Optimal Register Allocation in Polynomial Time

Philipp Klaus Krause

Goethe-Universität, Frankfurt am Main

Abstract. A graph-coloring register allocator that optimally allocates registers for structured programs in polynomial time is presented. It can handle register aliasing. The assignment of registers is optimal with respect to spill and rematerialization costs, register preferences and coalescing. The register allocator is not restricted to programs in SSA form or chordal interference graphs. It assumes the number of registers is to be fixed and requires the input program to be structured, which is automatically true for many programming languages and for others, such as C, is equivalent to a bound on the number of goto labels per function. Non-structured programs can be handled at the cost of either a loss of optimality or an increase in runtime. This is the first optimal approach that has polynomial runtime and works for such a huge class of programs.

An implementation is already the default register allocator in most backends of a mainstream cross-compiler for embedded systems.

Keywords: register allocation, tree-decomposition, structured program.

1 Introduction

Compilers map variables to physical storage space in a computer. The problem of deciding which variables to store into which registers or into memory is called register allocation. Register allocation is one of the most important stages in a compiler. Due to the ever-widening gap in speed between registers and memory the minimization of spill costs is of utmost importance. For CISC architectures, such as ubiquitous x86, register aliasing (i. e. multiple register names mapping to the same physical hardware and thus not being able to be used at the same time) and register preferences (e. g. due to certain instructions taking a different amount of time depending on which registers the operands reside in) have to be handled to generate good code. Coalescing (eliminating moves by assigning variables to the same registers, if they do not interfere, but are related by a copy instruction) is another aspect, where register allocation can have a significant impact on code size and speed.

Our approach is based on graph coloring and assumes the number of registers to be fixed. It can handle arbitrarily complex register layouts, including all kinds of register aliasing. Register preferences, coalescing and spilling are handled using a cost function. Different optimization goals, such as code size, speed, energy consumption, or some aggregate of them can be handled by choice of the cost function. The approach is particularly well-suited for embedded systems, which often have a small number of registers, and where optimization is of utmost

K. De Bosschere and R. Jhala (Eds.): CC 2013, LNCS 7791, pp. 1–20, 2013.

importance due to constraints on energy consumption, monetary cost, etc. We have implemented a prototype of our approach, and it has become the default register allocator in most backends of sdcc [12], a mainstream C cross-compiler for architectures commonly found in embedded systems. Virtually all programs are structured, and for these the register allocator has polynomial runtime. This is the first optimal approach that has polynomial runtime and works for such a huge class of programs.

Chaitin's classic approach to register allocation [7] uses graph coloring. The approach assumes r identical registers, identical spill cost for all variables, and does not handle register preferences or coalescing. Solving this problem optimally is equivalent to finding a maximal r-colorable induced subgraph in the interference graph of the variables and coloring it. In general this is NP-hard [8]. Even when it is known that a graph is r-colorable it is NP-hard to find a r-coloring compatible with a fraction of $1 - \frac{1}{33r}$ of the edges [18]. Thus Chaitin's approach uses heuristics instead of optimally solving the problem. It has been generalized to more complex architectures [31]. The maximum r-colorable induced subgraph problem for fixed r can be solved optimally in polynomial time for chordal interference graphs [27,35], which can be obtained when the input programs are in static single assignment (SSA) form [20]. Recent approaches have modeled register allocation as an integer linear programming (ILP) problem, resulting in optimal register allocation for all programs [17,15]. However ILP is NP-hard, and the ILP-based approaches tend to have far worse runtime compared to graph coloring. There are also approaches modeling register allocation as a partitioned boolean quadratic programming (PBQP) problem [30,22]. They can handle some irregularities in the architecture in a more natural way than older graph-coloring approaches, but do not handle coalescing and other interactions that can arise out of irregularities in the instruction set. PBQP is NP-hard, but heuristic solvers seem to perform well for many, but not all practical cases. Linear scan register allocation [28] has become popular for just in time compilation [13]; it is typically faster than approaches based on graph coloring, but the assignment is further away from optimality. Kannan and Proebsting [23] were able to approximate a simplified version of the register allocation problem within a factor of 2 for programs that have series-parallel control-flow graphs (a subclass of 2-structured programs). Thorup [32] uses the bounded tree-width of structured programs to approximate an optimal coloring of the intersection graph by a constant factor. Bodlaender et alii [4] present an algorithm that decides in linear time if it is possible to allocate registers for a structured program without spilling.

Section 2 introduces the basic concepts, including structured programs. Section 3 presents the register allocator in its generality and shows its polynomial runtime. Section 4 discusses further aspects of the allocator, including ways to reduce the practical runtime and how to handle non-structured programs. Section 5 discusses the complexity of register allocation and why certain NP-hardness results do not apply in our setting. Section 6 presents the prototype implementation, followed by the experimental results in Section 7. Section 8 concludes and proposes possible directions for future work.

