Linear-Time Register Allocation for a Fixed Number of Registers

Hans Bodlaender[*] Jens Gustedt[†] Jan Arne Telle[‡]

Abstract We show that for any *fixed* number of registers there is a linear-time algorithm which given a structured ($\equiv$ `goto`-free) program finds, if possible, an allocation of variables to registers without using intermediate storage. Our algorithm allows for *rescheduling*, *i.e.* that straight-line sequences of statements may be reordered to achieve a better register allocation as long as the data dependencies of the program are not violated.

If we also allow for registers of different types, *e.g.* for integers and floats, we can give only a polynomial time algorithm. In fact we show that the problem then becomes hard for the W-hierarchy which is a strong indication that no $O(n^c)$ algorithm exists for it with c independent on the number of registers. However, if we do not allow for rescheduling then this non-uniform register case is also solved in linear time.

1 Introduction and Overview

In the register allocation problem one tries to assign hardware registers to the dynamically allocated variables (`auto`-variables in C) of a program written in a high level programming language. These variables might be explicitly defined by the program or implicitly generated to hold intermediate computational results. The allocation must be done in such a way that whenever two variables *interfere*, *i.e.* are needed at the same time during the execution of the program, they are found in different registers. In general this is not possible and some variables must be saved to the stack and then restored from there afterwards. This is called *spilling*. Spilling is time consuming so compilers try to avoid it whenever possible by assigning the same register to several variables that do not interfere.

Concrete machines have a fixed number of registers. It is an important task for a compiler written for a concrete machine to decide whether or not a given program (function, subroutine, etc.) can be realized on that machine without spilling. In this paper we give algorithms to achieve such a goal.

It is well known, see *e.g.* [1, 12], that the register allocation problem for a program P written in a high level programming language is equivalent to the problem of coloring its variable interference graph G_P. This graph has the variables of the program as vertices with an edge between two variables if their live variable ranges overlap, *i.e.* if there is a statement where both variables are needed in registers. Taking the set of available registers to be the set of available colors for G_P the translation to a coloring of G_P is immediate. Since coloring of graphs in general is NP-hard and any graph can appear as G_P, heuristics are usually applied to solve the problem, see [1].

The picture changes when certain restrictions on the programming language are imposed. The first result here seems to be in [15] where it is shown that for structured Pascal programs the graph G_P is in fact the intersection graph of subgraphs of a series-parallel graph. More recently, Kannan and Proebsting [13] gave a 2-approximation algorithm for the coloring of G_P of such a program P. This is a vast improvement over the general case, as under the assumption $NP \neq coRP$ general graph coloring is not approximable within a factor $n^{1-\epsilon}$ for any $\epsilon > 0$ [10].

But Pascal lacks many programming constructs, such as short circuit evaluation and multiple exits from loops and functions, that complicate the picture. Recently, Thorup [18] was able to show that even with those constructs, for so-called **structured programs**, that simply avoid `gotos`[1], G_P still has a lot of structure: it can be found as the intersection graph of subgraphs of a graph of treewidth at most 6. Thorup used that observation to give a 4-approximation algorithm for the coloring of G_P of such a program P.

In this paper we solve a problem with a different flavour. We are not interested in the minimum number of registers needed to realize a given program without spilling, but on the following *fixed parameter* problem:

[*] University of Utrecht, Department of Computer Science, Utrecht, P.O. Box 80 089, NL 3508 TB Utrecht, The Netherlands. Email: hansb@cs.ruu.nl.

[†] TU Berlin, Fachbereich Mathematik, Sekr. MA 6-1, D-10623 Berlin, Germany. Email: gustedt@math.TU-Berlin.DE. Supported by IFP *Digitale Filter*.

[‡] Universitetet i Bergen, Institutt for Informatikk, 5020 Bergen, Norway. Email: telle@ii.uib.no

[1] In the paper [13], structured is defined as just having a series-parallel control-flow graph. Thus the term "structured" has an *extended* meaning for us.

Uniform Register Allocation Problem

Instance: A structured program P.

Parameter: Integer w.

Question: Can P be realized without spilling using at most w uniform registers?

As the main result of this paper we are able to give a linear time algorithm for this problem:

Theorem 1. *For any uniform register allocation problem, with fixed w, there is a linear time algorithm that given a structured program P either finds a valid allocation with w registers or states that such an allocation does not exist. Herein rescheduling of statements that does not violate data dependencies is permitted.*

One main ingredient to the solution is Thorup's bound on the treewidth which in fact remains valid even if we allow rescheduling. Note that the restriction to structured programs is crucial, as otherwise we must solve w-coloring of general graphs, a problem NP-hard for any fixed $w \geq 3$ [11].

The approximation algorithms of [13, 18] mentioned above, deal with the 'macroscopic' view of a program, namely to solve the register allocation problem under the assumption that the local order of statements is fixed. This assumption is too strong for modern RISC architectures that gain much of their improved performance from the fact that statements might be reordered by the compiler, as long as the data dependencies are not violated, so-called **rescheduling**. It is known that if we allow rescheduling then the problem to determine the minimum number of registers needed is NP-hard even for straight-line code, *i.e.* code with no conditional branching or loops, [16]. Perhaps surprisingly, our algorithm is able to handle rescheduling of statements completely within the linear time bound, for a fixed number of registers w (with running time exponential in w.)

