithm that scales up to solve the real-world instances of spill-free register allocation.

## 1 Introduction and Related Works

Many classical tasks in program analysis, compiler optimization and formal verification are traditionally solved by means of a reduction to graph problems. Examples include such well-studied and ubiquitous problems as register allocation [8, 11], cache optimization approaches of data packing [42] and cache-conscious data placement [3, 10], interprocedural data-flow analysis [38], algebraic program analysis [30] and $\mu$-calculus model checking [34], as well as many others.

The resulting graph problems are often NP-hard [11, 33, 42] or even hard-to-approximate unless P=NP [35, 36]. Thus, it is natural to wonder whether the most general case of the graph problem, i.e. considering every graph as a possible input instance, is indeed an appropriate model for the original program analysis, compiler optimization or formal verification task. In many of these tasks, the underlying graph is either the control-flow graph (CFG) of a well-structured program or closely related to it, e.g. obtained by taking several copies of each vertex in the CFG. It is intuitively clear to see that control-flow graphs of programs are quite sparse, thus not every graph can be realized as the control-flow graph of a real program. Hence, any approach that tries to solve the general case of the graph problems is making the problem unnecessarily hard.

There has been ample research on formalizing the sparsity of CFGs. A particularly important result in this direction is that of [43], which proved that CFGs of structured (goto-free) programs in a wide variety of commonly-used languages have a treewidth of at most 6. Treewidth [39] is a measure of treelikeness of graphs. Intuitively, graphs with

smaller treewidth resemble trees more strongly. A graph with treewidth $k$ can be decomposed into small parts (called "bags") of at most $k + 1$ vertices each, such that the bags are connected to each other in a tree-like manner. See Figure 1 for an example and [5, 20] for a more formal treatment. This in turn enables tree-based bottom-up and top-down dynamic programming algorithms and divide-and-conquer approaches [4, 40].

The result of [43] showed that the treewidth of CFGs is at most 6 in languages such as C and Pascal and was followed by a number of similar results extending it to other languages, such as Java [27], Ada [9] and Solidity [12], both theoretically and experimentally. The work [32] showed that the treewidth can be 7 in some variants of C.

These bounded-treewidth results effectively opened up a completely new direction of research. When a program-related task, such as those in compiler optimization or verification, is reduced to a graph problem, we often need not solve it over general graphs, but only graphs of bounded treewidth, since they already cover the set of all possible CFGs. Thus, we are no longer looking for polynomial-time algorithms with a runtime of $O(n^c)$ where $n$ is the size of the input, but instead aim to find so-called *piecewise-polynomial (XP)* or *fixed-parameter tractable (FPT)* algorithms [22] with runtimes of the form $O(n^{f(k)})$ and $O(n^c \cdot f(k))$, respectively, where the parameter $k$ is the treewidth and $f$ is an arbitrary computable function, possibly not a polynomial. Intuitively, since we know that the treewidth is small and at most 7 in all the instances that we care for, i.e. CFGs, an algorithm with a runtime of $O(n^c \cdot f(k))$ is effectively polynomial-time for all real-world intents and purposes. For example, an algorithm that takes $O(n \cdot 2^k)$ will behave just like a linear-time algorithm over CFGs since $k \leq 7$ and thus $2^k$ is a constant.

This approach has been extremely successful in finding polynomial-time (FPT) algorithms for NP-hard graph problems arising in program-related tasks, making their real-world instances tractable. Some prominent examples are as follows:

- After proving the treewidth bound, [43] showed that register allocation can be approximated over graphs of bounded treewidth $k$ within a factor of $\lfloor k/2 + 1 \rfloor$ from optimal in linear time. This leads to a 4-approximation for CFGs. In contrast, it is well-known that the problem is equivalent to graph coloring [11] and thus hard-to-approximate within any constant factor over general graphs unless P=NP.
- This was later improved to an optimal linear-time FPT algorithm for the decision variant of register allocation, i.e. answering whether it is possible to have no spilling with a fixed number $r$ of registers [6].
- It was later shown that the problem of optimal-cost register allocation, i.e. minimizing the total cost of spills given a fixed number of registers, is also solvable in polynomial time (XP) when the treewidth is bounded [31].
- In [7, 19], it was established that any formula in the monadic second-order logic of graphs can be model-checked in linear-time over graphs of bounded treewidth.
- [1, 15] showed that both data packing and cache-conscious data-placement, which are two classical cache management problems in compiler optimization, are solvable/approximable in polynomial time over instances with bounded treewidth, despite their hardness over general instances [33, 35].
- The work [34] showed that the winner in a parity game whose underlying graph has bounded treewidth can be decided in polynomial time. The well-known correspondence between parity games and $\mu$-calculus/LTL model checking makes this applicable to real-world instances obtained from CFGs.

The bounded treewidth result has been useful even when the original graph problem already admitted a polynomial-time solution on all graphs. In many problems, one can obtain a lower-degree polynomial algorithm by exploiting the treewidth. For example:

- The works [13, 14] consider interprocedural data-flow analyses, i.e. problems such as null-pointer identification and available expressions, and provide linear-time FPT algorithms for their on-demand variant. This is in contrast to the non-parameterized methods' quadratic runtime [28, 38]. A similar result is obtained for algebraic program analysis in [17].
- Various problems in linear algebra [25], including Gaussian elimination and many classical qualitative and quantitative tasks on Markov chains and Markov decision processes [2, 16] can be solved in linear-time FPT if the underlying graph has bounded treewidth. This is in contrast to the best-known algorithms for general graphs which take $O(n^\omega)$ where $\omega$ is the matrix multiplication constant. A similar improvement was recently obtained for linear programming [21].

All the advances above in program-related graph problems are based on the same fundamental intuition: CFGs are sparse and solving problems over CFGs is very different and often much easier than solving the same problems over general graphs. Thus, taking this intuition to its natural conclusion, one should wonder if bounded treewidth sufficiently captures the sparsity of control-flow graphs.

On the one hand, it is easy to come up with graphs of bounded treewidth that are not realizable as a CFG. For example, consider a graph with $n$ connected components, each of which is a complete graph on 6 vertices, i.e. $K_6$. Such a graph will have a treewidth of 5 but is clearly not the CFG of any structured program, given that any node in a CFG can have an out-degree of at most two. On the other hand, the recent work [18] shows that CFGs not only have bounded

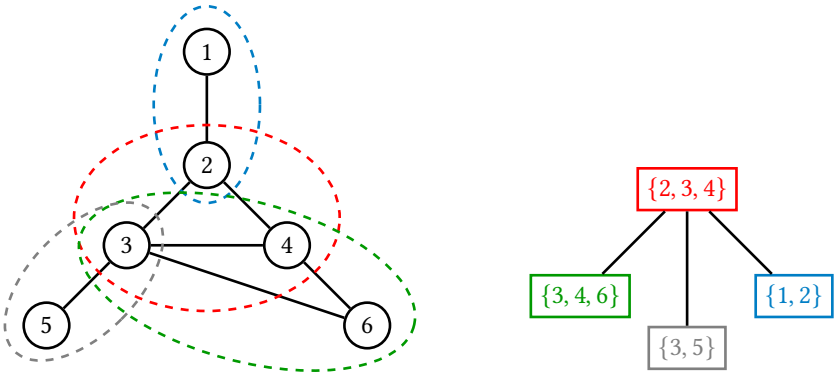



**Figure 1.** A graph $G$ (left) and a tree decomposition of $G$ (right). This example is taken from [18].

treewidth but also often have small pathwidth, i.e. they can be decomposed into small bags that are connected to each other not just in a tree-like manner but in a path-like manner. It then shows that this enables much more efficient dynamic programming algorithms and significant runtime gains for problems such as register allocation. Intuitively, this is because dynamic programming on paths is often simpler than on trees. However, the fundamental question remains: Is bounded pathwidth exactly capturing the sparsity of control-flow graphs? The answer is negative with the same counterexample as above. Thus, we consider this problem: Can we come up with a standard way of decomposing graphs that captures exactly the set of CFGs? Is there a decomposition method that (i) enables fast dynamic programming for problems in compiler optimization and formal verification, and (ii) is applicable to precisely the same set of graphs as CFGs?