2 Structured Programs

Compilers transform their input into an intermediate representation, on which they do many optimizations. At the time when register allocation is done, we deal with such an intermediate representation. We also have the *control-flow graph* (CFG) (Π, K), with node set Π and edge set $K \subseteq \Pi^2$ which represents the control flow between the instructions in the intermediate representation. For the code written in the C programming language from Figure 1, the sdcc compiler [12] generates the CFG in Figure 2(a). In this figure, the nodes of the the CFG are numbered, and annotated with the set of variables alive there, and the intermediate representation. As can be seen, sdcc introduced two temporary variables, which make up the whole set of variables $V = \{a, b\}$ to be handled by the register allocator for this code.

```
#include <stdint.h>
#include <stdbool.h>

bool get_pixel(uint_fast8_t x, uint_fast8_t y);
void set_pixel(uint_fast8_t x, uint_fast8_t y);

void fill_line_left(uint_fast8_t x, const uint_fast8_t y)
{
    for(;; x--)
    {
        if(get_pixel(x, y))
            return;
        set_pixel(x, y);
    }
}
```

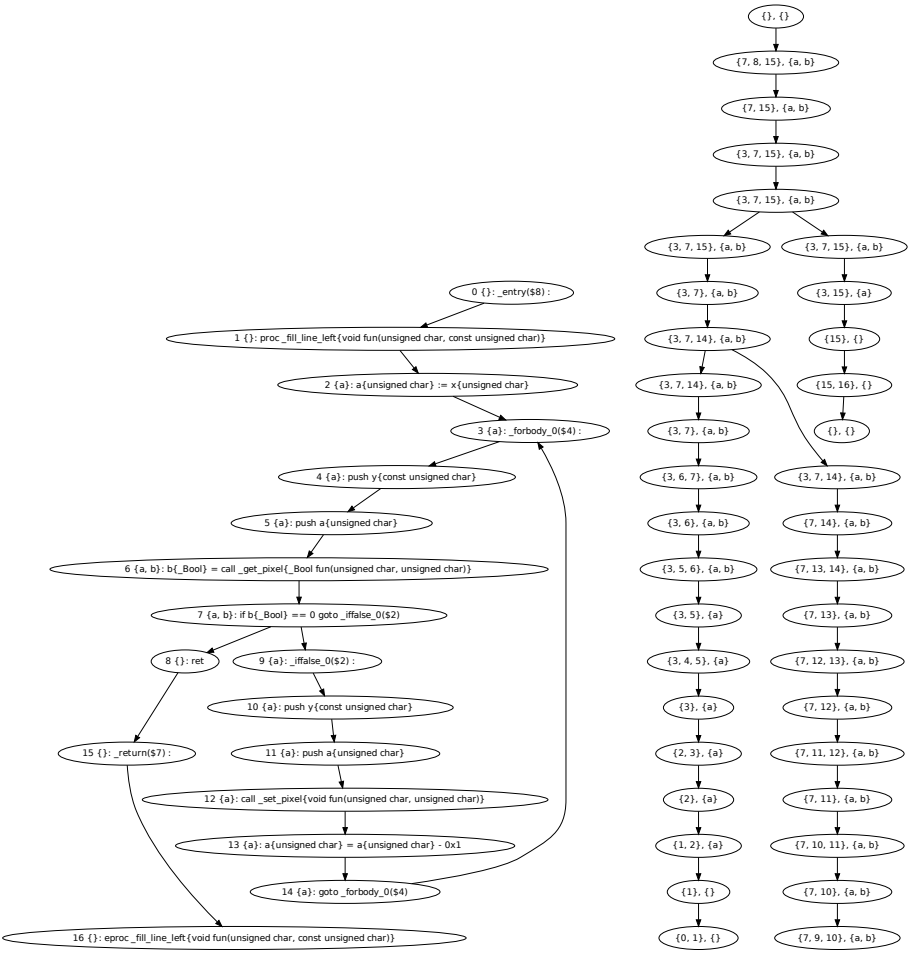
Fig. 1. C code example

Let r be the number of registers. Let $[r] := \{0, \ldots, r-1\}$ be the the set of registers.

Definition 1. *Let V be a set of variables. An* assignment *of variables V to registers $[r]$ is a function $f\colon U \to [r], U \subseteq V$. The assignment is* valid *if it is possible to generate correct code for it, which implies that no conflicting variables are assigned to the same register.*

Variables in $V \setminus U$ are to be placed in memory (spilt) or removed and their value recalculated as needed (rematerialized).