In the fixed parameter *non-uniform* register allocation problem we allow for special-purpose registers for variables of specific types, *e.g.* separate registers for integers and floats:

Non-Uniform Register Allocation Problem

Instance: A structured program P.

Parameter: Integers $w_1, ..., w_k$.

Question: Can P be realized without spilling using at most w_i registers for variables of type t_i?

In the case that we do not allow for rescheduling we can give a linear time algorithm for this problem. Even when allowing for rescheduling we are able to give a (high-order) polynomial-time algorithm. However, we show that we may then not expect to get an $O(n^c)$ algorithm where c is not dependent on the number of registers:

Theorem 2. *Non-uniform register allocation with rescheduling with at least two different types of registers is $W[t]$-hard for any $t > 0$.*

Our main algorithmic technique is to separate the problems into two parts. One that deals with the statements and variables that are involved in possible branchings of the program and another that deals with straight code in which no branching is involved. The different types of graphs we obtain all have bounded treewidth, if a solution with w registers exists.

Section 2 introduces the basic notation, gives the modelling for the straight line code part and for the remaining branching part and shows how both parts lead to graphs with bounded treewidth. Thorup[18] shows how parsing of a structured program in a straightforward way gives a tree-decomposition of its control-flow graph of width at most 6. In using this result for the variables involved in branchings of the program, our algorithms avoid the usual computation of the tree-decomposition [4] and thereby overcome the major obstacle to the practical use of bounded treewidth algorithms. For rescheduling of the straight-line part we apply in a pre-processing step the algorithm from [4] to compute a path-decomposition, but have hope that a good approximation could instead be used for practical purposes.

In Section 3 we use dynamic programming to solve the problems in polynomial time. The modelling is used relatively easily in Section 3.3 to give, in a brute force manner, a coloring (and thus register allocation) in polynomial time for the non-uniform register case. In Section 3.2 we are able to prove Theorem 1. The methods used are similar to techniques in [7] to solve problems on graphs of bounded treewidth.

Section 4 gives a brief introduction to the fixed parameter W-hierarchy and sketches a proof of the hardness result of Theorem 2 for the non-uniform register case.

2 Treewidth Modelling

In the following graphs may either be directed or undirected. We will use the standard notions of **path-** and **tree-decompositions** of undirected graphs and the corresponding parameters **pathwidth**, pw, and **treewidth**, tw. These notions are defined in *e.g.* [3]. We describe on assembly level the directed data flow graph $\vec{F}_P$ that models data dependencies of program P. Every assembler statement uses a certain set of registers for its input, the right hand side, **RHS**, and assigns values to another set of registers, the left hand side, **LHS**. Observe that these sets may overlap and that they may be empty for a particular statement.

The vertices of the data flow graph are the assembler statements and there is an arc from statement s to statement t if there is a register R with either $R \in \mathrm{LHS}(s) \cap \mathrm{RHS}(t)$ (def-use dependency) or $R \in LHS(s) \cap \mathrm{LHS}(t)$ (def-def dependency) or $R \in RHS(s) \cap LHS(t)$ (use-def dependency) and in each case there is some execution of the program where s is the last statement containing R as specified before statement t. For the sake of simplicity we do not consider other data dependencies such as memory aliasing. Since the program P may have loops the graph $\vec{F}_P$ is not necessarily a dag (directed acyclic graph).

High level programming languages abstract from assembler by introducing the concept of variables of the program. When translating such a program into assembler the variables are assigned to particular registers if possible. The definition of the data flow graph is easily extended to pseudo assembler, only that variables play the role that registers did before. For our purposes this translation process may be seen as a multi step procedure:

1. Pseudo assembler statements that still operate on the variables are generated.
2. The pseudo assembler statements are topologically sorted according to the data flow graph, **statement scheduling**.
3. Variables are mapped to hardware registers if possible, **register allocation**.

Our main concern in this paper is the last step, the register allocation. But since statement scheduling greatly influences the possibilities for allocation we are forced to consider both problems simultaneously.

A variable x is **alive** in statement u if x is either directly affected by u or in some execution of the program there is a def-use dependency on x from a statement preceding u to a statement succeeding u.

The **live range** of variable x is the set of all statements for which x is alive and is usually computed by exploring the flow graph in a reverse depth-first ordering. For the sake of simplicity we assume that the live range of a variable x forms a connected subgraph of $\vec{F}_P$. Otherwise we may easily reformulate the program by distinguishing different incarnations of x.

Two variables x and y **interfere** if their live ranges intersect, *i.e.* if there is a statement u where both variables are alive. It is easy to see that if x and y interfere they may not be assigned to the same register. The exceptional case is a variable copy statement s : $x = y$ where s is the last (resp. first) statement in the live range of x (resp. y) in which case we simply identify the two variables and drop s. In fact such anomalies disappear with a modelling of F_P that is a bit more involved, but which exceed the possibilities of this abstract. The **variable interference graph** G_P is now defined with the set of variables as vertices and an edge between two variables x and y iff x and y interfere.

2.1 The Straight Line Parts

In this section we show how to model the problem for pieces of code that do not contain any branching operations. The solutions for these pieces of code will then be used later as a subroutine to solve the problem as a whole.