In this work, we present such a decomposition based on a graph grammar that closely mimics the grammars used for defining the syntax of structured programming languages. Our approach is similar both to the standard definitions of series-parallel graphs [41] and previous program analysis methods based on graph grammars [29, 45]. However, our graphs are more general in the sense that they cover precisely the set of all control-flow graphs, including CFGs of structured programs that contain `break` and `continue` statements. We define a natural decomposition of CFGs based on our graph grammar and show that it can be used for efficient dynamic programming over the CFGs. As two concrete use-cases, we consider two variants of register allocation which are both classical and well-studied in the literature with both parameterized and non-parameterized solutions. We provide algorithms based on our decomposition that are asymptotically faster than not only the previous non-parameterized solutions, but even the state-of-the-art approaches that exploited bounded treewidth and pathwidth.

We provide extensive experimental results in Section 4, showing that our approach leads to significant runtime gains in practice, too. Most notably, for the problem of spill-free register allocation, our approach is the first algorithm that can handle real-world instances with up to 20 registers. This is a considerable improvement over previous methods which could only handle 8 registers. Crucially, standard and ubiquitous architectures, such as the x86 family, have 16 registers. Thus, we provide the first exact algorithm applicable to them. Given the efficiency of our algorithm, there is no longer a need for approximate or heuristic methods for spill-free register allocation. Since the NP-hardness of register allocation was established by Chaitin in 1982 [11], the community has been mainly focused on developing heuristics with no guarantees of optimality. This work finally breaks the curse and shows that real-world instances of the problem are efficiently solvable by an exact algorithm.

While runtime improvements for register allocation are significant in their own right, we believe that register allocation is but one example and similar improvements can be obtained for a much wider family of program-related analyses by relying on our notion of decomposition instead of treewidth/pathwidth.

In summary, this work takes the idea of exploiting the sparsity of control-flow graphs for faster graph algorithms to its ultimate end and defines a decomposition notion that, unlike parameters such as treewidth and pathwidth, captures *precisely* the set of graphs that can arise as CFGs of structured programs. It then provides faster algorithms using this new notion of decomposition for two classical problems in compiler optimization, i.e. the minimum-cost register allocation problem and the spill-free register allocation problem. We expect that this kind of decomposition would also be useful for the many other problems in which parameterizations by treewidth/pathwidth were applied in the past.

## 2 Our Grammatical Decomposition

To define structured programs, we follow a syntax similar to that of [43]. A program is generated from the following

grammar:

$$P := \epsilon \mid \texttt{break} \mid \texttt{continue} \mid P; P \mid \texttt{if } \varphi \texttt{ then } P \texttt{ else } P \texttt{ fi} \mid \texttt{while } \varphi \texttt{ do } P \texttt{ od}. \quad (1)$$

Here, $\epsilon$ is a neutral statement that does not affect control-flow, e.g. a variable assignment, and $\varphi$ is a boolean expression. We say a program $P$ is *closed* if every `break` and `continue` statement appears within the body of a `while` loop. The semantics of a program $P$ will be defined in the usual manner. In this section, we are only concerned with the control-flow graph of a program $P$. We also note that other common constructs such as `for` loops and `switch` statements can be defined as syntactic sugar [43].

Inspired by series-parallel graphs [44] and the approach of [29], we define a graph grammar that builds up new graphs by combining smaller graphs generated in the same grammar. To capture all control-flow graphs, we consider three composition operations: series, parallel and loop. Thus, we call the graphs generated by this grammar SPL graphs. Also, unlike series-parallel graphs which have two distinguished terminals, i.e. start and end, our graphs have four distinguished vertices, namely start ($S$), terminate ($T$), break ($B$) and continue ($C$). As base cases, the three graphs of Figure 2 are SPL and called *atomic* graphs $A_\epsilon$, $A_{\texttt{break}}$ and $A_{\texttt{continue}}$.

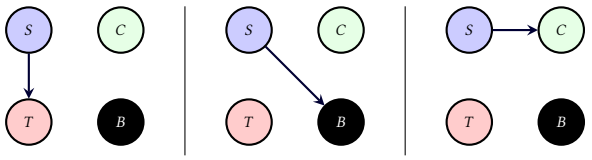



**Figure 2.** Atomic SPL graphs: $A_\epsilon$ (left), $A_{\texttt{break}}$ (middle), and $A_{\texttt{continue}}$ (right).

SPL graphs are generated by the repeated application of series, parallel and loop operations that are denoted by $\otimes$, $\oplus$, and $\circledast$ respectively. Formally, the set of SPL graphs is defined by the following context-free grammar:

$$G = A_\epsilon \mid A_{\texttt{break}} \mid A_{\texttt{continue}} \mid G \otimes G \mid G \oplus G \mid G^{\circledast}. \quad (2)$$

We denote an SPL graph by the tuple $G = (V, E, S, T, B, C)$, where $V$ is the finite set of vertices of the graph, $E$ is the finite set of its edges, and $S, T, B, C \in V$ are the distinguished start, terminate, break and continue vertices. We sometimes drop $V$ and $E$ when they are clear from the context.

We now define our three operations. Let $G_1 = (V_1, E_1, S_1, T_1, B_1, C_1)$ and $G_2 = (V_2, E_2, S_2, T_2, B_2, C_2)$ be two disjoint SPL graphs.

1. *Series Operation.* $G_1 \otimes G_2$ is generated by taking the union of $G_1$ and $G_2$ and merging the pairs of vertices $M = (T_1, S_2)$, $B = (B_1, B_2)$, and $C = (C_1, C_2)$. The distinguished vertices of $G_1 \otimes G_2$ are $(S_1, T_2, B, C)$. It is easy to verify that the series operation is associative. Figure 3 shows two examples of the series operation.
2. *Parallel Operation.* $G_1 \oplus G_2$ is generated by taking union of $G_1$ and $G_2$ and merging the pairs of vertices $S = (S_1, S_2)$, $T = (T_1, T_2)$, $B = (B_1, B_2)$, and $C = (C_1, C_2)$. The special vertex tuple of $G_1 \otimes G_2$ is $(S, T, B, C)$. Figure 4 shows an example of this operation.
3. *Loop Operation.* $G_1^{\circledast}$ is generated by adding four new vertices $S, T, B, C$ to $G_1$ and then adding the following edges: $(S, S_1)$, $(S, T)$, $(T_1, S)$, $(C_1, S)$, and $(B_1, T)$. The special vertex tuple of $G_1^{\circledast}$ is $(S, T, B, C)$. Figure 5 shows an example of the loop operation.