Definition 2 (Register Allocation). *Let the number of available registers be fixed. Given an input program containing variables and their live-ranges and a cost function, that gives costs for register assignments, the problem of* register allocation *is to find an assignment of variables to the registers that minimizes the total cost.*



(a) Control-flow graph (b) Nice tree-decomposition

Fig. 2. CFG and decomposition example

Our approach is based on tree decompositions. Tree decompositions [21] have commonly been used to find polynomial algorithms on restricted graph classes for many problems that are hard on general graphs, since their rediscovery by Robertson and Seymour [29]. This includes well known problems such as graph coloring and vertex cover.

Definition 3 (Tree Decomposition). *Given a graph $G = (\Pi, K)$ a* tree decomposition *of G is a pair $(T, \mathcal{X})$ of a tree T and a family $\mathcal{X} = \{X_i \mid i \text{ node of } T\}$ of subsets of Π with the following properties:*

- $\bigcup_{i \text{ node of } T} X_i = \Pi$,
- *For each edge* $\{x, y\} \in K$, *there is an* i *node of* T, *such that* $x, y \in X_i$,
- *For each node* $x \in \Pi$ *the subgraph of* T *induced by* $\{i \text{ node of } T \mid x \in X_i\}$ *is connected.*

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

Intuitively, tree-width indicates how tree-like a graph is. Nontrivial trees have tree-width 1. Cliques on $n \geq 1$ nodes have tree-width $n - 1$. Series-parallel graphs have tree-width at most 2. Tree-decompositions are usually defined for undirected graphs, and our definition of a tree-decomposition for a directed graph is equivalent to the tree-decomposition of the graph interpreted as undirected.

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

Programs written in Algol or Pascal are 2-structured, Modula-2 programs are 5-structured [32]. Programs written in C are $(7 + \mathfrak{g})$-structured if the number of labels targeted by gotos per function does not exceed $\mathfrak{g}$. Similarly, Java programs are $(6 + \mathfrak{g})$-structured if the number of loops targeted by labeled breaks and labeled continues per function does not exceed $\mathfrak{g}$ [19]. Ada programs are $(6 + \mathfrak{g})$-structured if the number of labels targeted by gotos and labeled loops per function does not exceed $\mathfrak{g}$ [6]. Coding standards tend to place further restrictions, resulting e. g. in C programs being 5-structured when adhering to the widely adopted MISRA-C:2004 [2] standard. A survey of 12522 Java methods from applications and the standard library found tree-width above 3 to be very rare. With one exception of tree-width 5, all methods had tree-width 4 or lower [19].

Often proofs and algorithms on tree-decompositions are easier to describe, understand and implement when using nice tree-decompositions:

Definition 5 (Nice Tree Decomposition). *A tree decomposition* $(T, \mathcal{X})$ *of a graph* G *is called* nice, *iff*

- T *is oriented, with root* t, $X_t = \emptyset$.
- *Each node* i *of* T *is of one of the following types:*
  - *Leaf node, no children*
  - *Introduce node, has one child* $j, X_j \subsetneq X_i$
  - *Forget node, has one child* $j, X_j \supsetneq X_i$
  - *Join node, has two children* $j_1, j_2, X_i = X_{j_1} = X_{j_2}$

Given a tree-decomposition, a nice tree-decomposition of the same width can be found easily. Figure 2(b) shows a nice tree-decomposition of width 2 for the CFG in Figure 2(a). At each node i in the figure, the left set is X_i.

From now on let $G = (\Pi, K)$ be the control flow graph of the program, let $I = (V, E)$ be the corresponding conflict graph of the variables of the program

(i. e. the intersection graph of the variables' live-ranges). The live-ranges are connected subgraphs of G. Let $(T, \mathcal{X})$ be a nice tree decomposition of minimum width of G with root t. For $\pi \in \Pi$ let V_π be the set of all variables $v \in V$, that are alive at π (in the example CFG in figure 2(a) this is the set directly after the node number). Let $\mathcal{V} := \max_{\pi \in \Pi}\{V_\pi\}$ be the maximum number of variables alive at the same node. For each $i \in T$ let $V_i := \bigcup_{\pi \in X_i} V_\pi$ be the set of variables alive at any of the corresponding nodes from the CFG (in the nice tree-decomposition in figure 2(b) this is the right one of the sets at each node).

3 Optimal Polynomial Time Register Allocation

The goal in register allocation is to minimize costs, including spill and rematerialization costs, costs from not respecting register preferences, costs from not coalescing, etc. These costs are modeled by a cost function that gives costs for an instruction π under register assignment f:

$$c\colon \{(\pi, f) \mid f\colon U \to [r], U \subseteq V_\pi, \pi \in \Pi\} \to [0, \infty]$$

Different optimization goals, such as speed or code size can be implemented by choosing c. E. g. when optimizing for code size c could give the code size for π under assignment f, or when optimizing for speed c could give the number of cycles π needs to execute multiplied by an execution probability obtained from a profiler. We assume that c can be evaluated in constant time. The goal is thus finding an f for which $\sum_{\pi \in \Pi} c(\pi, f|V_\pi)$ is minimal.