A subsequence of the pseudo assembler is called **straight line code** if it does not contain any jump statements (but the last) and if none of the statements (but the first) are jumped to. The maximal subsequences of straight line code of a program P are the **basic blocks** of P.

To each basic block B there corresponds a subgraph $\vec{F}_P|B$ of the data flow graph. Unfortunately, this graph is not necessarily a dag: jump statements that follow the basic block may cause data flow from the end of a basic block to its beginning. To handle this situation we assume that each basic block has `begin` and `end` statements. The variables that are alive for the `begin` and `end` statement respectively are the **inflow** and **outflow** of the block. On the level of the block, inflow resp. outflow variables are seen as if they were the LHS of the `begin` resp. the RHS of the `end` statement.

To be able to reduce the scheduling and allocation problems on a basic block B to a dag we compute the data flow graph for this basic block –as if it were a program of its own– resulting in the **constrained data flow graph** $\vec{F}_B$. Since the basic block does not contain any branching statement this graph is in fact a dag. The only additional information that we have to keep is which variables of the inflow and outflow are to be identified, the **border constraints**. When computing a solution for the constrained graph we have to ensure that the same register is used for a particular variable as it appears in the inflow and in the outflow. In the middle of a basic block this might well be relaxed.

In general we may assume that each variable is initialised exactly once inside a basic block B. This is so, since in a statement t that initializes a variable x the previous value of x –if present– is overwritten. Thus if x is initialized in statements $t_1, t_2, \ldots, t_k$ (in order of the original specification of B) it can be replaced by different incarnations $x_{t_1}, x_{t_2}, \ldots, x_{t_k}$. This means that we do not have def-def dependency arcs in $\vec{F}_B$.

Moreover, $\vec{F}_B$ will not contain use-def dependency arcs from s to t either, as a variable that is used before initialization must have been part of the inflow, and thus we have a def-use dependency arc from the `begin` statement to s, and the variable initialized in statement t is considered a different incarnation.

The only side constraint we have to worry about, is that if x is part of the inflow *and* part of the outflow of the block the registers assigned to x_{t_1} and x_{t_k} must be the same.

So in the following we may assume that every variable of a basic block is initialized exactly once, and that $\vec{F}_B$ contains only def-use dependency arcs. Many computer architectures, *e.g.* all modern RISC architectures, see [12], also ensure that in turn any statement initializes at most one register. Since such an assumption eases the following discussion we will assume that this is the case, and only mention at the end how the results can be extended to the more general case.

It is easy to see that any rescheduling and subsequent register allocation that obeys the border constraints is valid for the original problem and, vice versa, that every valid solution for the original problem leads to an equivalent one for the constrained problem.

For each basic block B any feasible sequencing of the assembler statements corresponds to a topological sort of $\vec{F}_B$ and vice versa. Observe that the set of topological sorts of $\vec{F}_B$ does not change if we add or delete transitivity edges to it. The number of registers needed translates to the **directed vertex separation number** of $\vec{F}_B$, which is defined as follows:

Let $D = (V, A)$ be a dag. For a topological sort $v_1, \ldots, v_n$ of D the **separator** at i is the set of vertices

- that have a number $j \leq i$ in the sort
- such that there is $(v_j, v_k) \in A$ with $i \leq k$.

The **vertex separation width** at i is $|\text{sep}(i)|$, and indicates how many variables are alive at i (recall that A contains only def-use dependency arcs). The vertex separation number for a given sort is the maximum width over all i and the directed vertex separation number of D, $\text{dvs}(D)$, is the minimum over all topological sorts. It is closely related to the vertex separation number of an undirected graph G, $\text{vs}(G)$, a parameter defined in an analogous way but instead of only allowing topological sorts this parameter is a minization over all linear orderings of the vertices of G. If G_D is the undirected graph obtained from D by dropping the directions on the arcs we therefore have $\text{vs}(G_D) \leq \text{dvs}(D)$. It is well-known that the undirected vertex separation number and pathwidth parameters coincide (McKinnersley [14]), so that we can bound $\text{pw}(G_D)$ by $\text{dvs}(D)$.

Lemma 2.1. *Let D be a dag and G_D its underlying undirected graph. Then $pw(G_D) = vs(G_D) \leq dvs(D)$.*

Consider now the **statement interference graph** F_B, the underlying undirected graph of $\vec{F}_B$. The presence of an edge $\{s, t\}$ in F_B means that either a result of s is needed in t or vice versa. So in particular this means that some variables initialised by s and t interfere. We can now state the main result of this section that for the cases of interest the graph modelling the straight-line code has bounded pathwidth (and hence also bounded treewidth.)

Corollary 2.2. *If the variables of a basic block B can be allocated to at most w registers then the directed vertex separation number of $\vec{F}_B$ is at most w and $pw(F_B) \leq w$.*

It is relatively easy to see that a similar result holds even without the restriction that all assembler statements initialise at most one register. This is so, since in that case we can show that F_B is a minor of G_B and the pathwidth parameter cannot increase by taking minors.

So far in this graph theoretic model the particular assignment of a specific register to a variable has not yet been important; we have only been interested in the number of registers needed at a certain time. To be able to use the constrained problem on basic blocks as a subroutine, we must add the border constraints to the problem. These will ensure that in the topological sort corresponding to a coloring the coupled variables of the in- and outflow are colored the same. This problem is dealt with in Section 3.2.