We say that an SPL graph $G = (V, E, S, T, B, C)$ is closed if there are no incoming edges to either $B$ or $C$. There is a natural surjective homomorphism $\text{cfg}(\cdot)$ between programs and SPL graphs, defined as follows:

$$\text{cfg}(\epsilon) = A_\epsilon \quad \text{cfg}(\texttt{break}) = A_{\texttt{break}} \quad \text{cfg}(\texttt{continue}) = A_{\texttt{continue}}$$
$$\text{cfg}(P_1; P_2) = \text{cfg}(P_1) \otimes \text{cfg}(P_2)$$
$$\text{cfg}(\texttt{if } \varphi \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ fi}) = \text{cfg}(P_1) \oplus \text{cfg}(P_2)$$
$$\text{cfg}(\texttt{while } \varphi \texttt{ do } P_1 \texttt{ od}) = \text{cfg}(P_1)^{\circledast}$$

It is easy to verify that this homomorphism matches the usual definition of control-flow graphs of programs and that $P$ is closed if and only if $\text{cfg}(P)$ is closed. Thus, the set of SPL graphs is precisely the same as the set of control-flow graphs of structured programs. Moreover, given a program $P$, we can simply parse it according to the grammar in (1) and obtain a parse tree, then apply our homomorphism to the parse tree to find an equivalent parse tree of its control-flow graph based on the grammar in (2). We call the latter a *grammatical decomposition* of the control-flow graph. Intuitively, the grammatical decomposition is simply the parse tree according to grammar (2), which shows us how the control-flow graph of the current program can be obtained by merging the control-flow graphs of its smaller parts using one of our three operations. Since parsing a context-free grammar takes linear time in the size of the program, i.e. $O(n)$, thus we obtain a grammatical decomposition of our control-flow graph in $O(n)$ time, too. Figure 6 shows an example. Note that in this example, we are also labeling the edges using commands/conditions in the program.

## 3 Register Allocation

As an application of our concept of grammatical decomposition, we consider the problem of minimum-cost register allocation as formalized in [31]. A cost is assigned to each allocation of variables to registers, which is supposed to model the time wasted on spills or rematerialization. We note that this is a more general formulation of the problem than those of [18, 43] which only focus on deciding whether it is possible to avoid spilling altogether and obtain a cost of zero. See Section 3.3. In [31], a treewidth-based algorithm is provided which obtains an optimal register allocation in XP time $O(|G| \cdot |\mathbb{V}|^{2\cdot(t+1)\cdot r})$, where $G$ is the control-flow graph,

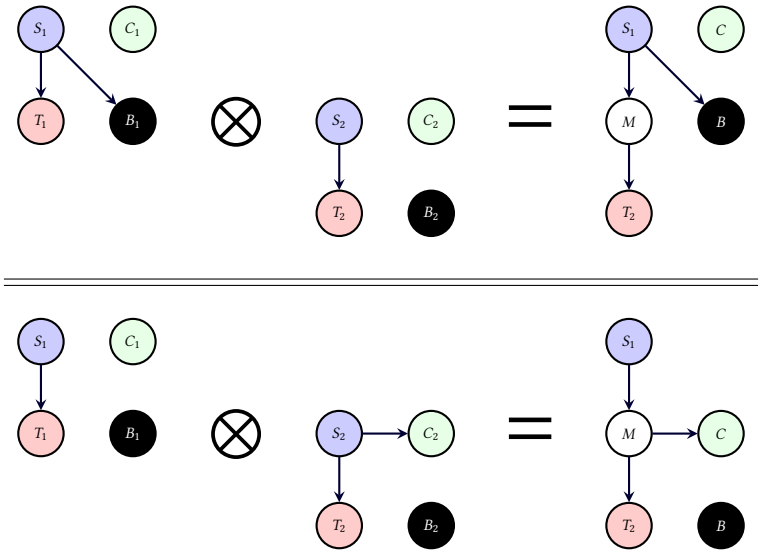



**Figure 3.** Two examples of the series operation $\otimes$.

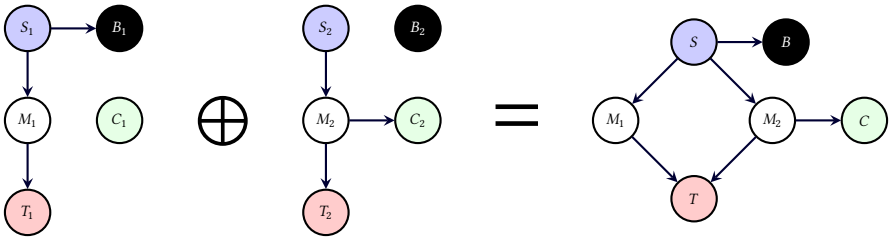



**Figure 4.** An example of the parallel operation $\oplus$.

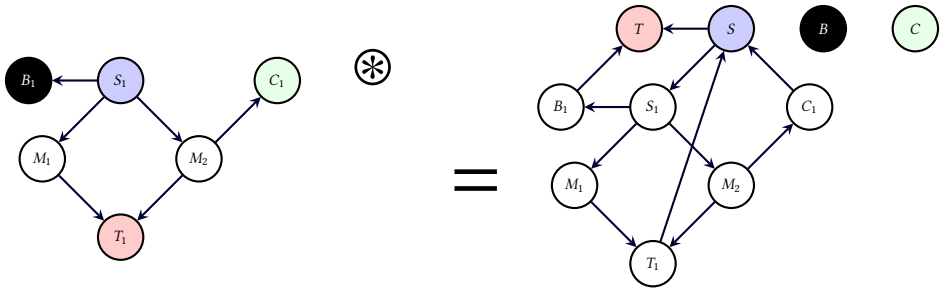



**Figure 5.** An example of the loop operation.

$\mathbb{V}$ is the set of program variables, $t$ is the treewidth of $G$ and $r$ is the number of available registers. If both $r$ and $t$ are constants, then this is a polynomial-time algorithm. However, since the treewidth of programs in languages such as C can be up to 7 [32], this algorithm's worst-case runtime over CFGs is $O(|G| \cdot |\mathbb{V}|^{16 \cdot r})$. In this section, we present an alternative algorithm using our grammatical decomposition which provides a significant runtime improvement and runs in time $O(|G| \cdot |\mathbb{V}|^{5 \cdot r})$. This is a huge asymptotic improvement over the algorithm of [32].

### 3.1 Problem Definition

Suppose we are given a program $P$ with control-flow graph $G = \mathrm{cfg}(P) = (V, E, S, T, B, C)$. Let $[r] = \{0, 1, \ldots, r-1\}$ be the set of available registers and $\mathbb{V}$ the set of our program variables. Every variable $v \in \mathbb{V}$ has a lifetime $\mathsf{lt}(v)$ which is a connected subgraph of $G$. See [37] for a more detailed treatment of lifetimes. Since lifetimes can be computed by a simple data-flow analysis, we assume without loss of generality that they are given as inputs to our algorithm. For a