Let S be the function that gives the minimum possible costs for instructions in the subtree rooted at $i \in T$, excluding instructions in X_i when assigning variables alive in the subtree rooted at $i \in T$ when choosing $f\colon U \to [r], U \subseteq V$ as the assignment of variables alive at instructions $i \subseteq \Pi$ to registers, i. e.

$$S\colon \{(i, f) \mid i \in T, f\colon U \to [r], U \subseteq V_i\} \to [0, \infty].$$
$$S(i, f) := \min_{g|_{V_i} = f|_{V_i}} \left\{ \sum_{\pi \in T_i} c(\pi, g|_{V_\pi}) \right\}.$$

Where T_i is the set of instructions in the subtree of T rooted at $i \in T$, excluding instructions in X_i. This function at the root $t \in T$, and the corresponding assignment that results in the minimum is what we want:

$$S(t, f) = \min_{g|_{V_t} = f|_{V_t}} \left\{ \sum_{\pi \in T_t} c(\pi, g|_{V_\pi}) \right\} =$$
$$= \min_{g|_\emptyset = f|_\emptyset} \left\{ \sum_{\pi \in \Pi} c(\pi, g|_{V_\pi}) \right\} = \min_g \left\{ \sum_{\pi \in \Pi} c(\pi, g|_{V_\pi}) \right\}.$$

To get S, we first define a function s, and then show that $S = s$ and that s can be calculated in polynomial time. We define s inductively, and depending on the type of i:

- Leaf: $s(i, f) := 0$
- Introduce with child j: $s(i, f) := s(j, f|_{V_j})$
- Forget with child j: $s(i, f) := \min\{\sum_{\pi \in X_j \setminus X_i} c(\pi, g|_{V_\pi}) + s(j, g) \mid g|_{V_i} = f\}$
- Join with children j_1 and j_2: $s(i, f) := s(j_1, f) + s(j_2, f)$

By calculating all the $s(i, f)$ and recording which g gave the minimum we can obtain an optimal assignment. We will show that s correctly gives the minimum possible cost and that it can be calculated in polynomial time.

Lemma 1. *For each $i \in T, f\colon U \to [r], U \subseteq V_i$ the value $s(i, f)$ is the minimum possible cost for instructions in the subtree rooted at $i \in T$, excluding instructions in X_i when assigning variables alive in the subtree rooted at $i \in T$ when choosing f as the assignment of variables alive at instructions $i \subseteq \Pi$ to registers, i. e. $s = S$. Using standard bookkeeping techniques we obtain the corresponding assignments for the subtree.*

Proof. By induction we can assume that the lemma is true for all children of i. Let T_i be the set of instructions in the subtree rooted at $i \in T$, excluding instructions in X_i.

Case 1: i is a leaf. There are no instructions in $T_i = X_i \setminus X_i = \emptyset$, thus the cost is zero: $s(i, f) = 0 = S(i, f)$.

Case 2: i is an introduce node with child j. $T_i = T_j$, since $X_i \supseteq X_j$, thus the cost remains the same: $s(i, f) = s(j, f) = S(j, f) = S(i, f)$

Case 3: i is a forget node with child j. $T_i = T_j \cup (X_j \setminus X_i)$, the union is disjoint. Thus we get the correct result by adding the costs for the instructions in $X_j \setminus X_i$:

$$s(i, f) = \min_{g|_{V_i} = f} \left\{ \sum_{\pi \in X_j \setminus X_i} c(\pi, f|_{V_\pi}) + s(j, g) \right\} =$$

$$\min_{g|_{V_i} = f} \left\{ \sum_{\pi \in X_j \setminus X_i} c(\pi, f|_{V_\pi}) + S(j, g) \right\} =$$

$$\min_{g|_{V_i} = f|_{V_i}} \left\{ \sum_{\pi \in X_j \setminus X_i} c(\pi, f|_{V_\pi}) + \sum_{\pi \in T_j} c(\pi, g|_{V_\pi}) \right\} =$$

$$\min_{g|_{V_i} = f|_{V_i}} \left\{ \sum_{\pi \in T_i} c(\pi, g|_{V_\pi}) \right\} = S(i, f).$$

Case 4: i is a join node with children j_1 and j_2. $T_i = T_{j_1} \cup T_{j_2}$, since $X_i = X_{j_1} = X_{j_2}$. The union is disjoint. Thus we get the correct result by adding the costs from both subtrees: $s(i, f) = s(j_1, f) + s(j_2, f) = S(j_1, f) + S(j_2, f) = S(i, f)$.