2.2 The Branching Part

We now turn to the branching part of the program. Since rescheduling is not an issue here, we focus on the control flow and not the data flow of the program. In general the control-flow graph and hence also the variable interference graph of a program can be arbitrary graphs. However, in a recent result [18], Thorup shows that for programs written in high-level languages that avoid goto's the underlying undirected graph of the (directed) control-flow graph has bounded treewidth:

Theorem 3. *(Thorup [18]) The control-flow graphs of structured (i.e. **goto**-free) C programs have treewidth bounded by 6, and similar results for other languages are Algol (3), Pascal (3), Modula-2 (5).*

The differing bounds for different languages are due to the presence or not of language constructs like short-circuit evaluation of boolean functions, multiple returns from functions and multiple exits from loops. For example, without short-circuit evaluation, the bounds on the treewidth in the results above will for each language drop by one [18].

The paper [18] also shows how treewidth is preserved under standard compiler optimizations, including rescheduling. We consider the pseudo-assembler program realizing the structured high-level program and assume that this translation does not introduce any unnecessary branching, so that the structural aspects of branching statements are preserved. For example a break or continue statement inside a loop which in the high-level language involves branching to the end or to the beginning of the loop translates in the pseudo-assembler code simply to a jump to the corresponding positions.

The **reduced control flow graph** of an assembler program has statements as vertices that are begin's and end's of basic blocks. *i.e.* the first and last statements of the basic blocks. Arcs in this graph specify that a basic block can immediately follow another in some execution of the program, and are given by the jumps: there is an arc from statement s to statement t if

1. s and t are the begin and end statements of a single basic block
2. end statement s is a jump and begin statement t is its target
3. end statement s is a conditional jump or not a jump statement and begin statement t follows it in the order of the program.

Arcs of type 1, the **straight line arcs**, correspond to basic blocks of the program.

In general only a subset of all variables will be alive at statements of the reduced control-flow graph, the **branching variables**. The other variables only play a role in what register combinations they enforce between the in- and outflow of a basic block.

So if we assume that we have solutions for all possible register combinations on basic blocks that correspond to arcs of type 1 we may place this information on the corresponding arcs of the reduced control-flow graph. The **branching problem** we are now left with is then to find an assignment of registers to the branching variables s.t. for every straight line arc of the flow control graph there is a valid pairing constraint for the basic block.

The reduced control-flow graph has vertices corresponding to *assembler* statements that are sources or targets of jumps. The vertices of this graph and its jump arcs can be identified with vertices and arcs of the control-flow graph of the *high-level* language, while the arcs representing basic blocks can be identified with paths of the latter graph that have been contracted to a single edge. The pseudo-assembler reduced control-flow graph is therefore a minor of the high-level control-flow graph. Since it is well-known that treewidth can only decrease by taking minors, we have:

COROLLARY 2.3. *The reduced control-flow graph of a structured program has bounded treewidth.*

For fixed values of k there exists a linear-time algorithm [4] which finds, if possible, a tree-decomposition of width k of an input graph, but unfortunately this algorithm is not practical for values of $k \geq 4$. However, in the case of control-flow graphs the parsing of the program can be used to quite easily discern their tree structure. Indeed, the paper of Thorup [18] shows how a small width tree-decomposition of these graphs can be found in a straighforward manner in linear-time from three-address code, and a similar technique will work for the reduced control-flow graphs given by pseudo-assembler code.

Now we show that, for fixed w, if w registers suffices to allocate variables of the reduced control flow graph without spilling, then its variable interference graph has bounded treewidth.

The live variable ranges correspond, for each variable, to a subgraph of the reduced control-flow graph. The variable interference graph is the intersection graph of live variable ranges, hence by Corollary 2.3 it is an intersection graph of a family of subgraphs of a bounded treewidth graph. If the bound on the treewidth is 1, then the intersection graph is a chordal graph and for this class of graphs it is well-known that treewidth equals chromatic number. In the general case, we get a slightly weaker result:

LEMMA 2.4. *If H is an intersection graph of a family of subgraphs of a treewidth-k graph G and the chromatic number of H is at most w, then the treewidth of H is at most $(k+1)w-1$*

Proof. At most w subgraphs of G corresponding to vertices in H can share a common vertex of G, since otherwise H has a clique of $w+1$ vertices and is not w-colorable. We build a tree-decomposition of H in the following way: take a width k tree-decomposition of G, every bag has at most $k+1$ vertices. Replace every vertex in every bag by the vertices of H whose subgraph contain that vertex. This gives a tree-dec of H, and by the previous observation, no bag has more than $(k+1)w$ vertices.

The treewidth upper bound of $(k+1)w-1$ is not optimal, but we can show that treewidth is lower bounded by $kw/2$. We omit the proof of this fact for the sake of brevity.

3 The Algorithms

This section develops the techniques that are necessary to achieve polynomial (and even linear) time algorithms. The principal ideas are the following.

For the branching part we will not use fewer registers by reordering *e.g.* the `then` and the `else` parts of an `if-then-else-fi` construct; valid colorings of the variable interference graph stand in one-to-one correspondence with valid register allocations. For the straight-line part however, reordering of the statements is really important and in fact this makes the problem more difficult to treat, both in the sense that we obtain W-hardness results for the non-uniform register problem and in the sense that the linear time algorithm for the uniform register problem is more complicated.