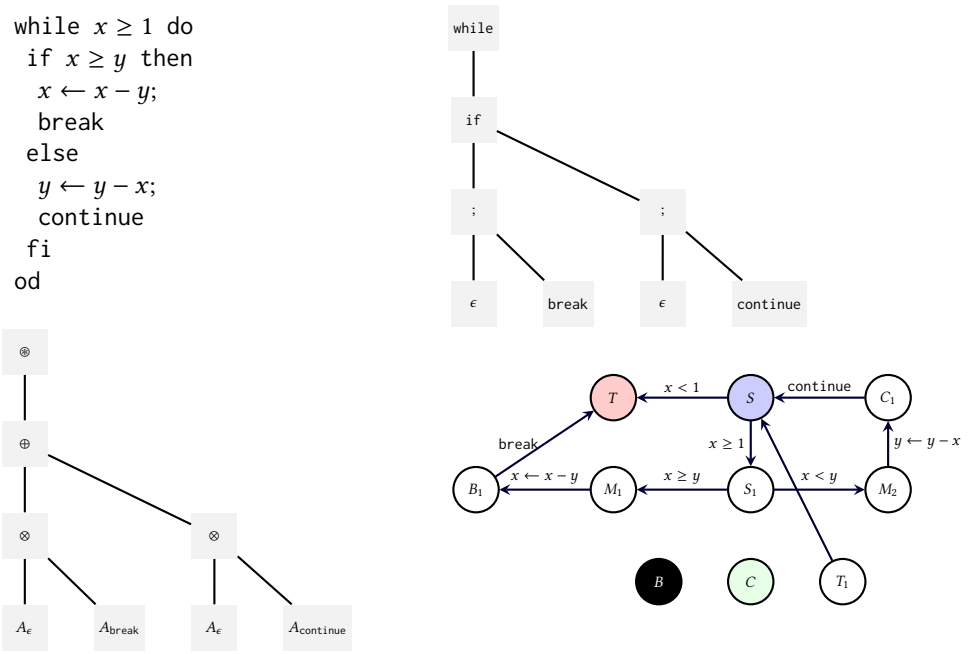



**Figure 6.** A program $P$ (top left), its parse tree (top right), the corresponding parse tree of $G = \text{cfg}(P)$, aka the grammatical decomposition of $G$ (bottom left) and the graph $G = \text{cfg}(P)$ (bottom right). The edges of the graph are labeled according to the commands/conditions of the program.

vertex $v$ or edge $e$ of $G$, we denote the set of variables that are alive at this vertex/edge by $L(v)$ or $L(e)$. An *assignment* is a function $f : \mathbb{V} \to [r] \cup \{\perp\}$ which maps each variable either to a register or to $\perp$ . The latter models the variable being spilled. An assignment is valid if it does not map two variables with intersecting lifetimes to the same register. We denote the set of all valid assignments by $F$.

The interference graph of our program $P$ is a graph $\mathbb{I} = (\mathbb{V}, E_{\mathbb{I}})$ in which there is one vertex for each program variable and there is an edge $\{u, v\}$ if the variables $u$ and $v$ can be alive at the same time, i.e. $\text{lt}(u) \cap \text{lt}(v) \neq \emptyset$. Any valid assignment $f$ is a valid coloring of a subset of vertices of $\mathbb{I}$ with colors in $[r]$. This correspondence between register allocation and graph coloring is well-known and due to Chaitin [11]. We note that for every vertex $v$, the set $L(v)$ of variables alive at $v$ forms a clique in $\mathbb{I}$.

We provide an example taken from [26]. Figure 7 shows a program $P$ and its control-flow graph $G = \text{cfg}(P)$, including live variables at each vertex, and the interference graph $\mathbb{I}$. Our goal is to color a subset of vertices of $\mathbb{I}$ with $r$ colors, where $r$ is the number of available registers. A complete coloring with 4 colors is shown in the figure. This avoids any spilling. We also show a partial coloring with 3 colors and some spilling.

A *cost function* [31] is a function $c : E \times F \to [0, \infty)$. For an edge $e \in E$ of the control-flow graph, which corresponds to one command of the program, $c(e, f)$ is the cost of running this command when the registers are allocated as per $f$. We assume that $c(e, f)$ only depends on the allocation decisions for variables that are alive at $e$. Following [31], we further assume constant-time oracle access to evaluations of $c$. In practice, $c$ is often obtained by profiling. Different optimization goals, such as total runtime or code size, may be modeled by choosing a suitable function $c$.

The problem of optimal register allocation provides $P, G, r, \mathbb{V}, c$ and the lifetimes of variables as input and asks for an assignment $f \in F$ that minimizes the total cost, i.e. our goal is to

$$\text{minimize} \sum_{e \in E} c(e, f)$$

by choosing the best possible $f$.

### 3.2 Our Algorithm

We now show how to perform dynamic programming on the grammatical decomposition of our control-flow graph $G$ to find an optimal register allocation. Our algorithm is quite simple and elegant. We process our grammatical decomposition in a bottom-up fashion and for every subgraph $H = (V_H, E_H, S_H, T_H, B_H, C_H)$ appearing in the grammatical decomposition, define the following dynamic programming variables:

$$\textsc{Opt}[H, f'] = \quad \text{The minimum total cost } \sum_{e \in E_H} c(e, f) \text{ of an assignment } f \text{ over } H \text{ such that } f_{|L(S_H) \cup L(T_H) \cup L(B_H) \cup L(C_H)} = f'.$$

Intuitively, for every possible assignment $f'$ of the variables that are alive at any of the distinguished vertices $(S_H, T_H, B_H, C_H)$,

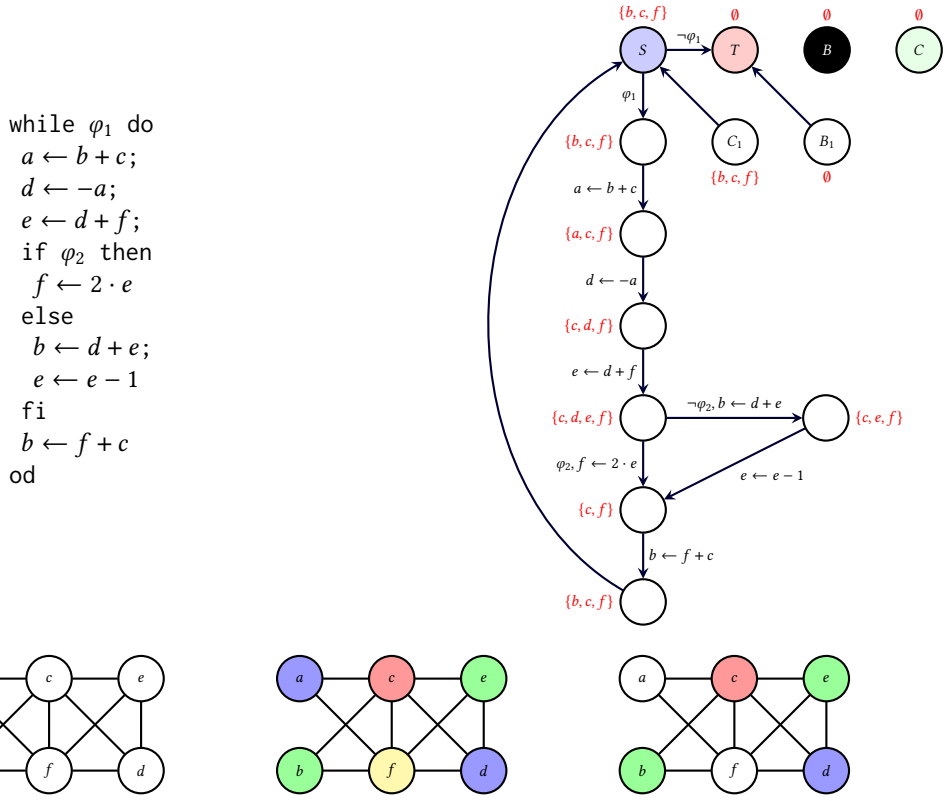



**Figure 7.** An Example Program $P$ (top left), its control-flow graph $G = \text{cfg}(P)$ (top right) in which every vertex is labeled by its set of live variables in red, the interference graph $\mathbb{I}$ (bottom left), a coloring of all vertices of $\mathbb{I}$ with 4 colors corresponding to allocating all variables to 4 registers (bottom center), and a coloring of a subset of vertices of $\mathbb{I}$ with 3 colors corresponding to spilling the variables $a$ and $f$ (bottom right).

we are asking for the minimum total cost of an assignment $f$ over all variables that agrees with $f'$ and extends it. After we compute our $\textsc{Opt}[\cdot,\cdot]$ values, the final answer of the algorithm, i.e. the minimum cost of a register allocation, is simply $\min_f \textsc{Opt}[G, f]$.