Lemma 2. *Given the tree-decomposition of minimum width, s can be calculated in polynomial time.*

Proof. Each $V_\pi, \pi \in \Pi$ is the union of two cliques, each of size at most $\mathcal{V}$: The variables alive at the start of the instruction form the clique, and so do the variables alive at the end of the instruction. Thus $V_i, i \in T$ is the union of at most $2(tw(G)+1)$ cliques. From each clique at most r variables can be placed in registers.

At each node i of the tree decomposition time $O(\mathcal{V}^{2(tw(G)+1)r})$ is sufficient:

Case 1: i is a leaf. There are at most $O(\mathcal{V}^{2|X_i|r}) \subseteq (\mathcal{V}^{2(tw(G)+1)r})$ possible f, and for each one we do a constant number of calculations.

Case 2: i is an introduce node with child j. The reasoning from case 1 holds.

Case 3: i is a forget node. There are at most $O(\mathcal{V}^{2|X_i|r})$ possible f. For each one we need to consider at most $O(\mathcal{V}^{2|X_j \setminus X_i|r})$ different g. Thus time $O(\mathcal{V}^{2|X_i|r}) \cdot (\mathcal{V}^{2|X_j \setminus X_i|r}) \subseteq O(\mathcal{V}^{2|X_i|r+2|X_j \setminus X_i|r}) = O(\mathcal{V}^{2|X_j|r}) \subseteq O(\mathcal{V}^{2(tw(G)+1)r})$ is sufficient.

Case 4: i is a join node with children j_1 and j_2. The reasoning from case 1 holds.

The tree decomposition has at most $|T|$ nodes, thus the total time is in $O(|T|\mathcal{V}^{2(tw(G)+1)r}) = O(|T|\mathcal{V}^{\mathfrak{c}})$ for a constant $\mathfrak{c}$ and thus polynomial.

Theorem 1. *The register allocation problem can be solved in polynomial time for structured programs.*

Proof. Given an input program of bounded tree-width we can calculate a tree-decomposition of minimum width in linear time [3]. We can then transform this tree-decomposition into a nice one of the same width. The linear time for these steps implies that $|T|$ is linear in $|G|$. Using this nice tree decomposition s is calculated in polynomial time as above. The total runtime is thus in $O(|G|\mathcal{V}^{2(tw(G)+1)r}) = O(|G|\mathcal{V}^{\mathfrak{c}})$ for a constant $\mathfrak{c}$.

4 Remarks

Remark 1. The runtime bound is reduced by a factor of $\mathcal{V}^{tw(G)r}$, if the intermediate representation is three-address code.

Proof. In that case there is at most one variable alive at the end of an instruction that was not alive at the start of the instruction, so in the proof of Lemma 2, we can replace $O(\mathcal{V}^{2(tw(G)+1)r})$ by $O(\mathcal{V}^{(tw(G)+2)r})$.

Remark 2. Bodlaender's algorithm [3] used in the proof above is not a practical option. However there are other, more practical alternatives, including a linear-time algorithm that is not guaranteed to give decompositions of minimal width, but will do so for many programming languages [32,10].

Remark 3. Implementations of the algorithm can be massively parallel, resulting in linear runtime.

Proof. At each $i \in T$ the individual $s(i, f)$ do not depend on each other. They can be calculated in parallel. By requiring that $|X_j| = |X_i| + 1$ at forget nodes, we can assume that the number of different g to consider is at most $\mathcal{V}^{2r}$, resulting in time $O(r)$ for calculating the minimum over the $s(j, g)$. Thus given enough processing elements the runtime of the algorithm can be reduced to $O(|G|r)$.

Remark 4. Doing live-range splitting as a preprocessing step is cheap.

The runtime bound proved above only depends on $\mathcal{V}$, not $|V|$. Thus splitting of non-connected live-ranges before doing register allocation doesn't affect the bound. When the splitting is done to allow more fine-grained control over spilling, then the additional cost is small (even the extreme case of inserting permutation instructions between any two original nodes in the CFG, and splitting all live-ranges there would only double Π and $\mathcal{V}$).

Remark 5. Non-structured programs can be handled at the cost of either a loss of optimality or an increase in runtime.

Programs of high tree-width are extremely uncommon (none have been found so far, with the exception of artificially constructed examples). Nevertheless they should be handled correctly by compilers. One approach would be to handle these programs like the others. Since $tw(G)$ is no longer constant, the algorithm is no longer guaranteed to have polynomial runtime. Where polynomial runtime is essential, a preprocessing step can be used. This preprocessing stage would spill some variables (or allocate them using one of the existing heuristic approaches). Edges of G, at which no variables are alive, can be removed. Once enough edges have been removed, $tw(G) \leq \mathfrak{k}$ and our approach can be applied to allocate the remaining variables. Another option is the heuristic limit used in our prototype as mentioned in Section 6.