The two principal parts of the problem solution – straight-line and branching– are combined by what we call the *border constraints* of a basic block. In the branching part, a basic block B is represented simply by an arc (s, t) where statement s can be taken to be the `begin` statement of B and t to be the `end`. Consider a variable x of the inflow but not the outflow of B, and conversely variable y of the outflow but not the inflow of B. A solution of the branching part might try to assign the same register to both variables, but the only w-colorings of B may be ones where x and y are assigned different registers (colored with different colors).

There are two reasons we are able to handle this combination of the two parts in linear time:

1. The size of the inflow and outflow are both bounded by a constant so if we are able to solve any particular instance of a block B in linear time, doing so for different pairings only multiplies the running time by a constant factor.

2. In the tree-decomposition of the variable interference graph for the branching part all variables of the inflow and outflow of a particular basic block B belong to the same node of the tree-decomposition and hence in the dynamic programming algorithm they are dealt with as a set, so that in any case all combinations of pairings are considered.

Section 3.1 describes a linear-time algorithm for the branching part, for both uniform and non-uniform registers. This algorithm relies on an oracle that tells whether or not certain combinations of pairing of variables are feasible – and the subroutine for this is given in the next two subsections. Section 3.2 the linear-time straight-line algorithm for the uniform register case, and Section 3.3 gives the polynomial-time straight-line algorithm for the case of non-uniform registers.

3.1 The Branching Problem

Since w-coloring for graphs of bounded treewidth can be solved in linear time by standard dynamic programming techniques, see *e.g.* [2], Lemma 2.4 has the corollary:

COROLLARY 3.1. *w-coloring for fixed w is linear for any class of graphs that are intersection graphs of subgraphs of bounded treewidth graphs.*

See [3] for definitions of graphs of bounded treewidth k, also called partial k-trees. In [17] w-coloring of partial k-trees is solved by a linear time algorithm having a constant whose dependency on w and k is $w^{2(k+1)}$. To handle registers of different types we modify the coloring algorithm in a straightforward manner to a restricted coloring algorithm where certain vertices can only receive certain colors. Were it not for the border constraints, this would suffice to solve the branching part of our problem.

To deal with the border constraints is rather straight forward, as mentioned in Section 3. We use dynamic programming similar to the algorithms mentioned above, except that we weed out, by a call to an oracle – the subroutine for the straight-line code – those table entries that correspond to solutions where variables of the border constraints of a basic block are given the same color in such a combination that does not allow a w-coloring of the basic block. A block is represented by an arc in the reduced control-flow graph and hence all vertices playing a role in a single border constraint belong to a clique of the variable interference graph, and are a set in some of the tables that are generated. This gives a linear-time algorithm.

3.2 The Straight Line Problem

In this subsection we will complete the proof of Theorem 1, by showing that for the uniform register case the straight-line part can be solved in linear time. This is a consequence of the following result.

THEOREM 4. *For each fixed w, there exists a linear time algorithm that, when given a dag D, decides whether the directed vertex separation number of D is at most w, and if so, finds a topological sort of D with vertex separation width at most w.*

Our algorithm uses Lemma 2.1: if the directed vertex separation number of D is at most w, then the pathwidth of the underlying undirected graph G_D is at most w. Thus, we start by finding a path-decomposition of G_D of width at most w, or, equivalently, a linear ordering f of G_D with vertex separation width at most w. If these do not exist, then we stop and output this fact.

The algorithm from [4] can be used for finding the path-decomposition; using the characterization from [14], the path-decomposition can be transformed to the linear ordering of G_D. Both steps cost linear time, but unfortunately the first has a high constant factor.

A more practical approach is to not compute the pathwidth with the algorithm from [4] but to use instead a topological sort of D produced by some good heuristic for the straight line problem. The hope is that this heuristic has directed vertex separation number close to optimality, and hence also is more or less to be taken as a constant.

So, the problem we are left with can be stated as follows: we have a linear ordering $f : V \to \{1, 2, \ldots, n\}$ of the undirected graph G_D with vertex separation width at most w, and want to find a topological sort of D with vertex separation width at most w, or decide that such a topological sort does not exist.

Given a topological sort g of an induced subdigraph $H = (W, B)$ of $D = (V, A)$, we say g can be **extended** to a topological sort of D with vertex separation width at most w, if the vertices in $V \setminus W$ can be inserted in the sort g, yielding a topological sort of D whose vertex separation width is at most w. A **spot** in a topological sort is a position between two successive vertices in the sort, or the position before or after all vertices. We say a topological sort is extended at set of spots S, iff it can be extended by inserting vertices only in these spots. Let D_i be the subdigraph of D induced by the vertices in $f^{-1}\{1, 2, \ldots, i\}$, i.e. the first i vertices in f.

The crux of the algorithm is the following. For each topological sort g of an induced subdigraph D_i, we can point out a constant size set of spots S, where the constant quadratically depends on w. S has the property that g can be extended, iff g can be extended at S. So we may assume we will only insert vertices in spots in the set S.