We now show how to compositionally compute $\textsc{Opt}[H, f']$ for any SPL graph $H$ assuming that we have already computed $\textsc{Opt}[\cdot,\cdot]$ values for the SPL subgraphs of $H$. This is done by casework:

- *Atomic Graphs:* If $H \in \{A_\epsilon, A_{\texttt{break}}, A_{\texttt{continue}}\}$, then $H$ does not have any vertices other than the distinguished vertices $(S_H, T_H, B_H, C_H)$. Thus, all variables that are alive at any point in $H$ are also alive at one of the distinguished vertices and we simply set $\textsc{Opt}[H, f'] = c(e, f')$ for every partial allocation $f'$. Here, $e$ is the unique edge in $H$.

**Compatible Assignments.** We say two partial assignments $f_1 : \mathbb{V}_1 \to [r] \cup \{\perp\}$ and $f_2 : \mathbb{V}_2 \to [r] \cup \{\perp\}$ are compatible and write $f_1 \leftrightarrows f_2$ if $\forall v \in \mathbb{V}_1 \cap \mathbb{V}_2 \quad f_1(v) = f_2(v)$. Informally, $f_1$ and $f_2$ never make conflicting decisions on any variable $v$ but we have no restrictions on variables that are decided only by $f_1$ or only by $f_2$. In other words, $f_1$ and $f_2$ can be combined in the same total assignment.

- *Series Operation:* If $H = H_1 \otimes H_2$, then we have

$$\textsc{Opt}[H_1 \otimes H_2, f'] =$$

$$\min_{\substack{f' \leftrightarrows f_1' \\ f' \leftrightarrows f_2' \\ f_1' \leftrightarrows f_2'}} \left( \textsc{Opt}[H_1, f_1'] + \textsc{Opt}[H_2, f_2'] - \sum_{e \in E_{H_1} \cap E_{H_2}} c(e, f_1') \right).$$

The correctness of this calculation is an immediate corollary of the definition of our series operation. By construction, we have $L(B_{H_1}) = L(B_{H_2}) = L(B_H)$ and $L(C_{H_1}) = L(C_{H_2}) = L(C_H)$ and also $L(T_{H_1}) = L(S_{H_2})$. Moreover, every edge of $H_1$ and $H_2$ is preserved in $H_1 \otimes H_2$. Thus, the total cost is simply the sum of costs in the two components. We should also be careful not to double-count the cost of edges that appear in both $H_1$ and $H_2$. Thus, we subtract these. Of course, the partial assignments $f', f_1'$ and $f_2'$ should be pairwise compatible.

- *Parallel Operation:* This case is handled exactly as in the series case:

$$\textsc{Opt}[H_1 \oplus H_2, f'] =$$

$$\min_{\substack{f' \leftrightarrows f_1' \\ f' \leftrightarrows f_2' \\ f_1' \leftrightarrows f_2'}} \left( \textsc{Opt}[H_1, f_1'] + \textsc{Opt}[H_2, f_2'] - \sum_{e \in E_{H_1} \cap E_{H_2}} c(e, f_1') \right).$$

  This is because our parallel operation also preserves all the edges in $H_1$ and $H_2$. Note that we might have edges that appear in both $H_1$ and $H_2$, e.g. we might have both $(S_{H_1}, B_{H_1})$ and $(S_{H_2}, B_{H_2})$ which are the same as the edge $(S_H, B_H)$. Thus, the total cost is the sum of costs in the components $H_1$ and $H_2$ minus the cost of their common edges. As before, we should also ensure that the partial assignments are all compatible.
- *Loop Operation:* Suppose $H = H_1^{\circledast}$. In this case, by our construction, $H$ has the same vertices and edges as $H_1$ except for the introduction of the four new distinguished vertices $(S_H, T_H, B_H, C_H)$ and five new edges $e_1 = (S_H, S_{H_1}), e_2 = (S_H, T_H), e_3 = (T_{H_1}, S_H), e_4 = (C_{H_1}, S_H)$ and $e_5 = (B_{H_1}, T_H)$. Thus, our total cost is simply the total cost in $H_1$ plus the cost incurred at these new edges. Therefore, we have:

$$\textsc{Opt}[H_1^{\circledast}, f'] = \min_{f_1' \leftrightarrows f'} \left( \textsc{Opt}[H_1, f_1'] + \sum_{i=1}^{5} c(e_i, f' \cup f_1') \right).$$

This concludes our algorithm which computes the cost of an optimal assignment $f$. As is standard in dynamic programming approaches, $f$ itself can be obtained by retracing the steps of the algorithm and remembering the choices that led to the minimum values at every step.

**Theorem 3.1.** *Given a program $P$ with variables $\mathbb{V}$ and control-flow graph $G = cfg(P)$, the number $r$ of available registers and a cost function $c(\cdot,\cdot)$ as input, our algorithm above finds an optimal allocation of registers, i.e. an optimal assignment function $f$, in time $O(|G| \cdot |\mathbb{V}|^{5 \cdot r})$.*

*Proof.* Correctness was argued above. We do a casework for runtime analysis:

- At atomic graphs, we are considering partial assignments $f'$ over variables that are alive at any of the four distinguished vertices. Let $a$ be one of these distinguished vertices. The set $L(a)$ of alive variables at $a$ forms a clique in the interference graph $\mathbb{I}$. Thus, any valid $f'$ can assign $f'(v) \neq \perp$ to at most $r$ variables $v$ in $L(a)$. Moreover, no two variables can be assigned to the same register. Given that $|L(a)| \leq |\mathbb{V}|$, the total number of possible assignments for variables in $L(a)$ is at most

$$\binom{|\mathbb{V}|}{r} \cdot r! + \binom{|\mathbb{V}|}{r-1} \cdot (r-1)! + \cdots + \binom{|\mathbb{V}|}{0} \cdot 0! \in O(r \cdot |\mathbb{V}|^r).$$