Remark 6. The runtime of the polynomial time algorithm can be reduced by a factor of more than $(2(tw(G)+1)r)!$, if there is no register aliasing and registers are interchangeable within each class. Furthermore r can then be chosen as the maximum number of registers that can be used at the same time instead of the total number of registers, which gives a further runtime reduction in case of register aliasing.

Proof. Instead of using $f\colon U \to [r]$ we can directly use U.

- Leaf: $s(i, U) := 0$
- Introduce with child j: $s(i, U) := s(j, U \cup V_j)$
- Forget with child j: $s(i, U) := \sum_{\pi \in X_j \setminus X_i} c(\pi, U) + \min\{s(j, W) \mid W \cap V_i = U\}$
- Join with children j_1 and j_2: $s(i, U) := s(j_1, U) + s(j_2, U)$

Most of the proofs of the lemmata are still valid. However instead of the number of possible f we now look at the number of possible U, which is at most

$$\binom{\mathcal{V}}{2(tw(G)+1)r}.$$

Remark 7. Using a suitable cost function and $r = 1$ we get a polynomial time algorithm for *maximum independent set* on intersection graphs of connected subgraphs of graphs of bounded tree-width.

Remark 8. The allocator is easy to re-target, since the cost function is the only architecture-specific part.

5 Complexity of Register Allocation

The complexity of register allocation in different variations has been studied for a long time and there are many NP-hardness results.

Publication	Difference to our setting
Register allocation via coloring [8]	$tw(G)$ unbounded
On the Complexity of Register Coalescing [5]	$tw(G)$ unbounded
The complexity of coloring circular arcs and chords [16]	r is part of input
Aliased register allocation for straight line programs is NP-complete [26]	r is part of input
On Local Register Allocation [14]	r is part of input

Given a graph I a program can be written, such that the program has conflict graph I [8]. Since 3-colorability is NP-hard [24], this proves the NP-hardness of register allocation, as a decision problem for $r = 3$. However the result does not hold for structured programs. Coalescing is NP-hard even for programs in SSA-form [5]. Again this result does not hold for structured programs. Register allocation, as a decision problem, is NP-hard, even for series-parallel control flow graphs, i. e. for $tw(G) \leq 2$ and thus for structured programs, when the number of registers is part of the input [16]. Register allocation, as a decision problem, is NP-hard when register aliasing is possible, even for straight-line programs, i. e. $tw(G) = 1$ and thus for structured programs, when the number of registers is part of the input [26]. Minimizing spill costs is NP-hard, even for straight-line programs, i. e. $tw(G) = 1$ and thus for structured programs, when the number of registers is part of the input [14].

It is thus fundamental to our polynomial time optimal approach, which handles register aliasing, register preferences, coalescing and spilling, that the input program is structured and the number of registers is fixed.

The runtime bound of our approach proven above is exponential in the number of registers r. However, even a substantially simplified version of the register allocation problem is W[SAT]- and co-W[SAT]-hard when parametrized by the number of registers even for $tw(G) = 2$ [25]. Thus doing optimal register allocation in time faster than $\mathcal{V}^{O(r)}$ would imply a collapse the parametrized complexity hierarchy. Such a collapse is considered highly unlikely in parametrized complexity theory. This means that not only we cannot get rid of the r in the exponent, but we can't even separate it from the $\mathcal{V}$ either.

6 Prototype Implementation

We have implemented a prototype of the allocator in C++ for the HC08, S08, Z80, Z180, Rabbit 2000/3000, Rabbit 3000A and LR35902 ports of sdcc [12], a C compiler for embedded systems. It is the default register allocator for these architectures as of the sdcc 3.2.0 release in mid-2012 and can be found in the public source code repository of the sdcc project.

S08 is the architecture of the current main line of Freescale microcontrollers, a role previously filled by the HC08 architecture. Both architectures have three 8-bit registers, which are assigned by the allocator. The Z80 architecture is a classic architecture designed by Zilog, which was once common in general-purpose computers. It currently is mostly used in embedded systems. The Z180, Rabbit 2000/3000 and Rabbit 3000A are newer architectures derived from the Z80, which are also mostly used in embedded systems. The differences are in the instruction set, not in the register set. The Z80 architecture is simple enough to be easily understood, yet has many of the typical features of complex CISC architectures. Nine 8-bit registers are assigned by the allocator (A, B, C, D, E, H, L, IYL, IYH). IYL and IYH can only be used together as 16-bit register IY; there are instructions that treat BC, DE or HL as 16-bit registers; many 8-bit instructions can only use A as the left operand, while many 16-bit instructions can only use HL as the left operand. There are some complex instructions, like djnz, a decrement-and-jump-if-not-zero instruction that always uses B as its operand, or ldir, which essentially implements memcpy() with the pointer to the destination in DE, source pointer in HL and number of bytes to copy in BC. All these architectural quirks are captured by the cost function. The LR35902 is the CPU used in the Game Boy video game system. It is inspired by the Z80 architecture, but has a more restricted instruction set and fewer registers. Five 8-bit registers are assigned by the allocator.