For a topological sort g of induced subdigraph $H = (W, B)$, the following procedure computes this set of spots S: Let Z be the set of vertices in W, adjacent to a vertex not in W. When working with a path decomposition of width w, we have that $|Z| \leq w+1$. Say that $v \in Z$ *belongs* to a certain gap, if v appears before the gap, and a neighbor of v appears after the gap. The gaps to which a vertex $v \in Z$ belong will be consequtive. Thus, we can partition the gaps in g in at most $2w + 3$ ranges, where every range is a set of consequtive gaps with the same set of vertices in Z belonging to each gap in the range. In each range, we write down the separation widths at the gaps (or more precisely, the number of vertices belonging to the gaps.) Thus, for each range, we have a sequence of integers. When we have a subsequence $s_i, s_{i+1}, \ldots, s_j$ of such integers with s_i, s_j the smallest and largest integer in the subsequence or vice versa, we can remove the integers $s_{i+1}, \ldots, s_{j-1}$, from the sequence. Repeat this operation until it is no longer possible. The gaps corresponding with the resulting integers are placed in S.

In essence, these ideas borrow from a technique given in [7], and correctness can be proven using results and techniques from that paper; but in the current framework, the result is both more practical and easier to understand. It also follows from [7] that S will have $O(w^2)$ gaps.

It can be shown that for each subdigraph D_i, there is a constant sized set of topological sorts, **the full set of partial solutions** of D_i, such that if D has directed vertex separation number at most w, then one of the elements of this set can be extended. (Basically, if two topological sorts have the same ranges and sequences of integers in each corresponding range, as computed above, then only one needs to be in this set.) Moreover, we can compute a representation of the set for D_{i+1} in constant time from a similar representation of the set for D_i. Then, the linear time algorithm works as follows:

- We compute first a full set of partial solutions for D_1, then for D_2, etc.
- When we have a full set for D_n, we can already decide whether the vertex separation of $D = D_n$ is at most w.
- With some bookkeeping we can also construct a topological sort with vertex separation width at most w if one exists.

Additional technical ideas are needed to deal with the border constaints, which enforce that a variable of the inflow and another of the outflow, which were renamed for the sake of lowering the register usage, are actually one and the same and need to be assigned to the same register. The main idea here is that when two registers are both free at the same time, then they must no longer be distinguished. In fact any border constraints concerning those two registers can be fulfilled if the two corresponding variables are assigned any of those registers. In general a partial solution can be characterized in addition to what is stated above with the relaxation of the border constraints that it allows. Since the possible set of border constraints is a constant, only depending on w, there are only a constant number of positions i where these may change. So there are at most a constant number of spots where the additional relaxation of the border constraints must be kept track of.

The algorithm also allows for a solution method that does not necessarily have a constant factor coming close to the upper bound. For each i, we keep a subset of a full set of partial solutions. We start with only one element in D_1. We always can take an element from one of the sets, say D_i, insert the vertex $f^{-1}(i+1)$ in one of the $O(1)$ spots, and put the resulting topological sort in the set for D_{i+1}, unless that set contained an element with the same characteristic, or the vertex separation width of the sort became too large. The aim is to get an element of D_n. Different strategies can be tried here: the dynamic programming algorithm corresponds to a 'breadth first search' strategy, but other methods could, at least on the average, lead faster to a solution.

3.3 Non-uniform Registers If we do not allow for rescheduling, then we need only concentrate on the control-flow graph. Therefore, as discussed in Section 3.1, a restricted coloring of bounded treewidth graphs solves the problem of fixed parameter non-uniform register allocation without rescheduling, in linear time. Now we indicate how to solve this problem while allowing for rescheduling, by a polynomial time algorithm where the exponent is a function of the number of registers. The previous discussion showed that we may restrict ourselves to straightline code as input.

So suppose we are given the dataflow graph $\vec{F}_B$ of a block B. A topological sort $s_1, s_2, \ldots, s_n$ that corresponds to a solution using at most w registers defines a sequence of separators, $\operatorname{sep}(1), \operatorname{sep}(2), \ldots, \operatorname{sep}(n)$, cf. Section 2.1, that are all of size at most w. We will do dynamic programming on $\vec{F}_B$ such that the family of sets of vertices of size at most w can basically be identified with the set of states of the dynamic programming: every such set will occur at most twice as such a separator. So in total there will be less than $O(n^w)$ states that are visited by the algorithms.

The dynamic programming is done in phases, each phase corresponding to a position i in the topological sort of $\vec{F}_B$. A subset P of the statements is called an *initial segment* if for all statements s that have a path in $\vec{F}_B$ to some element in P we already have that $s \in P$. In other words an initial segment P is a set of statements for which there exists a topological sort of $\vec{F}_B$ such that the elements of P form the first $|P|$ elements of the sort.

To every initial segment P there corresponds a separator $S(P)$, namely the subset $S \subseteq P$ of statements in P that have an arc leading to the outside of P. Any state s of the dynamic programming will have an initial segment P_s that is associated with it and that gives rise to a separator $S_s = S(P_s)$ of size less than w.

For a set of statements S, define $\operatorname{Pred}(S)$, the *open* predecessor set, to be the set of statements that have a path to any of the elements S in $\vec{F}_B$ and $\operatorname{Pred}[S] = \operatorname{Pred}(S) \cup S$, the *closed* predecessor set. If P is an initial segment then $\operatorname{Pred}[S(P)] = P$.

To start a dynamic programming algorithm to solve the problem, first compute the set of minimal statements, i.e those that can immediately be executed after the `begin`. Any of these may be chosen first so any set $\{x\}$ with x among these minimal elements constitutes a possible initial state.