  Thus, the total number of $f'$ functions is at most $O(r^4 \cdot |\mathbb{V}|^{4 \cdot r})$ given that we have four distinguished vertices. Our algorithm spends a constant amount of time for each $f'$, simply querying the cost of a single edge.
- When $H = H_1 \otimes H_2$, we note that we have $B_H = B_{H_1} = B_{H_2}$ and $C_H = C_{H_1} = C_{H_2}$. Similarly, we have $T_{H_1} = S_{H_2}$. Thus, $f', f_1'$ and $f_2'$ need to jointly choose a register assignment for the variables that are alive at one of five vertices: $S_{H_1}, T_{H_1}, T_{H_2}, B$ and $C$. An argument similar to the previous case shows that there are $O(r^5 \cdot |\mathbb{V}|^{5 \cdot r})$ such assignments. We also note that $E_{H_1} \cap E_{H_2}$ has $O(1)$ many edges since any such edge must be connecting two distinguished vertices and we have only four such vertices. Thus, the total runtime here is also $O(r^5 \cdot |\mathbb{V}|^{5 \cdot r})$.
- When $H = H_1 \oplus H_2$, a similar argument applies. In this case, we have $S_H = S_{H_1} = S_{H_2}, T_H = T_{H_1} = T_{H_2}, B_H = B_{H_1} = B_{H_2}$ and $C_H = C_{H_1} = C_{H_2}$. Thus, we need to look at assignments for live variables at only four different vertices and our runtime is $O(r^4 \cdot |\mathbb{V}|^{4 \cdot r})$.
- Finally, when $H = H_1^{\circledast}$, we are introducing four new distinguished vertices. So, it seems that we have to consider the live variables at eight vertices in total, i.e. the distinguished vertices of both $H$ and $H_1$. However, note that $B_{H_1}$ has only one outgoing edge in our control-flow graph $G$ which goes to $T_H$. Thus, we have $L(B_{H_1}) \subseteq L(T_H)$. For similar reasons, $L(T_{H_1}) \subseteq L(S_H)$ and $L(C_{H_1}) \subseteq L(S_H)$. Therefore, we only need to consider the program variables that are alive at one of the five vertices $S_H, T_H, B_H, C_H$ and $S_{H_1}$. An argument similar to the previous cases shows that our runtime is $O(r^5 \cdot |\mathbb{V}|^{5 \cdot r})$.

Finally, our algorithm has to process the grammatical decomposition in a bottom-up manner and compute the $\textsc{Opt}[\cdot,\cdot]$ values at every node. We have $O(|G|)$ nodes. Thus, the total worst-case runtime is $O(|G| \cdot r^5 \cdot |\mathbb{V}|^{5 \cdot r})$. Following [31] and other works on minimum-cost register allocation, we assume that $r$ is a constant. Thus, our runtime is $O(|G| \cdot |\mathbb{V}|^{5 \cdot r})$. □

**Parallelization.** We note that our algorithm is perfectly parallelizable since at every SPL subgraph $H$, one can compute the $\textsc{Opt}[H, f]$ values for different $f$ functions in parallel. Thus, if we have $k$ threads and $k$ is less than the number of possible partial functions $f$, then our runtime is reduced to $O\left(\frac{|G| \cdot |\mathbb{V}|^{5 \cdot r}}{k}\right)$.

### 3.3 Spill-free Register Allocation

We remark that the works [6] (SODA 1998) and [18] (OOPSLA 2023) provide linear-time algorithms with respect to the size of the CFG for the decision problem of existence of a spill-free register allocation, i.e. setting $c(e, f) = 0$ if $f$ is valid and allocates all variables to registers meaning that it does not map anything to $\perp$ and $c(e, f) = +\infty$ otherwise,

and simply asking whether an assignment with zero total cost is attainable. This is a special case of the problem we considered above. Unlike the general case, works on this special case often do not consider $r$ to be a constant and analyze their runtimes based on both $|G|$ and $r$. The former work provides an algorithm with a runtime of $O(|G| \cdot r^{2 \cdot t \cdot r+2 \cdot r})$ where $t$ is the treewidth of the control-flow graph. The latter uses a different parameter, namely pathwidth, and obtains a runtime of $O(|G| \cdot p \cdot r^{p \cdot r+r+1})$. Given that the treewidth of a structured program in languages such as C can be up to 7 [32] and the pathwidth is also empirically observed to be no more than 17 in [18], these approaches provide runtimes of $O(|G| \cdot r^{16 \cdot r})$ and $O(|G| \cdot r^{18 \cdot r+1})$ respectively. Moreover, [18] (Figure 9) observes that the vast majority of real-world programs have a pathwidth of 6 or lower. For these instances, their runtime would be $O(|G| \cdot r^{7 \cdot r+1})$. While this is applicable to a majority of real-world CFGs, it does not cover all of them. In general, there is no known constant bound on the pathwidth of CFGs and thus [18]'s algorithm has a much higher running time in the theoretical worst case.

We now show that our algorithm above significantly improves the time complexity, i.e. the dependence on $r$, for the spill-free register allocation problem, as well.

**Theorem 3.2.** *The algorithm of Section 3.2 can be directly applied to spill-free register allocation, i.e. the case where the cost $c(e, f)$ is zero when $f$ does not map anything to $\perp$ and $+\infty$ otherwise, and solves the problem in time $O(|G| \cdot r^{5 \cdot r+5})$.*

*Proof.* The proof is exactly the same as that of Theorem 3.1, with one additional observation as follows: If we have $|L(a)| > r$ for some vertex $a$ of the control-flow graph, i.e. if there are more than $r$ variables alive at the same time, then the answer to spill-free register allocation is "no". This is because the vertices in $L(a)$ form a clique in $\mathbb{I}$. Thus, we only have to consider the case where $|L(a)| \leq r$ for every $a$. Therefore, in the analysis of the proof of Theorem 3.1, when we consider the variables that are alive at $k \leq 5$ vertices, we can be sure that there are at most $k \cdot r$ such variables. Hence, our total runtime is $O(|G| \cdot r^5 \cdot r^{5 \cdot r}) = O(|G| \cdot r^{5 \cdot r+5})$. □

**Further Optimization.** There is also a simple optimization which can improve the performance of our algorithm in practice. For the spill-free register allocation problem, we do not have any register preferences. Thus, renaming and permuting registers does not invalidate a valid assignment. Similarly, if an assignment is invalid, renaming registers cannot fix the conflicts. Thus, register assignments that are the same modulo register renaming are the same to us. This means we only need to store one representative from each equivalence class as the canonical representation. This significantly reduces the number of dynamic programming values that our algorithm has to compute.

## 4 Experimental Results

In this section, we provide experimental results comparing our algorithm for spill-free register allocation with previous approaches that are based on treewidth and pathwidth. More specifically, given an input program, our goal is to find the smallest number $r$ of registers that would suffice for spill-free allocation. As mentioned above, spill-free register allocation is a special case of register allocation with minimum cost, where the cost is zero if there is no spilling and infinite otherwise. We chose this problem for our experimental evaluation for two reasons: (i) Most of the previous works in the literature focus on this variant, and (ii) In general, there is no standard approach to choosing the cost function $c$ and each compiler defines this function differently according to its own setting and use-cases, often based on dynamic analysis and profiling. However, spill-free allocation is well-defined and the same in all compilers.

**Baselines.** We compare our spill-free register allocation algorithm with the previous state-of-the-art methods that parameterize based on pathwidth [18] and treewidth [6]. To the best of our knowledge, these are the only exact and non-heuristic algorithms for spill-free register allocation in the literature. Other methods, such as [43], are approximate and thus not directly comparable.

**Implementation.** We implemented our approach in `C++` and integrated it with the Small Device C Compiler (SDCC) [23, 24]. SDCC already contains a heavily-optimized variant of the algorithms of [32, 43] to find tree decompositions. Moreover, [18] provides SDCC support for path decompositions and implementations of both their own algorithm and [6] in SDCC. Thus, both previous state-of-the-art parameterized algorithms were already available in SDCC. Despite our approach being perfectly parallelizable, we did not use parallelization in our experiment in order to provide a fair comparison with the available implementations of previous methods, which are not parallel.