The prototype still has some limitations, e. g. current code generation does not allow the A or IY registers to hold parts of a bigger variable in the Z80 port. Code size was used as the cost function, due to its importance in embedded systems and relative ease of implementation (optimal speed or energy optimization would require profiler-guided optimization). We obtain the tree decomposition using Thorup's method [32], and then transform it into a nice tree decomposition. The implementation of the allocator essentially follows Section 3, and is neither very optimized for speed nor parallelized. However a configurable limit on the number of assignments considered at each node of the tree decomposition has been introduced. When this limit is reached, some assignments are discarded heuristically. The heuristic mostly relies on the $s(i, f)$ to discard those assignments that have the highest cost so far first, but takes other aspects into account to increase the chance that compatible assignments will exist at join nodes. When the limit is reached, and the heuristic applied, the assignment is no longer provably optimal. This limit essentially provides a trade-off between runtime and quality of the assignment.

The prototype was compared to the current version of the old sdcc register allocator, which has been improved over years of use in sdcc. The old allocator

is basically an improved linear scan [28,13] algorithm extended to take the architecture into account, e.g. preferring to use registers HL and A, since accesses to them typically are faster than those to other registers and taking coalescing, register aliasing and some other preferences into account. This comparison was done using the Z80 architecture, which has been around for a long time, so there is a large number of programs available for it.

Furthermore we did a comparison between the different architectures, which shows the impact of the number of registers on the performance of the allocator.

7 Experimental Results

Six benchmarks considered representative of typical applications for embedded systems have been used to evaluate the register allocator, by compiling them with sdcc 3.2.1 #8085:

- The Dhrystone benchmark [33], version 2 [34]. An ANSI-C version was used, since sdcc does not support K&R C.
- A set of source files taken from real-world applications and used by the sdcc project to track code size changes over sdcc revisions and to compare sdcc to other compilers.
- The Coremark benchmark [1], version 1.0.
- The FatFS implementation of the FAT filesystem [9], version R0.09.
- Source code from two games for the ColecoVision video game console. All C source code has been included, while assembler source files and C source files that only contain data have been omitted.
- The Contiki operating system [11], version 2.5.

We first discuss the results of compiling the benchmarks for the Z80 architecture. Figure 3 shows the code size with the peephole optimizer (a post code-generation optimization stage not taken into account by the cost function) enabled, Figure 4 with the peephole optimizer disabled. Furthermore, Figure 3 shows the compilation time, and Figure 4 shows the fraction of provably optimally allocated functions (i. e. those functions for which the heuristic never was applied); the former is little affected by enabling the peephole optimizer and the latter not at all.

The dhrystone benchmark is rather small. At 10^8 assignments per node we find a provably optimal assignment for 83.3% of the functions. This also results in a moderate reduction in code size of 6.0% before and 4.9% after the peephole optimizer when compared to the old allocator. The sdcc benchmark, even though small, contains more complex functions; at 10^8 assignments per node we find a provably optimal assignment for 93.9% of the functions. However code size seems to be stable from 6×10^7 onwards. We get a code size reduction of 16.9% before and 17.3% after the peephole optimizer. For Coremark, we find an optimal assignment for 77% of the functions at 10^8 assignments per node. We get a code size reduction of 7.8% before and 6.9% after the peephole optimizer.

FatFS is the benchmark which is the most problematic for our allocator; it contains large functions with complex control flow, some containing nearly a kilobyte of local variables. Even at 4.5×10^7 assignments per node (we did not run compilations at higher values due to lack of time) only 45% of the functions are provably optimally allocated. We get a reduction in code size of 9.8% before and 11.4% after the peephole optimizer. Due to the low fraction of provably optimally allocated functions the code size reduction and compilation time are likely to be much higher for a higher number of assignments per node.