Then in phase i we take the set of all reachable states, *i.e.* all possible separators at position i, and compute the set for $i+1$. Here we can go from a state s on level i to s' on level $i+1$ iff

- $|S_{s'}| \leq w$,
- $P_{s'}$ has just one element more than P_s (this is the vertex that we can place next in the linear order).

The solution is obtained if there is a final state that can be reached in phase n. This approach leads to a polynomial time algorithm if we are able to bound the the number of states by a polynomial in n.

Therefore let us assume first that we don't have statements on which the outflow doesn't depend, *i.e.* we have no `store` statements. Any set S may then only occur as a separator for exactly one such phase. Indeed the phase in which S may occur corresponds to $|\operatorname{Pred}[S]|$. So clearly the number of states then is bounded by $O(n^w)$.

The general case must be handled with more care. The problem is that for a separator S there may be some set of `store`s T that only depend on S or a subset and may thus be scheduled in any order after S. Then after having processed the first element in T we would again reach a state that has S as its separator and so forth. This could blow up the set of possible states in a non-polynomial way.

To handle such a situation we need some new definitions. For a set S call the statements t that lie on a non-trivial path from an $s \in S$ to some member of the outflow of B the *strict* successors of S. Denote the set of strict successors with $\mathit{Strict}(S)$. $\mathit{Strict}(S)$ is such that whenever we have scheduled any element $t \in \mathit{Strict}(S)$, S will never occur as a separator again. A separator S itself is called *strict* if every of its elements lie on a path to the outflow of B.

We modify the dynamic programming such that the separators that are associated to states will always be strict. Other separators and thus initial segments that naturally may occur in a solution are dealt with differently.

Call a statement $t \notin S$ a *weak* successor of S if there is a path from some element in S to t and if $t \notin \mathit{Strict}(S)$. These weak successors must not necessarily be `store`s but may also represent some intermediate computations that are needed to store a particular value. Now call a set W of weak successors of S *feasible* for S if

- W can be scheduled immediately after S if S is the current separator and by respecting the given constraints on registers, and
- after having scheduled W, S may act again as the actual separator.

Now observe that if we have two such weak successor sets W and W' that are feasible for S then $W \cup W'$ also is such a feasible weak successor set. Thus there exist a unique maximal feasible weak successor set $Weak(S)$ for S.

The first rule that we impose now to reduce the possible number of states is the following

greedy rule Whenever we have that a strict set $S = S_s$ is the separator of a state s we schedule $Weak(S)$ immediately after s and proceed with the dynamic programming on the state that is given by $P_s \cup Weak(S)$.

The second rule is the following

lazy rule Whenever we find in phase i that a set $S_{s'}$ of a state s' that we want to visit by the dynamic programming has been used as a separator for some phase $i' < i$ we discard s' from the set of possible states.

This lazy rule may be used since it can be shown, that for a strict set S that occurs as a separator at some point there does indeed exist a unique first state s where it occurs first in the dynamic programming.

The third and fourth ingredients that we need to turn all of this into a polynomial time algorithm are the ability to compute for every appropriate S

- the set $Weak(S)$ and
- a feasible ordering for it.

This can be done by another dynamic programming, that goes backward from sets of store statements to the possible strict sets S.

A detailed discussion of that approach lies far beyond the possibilities of this extended abstract and must be delayed to the full version of this paper.

So together Section 3.1 we obtain:

THEOREM 5. *The fixed parameter register allocation problem for non-uniform registers can be solved in linear time without rescheduling and in polynomial time with rescheduling.*

4 Hardness of Non-uniform Registers

An interesting question for many problems whose instance has a parameter k is: what is the complexity of the problem when the parameter k is fixed? This is precisely the question we have been asking in this paper for the register allocation problems. One would like to distinguish not only between polynomial time solvability and (conjectured) needing exponential time (using the theory of NP-completeness), but also between behavior where the time is

- $O(n^c)$ with c a fixed constant, *i.e* all dependency on the constant factor k is hidden in the 'O', and
- $\Omega(n^{f(k)})$, where f grows with k, *i.e* the exponent depends on k.

To distinguish between these latter two behaviors, Downey and Fellows called the class of problems having algorithms of the first type FPT (Fixed Parameter Tractable) and introduced the theory of fixed-parameter complexity, see *e.g* [9, 8]. They introduced a hierarchy of classes of parameterized problems, with $FPT \subseteq W[1] \subseteq W[2] \subseteq \cdots \subseteq W[P]$. It is conjectured that the hierarchy is proper, and hence that all problems hard for $W[1]$ (for definition of reductions see *e.g.* [8]) do not belong to FPT, *i.e*, the running time to solve those problems has the parameter in the exponent above the problem size.

We consider the following restricted version of our straight line code optimization problem – but now with the restriction that variables have specific types of registers that they must be assigned to. For technical reasons we still keep the assumption that any statement initializes at most one variable. Thus we may label such a statement s with $c(s)$, the required type for the variable that is initialized by it.

In [5], a version of this problem (in the abstract graph setting) was proved to be hard for all classes $W[t]$, for all $t \in \mathbf{N}$: there all registers were of a different type and each variable had one specific register it could be assigned to. Here we look at the more natural problem where the number of different types of registers is small, but larger than one. In fact, we are able to show that the problem becomes hard (in the fixed parameter sense), even when there are two different types of registers. A typical example of this is when there are floating point registers, and registers for integer objects like integers, pointers or booleans.