**Machine.** Our results were obtained on a Linux machine with an Intel i9-12900HK processor (14 cores, 20 threads, 24 MB cache) and 32 GB of memory.

**Benchmarks.** We followed [18] in our setup. We used the SDCC regression test suite as our set of benchmarks, which consists of 15,888 instances. These benchmarks are all embedded programs which are expected to be executed in systems with limited resources. Thus, spilling constitutes an important performance bottleneck for them and spill-free register allocation with the minimum possible number of registers is highly desirable. We set a time limit of 10 minutes and a memory limit of 4 GB per benchmark.

**Number of Registers.** In [18]'s experiments, the number of registers $r$ is limited to 8. Their approach simply gives up if no spill-free allocation with 8 registers exists. This is the maximum number of registers that can be practically supported

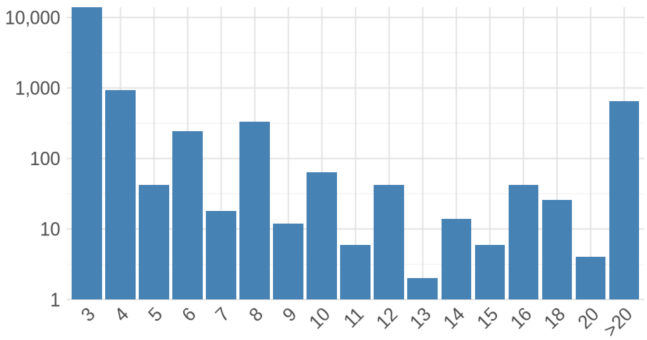



**Figure 8.** Histogram of the minimum number of registers required for spill-free allocation. The $x$ axis is number of registers and the $y$ axis is the number of instances requiring that many registers. The $y$ axis is in logarithmic scale.

by either [18] or [6], due to their exponential runtime and memory dependence on $r$. Given that we have improved this exponential dependence from $r^{16 \cdot r}$ to $r^{5 \cdot r+5}$ (Thereom 3.2), we observed significant practical runtime improvements, too, and our approach can handle up to 20 registers in practice. Thus, we raised the maximum number of registers to 20. This is a huge improvement since [6, 18] were only applicable to domain-specific embedded architectures with up to 8 registers, whereas we can, for the first time, support much more general and ubiquitous architectures such as the x86 family, used in most modern computers, which have 16 registers.

**Failure Statistics.** Our approach successfully handled all input instances within the prescribed time and memory limits, either finding the optimal number of registers needed for spill-free allocation or reporting that more than 20 registers are required. Figure 8 shows a histogram of the number of required registers. In contrast, the pathwidth-based algorithm of [18] and the treewidth-based approach of [6] both failed in 554 instances, including all instances requiring more than 8 registers.

**Runtimes.** Figure 9 shows a comparison of the runtimes of our algorithm vs the treewidth-based approach of [6] and the pathwdith-based approach of [18], when we set $r \leq 20$. The average runtimes were 3.87 microseconds for our approach, 21,544 microseconds for [18] and 1,191,284 microseconds for [6]. These averages are excluding the instances over which the previous methods failed. The runtimes were dominated by 704 instances for the treewidth-based approach and 24 instances for the pathwidth-based approach in which they were unusually slow, presumably due to high treewidth/pathwidth. Excluding these outlier instances, the average runtime was 33.88 microseconds for [18] and 372.34 microseconds for [6].

**Discussion.** In summary, the exponential asymptotic runtime improvements of Theorem 3.2 are also evident in practice and our approach is the first exact algorithm for spill-free register allocation that can scale up to realistic architectures with up to 20 registers, such as the x86 family. Given the efficiency of our approach, obtaining an average runtime of merely 4 microseconds per instance, we believe there is no longer a case for using approximations or heuristics in spill-free register allocation and, despite its NP-hardness and hardness-of-approximation in theory, this problem is efficiently solved by our method for all practical instances.

## 5 Conclusion and Future Work

In this work, we provided a grammar-based decomposition for control-flow graphs of structured programs which (i) precisely captures the set of control-flow graphs and (ii) enables bottom-up dynamic programming in a manner similar to algorithms that exploit treewidth and pathwidth. As an application of our decomposition method, we improved the best known asymptotic runtimes for two classical problems in compiler optimization, namely spill-free and minimum-cost register allocation, by exponential factors with respect to the number of variables and registers. We provided extensive experimental results comparing our approach with previous state-of-the-art parameterized algorithms. Most notably, our approach is the first exact algorithm for spill-free register allocation that can handle ubiquitous architectures such as x86, which have 16 registers. Previous approaches were limited to only 8 registers and could only be applied to domain-specific architectures such as embedded systems with few registers. As future work, we believe that our decomposition has the potential to lead to similar significant runtime improvements for a wide variety of tasks in compiler optimization, program verification and model checking which are currently handled by treewidth/pathwidth-based methods.

## References

[1] Ali Ahmadi, Majid Daliri, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. Efficient approximations for cache-conscious data placement. In *PLDI*, pages 857–871. ACM, 2022.

[2] Ali Asadi, Krishnendu Chatterjee, Amir Kafshdar Goharshady, Kiarash Mohammadi, and Andreas Pavlogiannis. Faster algorithms for quantitative analysis of mcs and mdps with small treewidth. In *ATVA*, volume 12302 of *Lecture Notes in Computer Science*, pages 253–270. Springer, 2020.

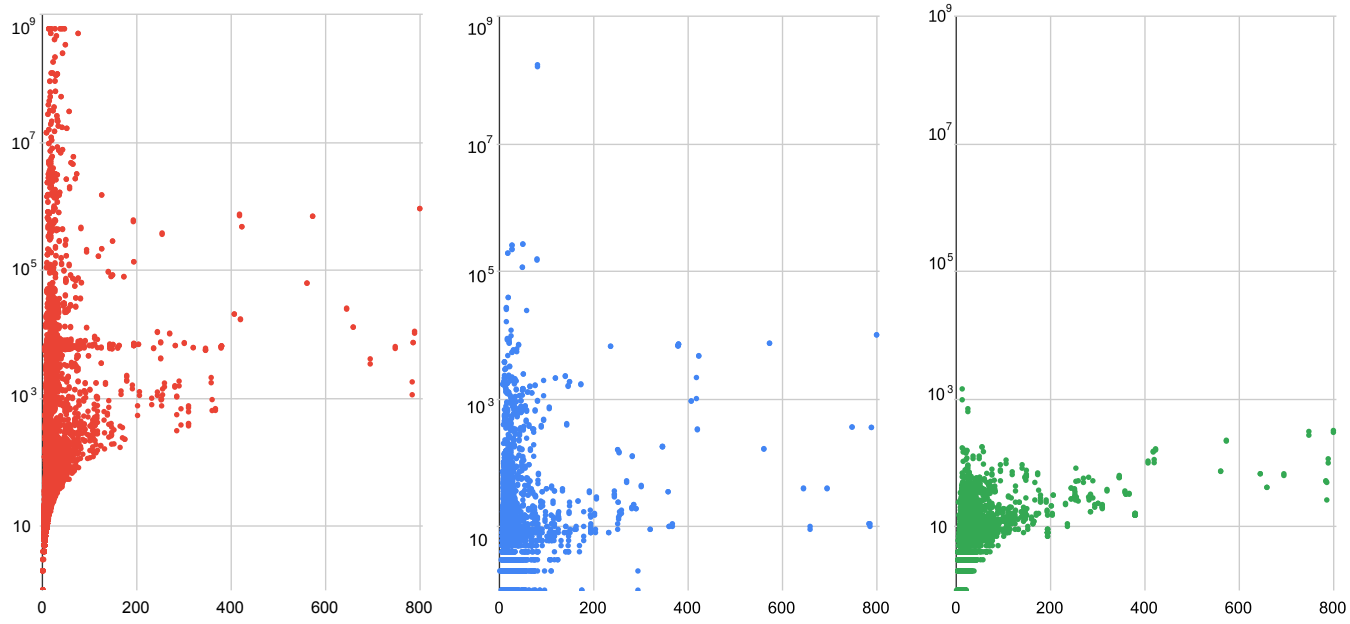
[3] Mirza Omer Beg and Peter van Beek. A graph theoretic approach to cache-conscious placement of data for direct mapped caches. In *ISMM*, pages 113–120. ACM, 2010.