If we have a topological sort f of D and a particular type ζ of registers the **ζ-vertex separation width** at i, $\mathrm{width}_{\zeta,f}(i)$, is defined analogously as $\mathrm{width}(i)$ for the uniform problem, but with the difference that only variables of type ζ are taken into account. Thus it is the number of vertices v such that $c(v) = \zeta$, $f(v) \leq i$ and such that there is an arc $(v, v') \in A$ with $i \leq f(v')$. Following arguments as in Section 2, one can see that the problem is equivalent to the following:

2-Color Directed Vertex Separation Number Problem

Instance: Directed acyclic graph $G = (V, E)$, coloring $c : V \rightarrow \{1, 2\}$.

Parameter: The pair (k_1, k_2).

Question: Is there a topological sort f of G, such that for any color $\zeta \in \{1, 2\}$, and for all i, $1 \leq i \leq n$ we have that $\text{width}_{\zeta,f}(i) \leq k_\zeta$?

Theorem 6. *2-color directed vertex separation number is hard for $W[t]$, for all $t \in$ **N**, for graphs with indegree at most 2.*

This can be proven by using a reduction to the problem of finding a topological sort of a graph whose edges are colored with one of two colors such that for each color, the cutwidth is bounded by a given parameter. This latter problem can be shown to be $W[t]$-hard by a reduction to a directed variant of bandwidth, see [6]. We omit the proofs for the sake of brevity. Theorem 2 follows as a corollary.

The restriction to indegree 2 strengthens the result as it thereby applies to straight-line problems where assembler statements never use more than a fixed amount of registers on the RHS, which indeed is the case for many modern RISC architectures, see [12].

4.1 Acknowledgements We thank the anonymous referee for helping to formalize several important details.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques and Tools*, Addison-Wesley, 1986.

[2] S. Arnborg and A. Proskurowski, *Linear time algorithms for NP-hard problems restricted to partial k-trees*, Discrete Appl. Math., 23 (1989), pp. 11–24.

[3] H. L. Bodlaender, *A tourist guide through treewidth*, Acta Cybernetica, 11 (1993), pp. 1–23.

[4] ——, *A linear-time algorithm for finding tree-decompositions of small treewidth*, SIAM J. Comput., 25 (1996), pp. 1305–1317.

[5] H. L. Bodlaender, M. R. Fellows, M. T. Hallett, H. T. Wareham, and T. J. Warnow, *The hardness of problems on thin colored graphs*, Tech. Rep. UU-CS-1995-36, University of Utrecht, Utrecht, 1995. Submitted for publication.

[6] H. L. Bodlaender, M. R. Fellows, and T. J. Warnow, *Two strikes against perfect phylogeny*, in Proceedings 19th International Colloquium on Automata, Languages and Programming, Berlin, 1992, Springer Verlag, Lecture Notes in Computer Science, vol. 623, pp. 273–283.

[7] H. L. Bodlaender and T. Kloks, *Efficient and constructive algorithms for the pathwidth and treewidth of graphs*, Journal of Algorithms, 21 (1996), pp. 358–402.

[8] R. G. Downey and M. R. Fellows., *Fixed parameter tractability and completeness II : On completeness for $W[1]$*, Theoretical Computer Science, 141 (1995), pp. 109–131.

[9] R. G. Downey and M. R. Fellows, *ixed-parameter tractability and completeness I: Basic results*, SIAM J. Comput., 24 (1995).

[10] U. Feige and J. Killian, *Zero-knowledge and the chromatic number*, in Proceedings of the 11th Annual Conference on Structure in Complexity Theory, IEEE, 1996.

[11] M. Garey, D. Johnson, and L. Stockmeyer, *Some simplified np-complete graph problems*, Theoretical Computer Science, (1976), pp. 237–267.

[12] J. L. Hennessy and D. A. Patterson, *Computer Architecture, A Quantitative Approach*, Morgan-Kaufmann Publishers, 2^{nd} ed., 1996.

[13] S. Kannan and T. Proebsting, *Register allocation in structured programs*, in Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, California, 22–24 Jan. 1995, pp. 360–368.

[14] N. G. Kinnersley, *The vertex separation number of a graph equals its path width*, Inform. Process. Lett., 42 (1992), pp. 345–350.

[15] T. Nishizeki, K. Takamizawa, and N. Saito, *Algorithms for detecting series-parallel graphs and D-charts*, Trans. Inst. Elect. Commun. Eng. Japan, 59 (1976), pp. 259–260.

[16] R. Sethi, *Complete register allocation problems*, SIAM J. Comput., 4 (1975), pp. 226–248.

[17] J. Telle and A. Proskurowski, *Algorithms for vertex partitioning problems on partial k-trees*, SIAM J. Disc. Math., (1997). to appear.

[18] M. Thorup, *Structured programs have small tree-width and good register allocation*, in Graph-Theoretic Concepts in Computer Science, 23rd International Workshop WG '97, Möhring et al., eds., vol. 1198 of Lecture Notes in Computer Science, Springer Verlag, 1997, pp. 318–332. journal version accepted for *Information and Computation.*