[4] Hans L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In *ICALP*, volume 317 of *Lecture Notes in Computer Science*, pages 105–118. Springer, 1988.

[5] Hans L. Bodlaender. Discovering treewidth. In *SOFSEM*, volume 3381 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.

[6] Hans L. Bodlaender, Jens Gustedt, and Jan Arne Telle. Linear-time register allocation for a fixed number of registers. In *SODA*, pages 574–583. ACM/SIAM, 1998.

[7] Richard B. Borie, R. Gary Parker, and Craig A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7(5&6):555–581, 1992.




**(a)** Treewidth-based [6] **(b)** Pathwidth-based [18] **(c)** Our approach

**Figure 9.** Runtime comparison of the treewidth-based algorithm of [6] (left), the pathwdith-based algorithm of [18] (center) and our approach (right). The $x$ axis is the number of vertices in the CFG and the $y$ axis is the time in microseconds. The $y$ axis is in logarithmic scale. In each panel, the data excludes failure instances. For previous approaches, this effectively limits the results to instances requiring 8 registers or fewer.


[8] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the np-completeness proof of chaitin et al. really prove? or revisiting register allocation: Why and how. In *LCPC*, pages 283–298, 2006.

[9] Bernd Burgstaller, Johann Blieberger, and Bernhard Scholz. On the tree width of ada programs. In *Ada-Europe*, volume 3063 of *Lecture Notes in Computer Science*, pages 78–90. Springer, 2004.

[10] Brad Calder, Chandra Krintz, Simmi John, and Todd M. Austin. Cache-conscious data placement. In *ASPLOS*, pages 139–149. ACM, 1998.

[11] Gregory J. Chaitin. Register allocation and spilling via graph coloring (with retrospective). In *Best of PLDI*, pages 66–74, 1982.

[12] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. The treewidth of smart contracts. In *SAC*, pages 400–408. ACM, 2019.

[13] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. Algorithms for algebraic path properties in concurrent systems of constant treewidth components. In *POPL*, pages 733–747. ACM, 2016.

[14] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. Optimal and perfectly parallel algorithms for on-demand data-flow analysis. In *ESOP*, volume 12075 of *Lecture Notes in Computer Science*, pages 112–140. Springer, 2020.

[15] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Nastaran Okati, and Andreas Pavlogiannis. Efficient parameterized algorithms for data packing. *Proc. ACM Program. Lang.*, 3(POPL):53:1–53:28, 2019.

[16] Krishnendu Chatterjee and Jakub Lacki. Faster algorithms for markov decision processes with low treewidth. In *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 543–558. Springer, 2013.

[17] Giovanna Kobus Conrado, Amir Kafshdar Goharshady, Kerim Kochekov, Yun Chen Tsai, and Ahmed Khaled Zaher. Exploiting the sparseness of control-flow and call graphs for efficient and on-demand algebraic program analysis. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1993–2022, 2023.

[18] Giovanna Kobus Conrado, Amir Kafshdar Goharshady, and Chun Kit Lam. The bounded pathwidth of control-flow graphs. *Proc. ACM Program. Lang.*, 7(OOPSLA2):292–317, 2023.

[19] Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.

[20] Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*, volume 5. Springer, 2015.

[21] Sally Dong, Yin Tat Lee, and Guanghao Ye. A nearly-linear time algorithm for linear programs with small treewidth: a multiscale representation of robust central path. In *STOC*, pages 1784–1797. ACM, 2021.

[22] Rodney G Downey and Michael Ralph Fellows. *Parameterized complexity*. Springer, 2012.

[23] Sandeep Dutta. Anatomy of a compiler: A retargetable ansi-c compiler. *Circuit Cellar*, 121(5), 2000.

[24] Sandeep Dutta, Daniel Drotos, Kevin Vigor, et al. Small device C compiler, 2003.

[25] Fedor V Fomin, Daniel Lokshtanov, Saket Saurabh, Michał Pilipczuk, and Marcin Wrochna. Fully polynomial-time parameterized computations for graphs and matrices of low treewidth. *ACM Transactions on Algorithms (TALG)*, 14(3):1–45, 2018.

[26] H. C. Grossman. Register allocation example, 2013.

[27] Jens Gustedt, Ole A. Mæhle, and Jan Arne Telle. The treewidth of java programs. In *ALENEX*, volume 2409 of *Lecture Notes in Computer Science*, pages 86–97. Springer, 2002.

[28] Susan Horwitz, Thomas W. Reps, and Shmuel Sagiv. Demand interprocedural dataflow analysis. In *SIGSOFT FSE*, pages 104–115. ACM, 1995.

[29] Ken Kennedy and Linda Zucconi. Applications of a graph grammar for program control flow analysis. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 72–85, 1977.

[30] Zachary Kincaid, Thomas W. Reps, and John Cyphert. Algebraic program analysis. In *CAV (1)*, volume 12759 of *Lecture Notes in Computer Science*, pages 46–83. Springer, 2021.

[31] Philipp Klaus Krause. Optimal register allocation in polynomial time. In *CC*, volume 7791 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2013.

[32] Philipp Klaus Krause, Lukas Larisch, and Felix Salfelder. The tree-width of C. *Discret. Appl. Math.*, 278:136–152, 2020.

[33] Rahman Lavaee. The hardness of data packing. In *POPL*, pages 232–242. ACM, 2016.

[34] Jan Obdrzálek. Fast mu-calculus model checking when tree-width is bounded. In *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 80–92. Springer, 2003.

[35] Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. In *POPL*, pages 101–112. ACM, 2002.

[36] Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. *Nord. J. Comput.*, 12(3):275–307, 2005.

[37] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *TOPLAS*, 21(5):895–913, 1999.

[38] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61. ACM, 1995.

[39] Neil Robertson and Paul D. Seymour. Graph minors. III. planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.

[40] Neil Robertson and Paul D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.

[41] Kazuhiko Takamizawa, Takao Nishizeki, and Nobuji Saito. Linear-time computability of combinatorial problems on series-parallel graphs. *Journal of the ACM (JACM)*, 29(3):623–641.

[42] Khalid Omar Thabit. *Cache management by the compiler*. Rice University, 1982.

[43] Mikkel Thorup. All structured programs have small tree-width and good register allocation. *Inf. Comput.*, 142(2):159–181, 1998.

[44] Jacobo Valdes, Robert Endre Tarjan, and Eugene L. Lawler. The recognition of series parallel digraphs. In *STOC*, pages 1–12. ACM, 1979.

[45] Niklaus Wirth. On the composition of well-structured programs. *ACM Computing Surveys (CSUR)*, 6(4):247–259, 1974.