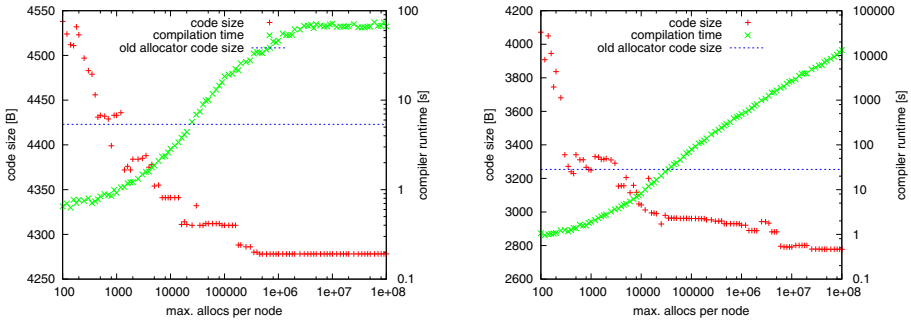
In the games benchmark, about 73% of the functions are provably optimally allocated at 4.5×10^7 assignments per node; at that value the code size is reduced by 11.2% before and 12.3% after the peephole optimizer. This result is consistent with the previous two: The source code contains both complex and simple functions (and some data, since only source files containing data only were excluded, while those that contain both code and data were included).

For Contiki, about 76% of the functions are provably optimally allocated at 4.5×10^6 assignments per node (we did not run compilations at higher values due to lack of time); at that value the code size is reduced by 9.1% before and 8.2% after the peephole optimizer. Contiki contains some complex control flow, but it tends to use global instead of local variables; where there are local variables they are often 32-bit variables, of which neither the optimal nor the old allocator can place more than one in registers at a given time (due to the restriction in code generation that allows the use of IY for 16-bit variables only).

We also did a comparison of the different architectures (except for the Rabbit 3000A, since it is very similar to the Rabbit2000/3000). Figure 5 shows the code size with the peephole optimizer enabled, Figure 6 with the peephole optimizer disabled. Furthermore, Figure 5 shows the compilation time, and Figure 6 shows the fraction of provably optimally allocated functions.

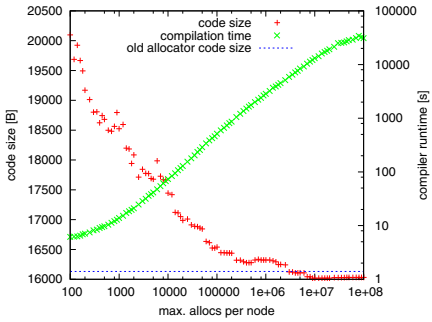
The results clearly show that for the runtime of the register allocator and the fraction of provably optimally allocated function the number of registers is much more important than the architecture: For the architectures with 3 registers, code size is stable from 3.8×10^3 (for HC08) and 4.0×10^3 (for S08) assignments, and all functions are provably optimally allocated from 2.5×10^4 assignments onwards. The effect of the register allocator on compiler runtime is mostly lost in noise. For the architecture with 5 registers (LR35902), code size is stable from 9.0×10^3 assignments onwards, and all functions are provably optimally allocated from 1.4×10^5 assignments onwards. For the architectures with 9 registers, there are still functions for which a provably optimal assignment is not found at 1.0×10^8 assignments. Architectural differences other than the number of registers have a substantial impact on code size, but only a negligible one on the performance of the register allocator.

We also see that the improvement in code size compared to the old allocator was the most substantial for architectures that have just three registers: For the HC08 17.6% before and 18% after the peephole optimizer, for the S08 20.1% before and 21.1% after the peephole optimizer. This has substantially reduced, but not completely eliminated the gap in generated code size between sdcc and

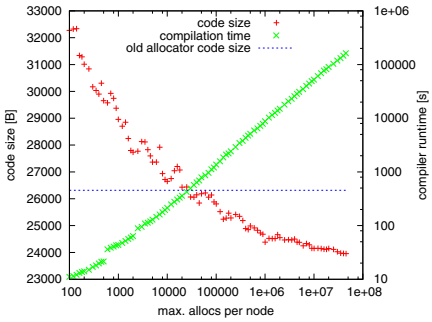


(a) Dhrystone

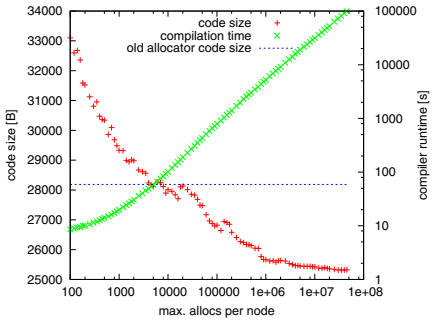
(b) sdcc benchmark



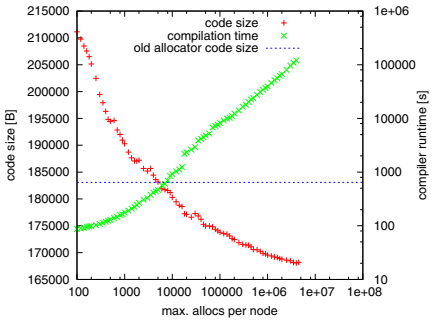
(c) Coremark



(d) FatFS

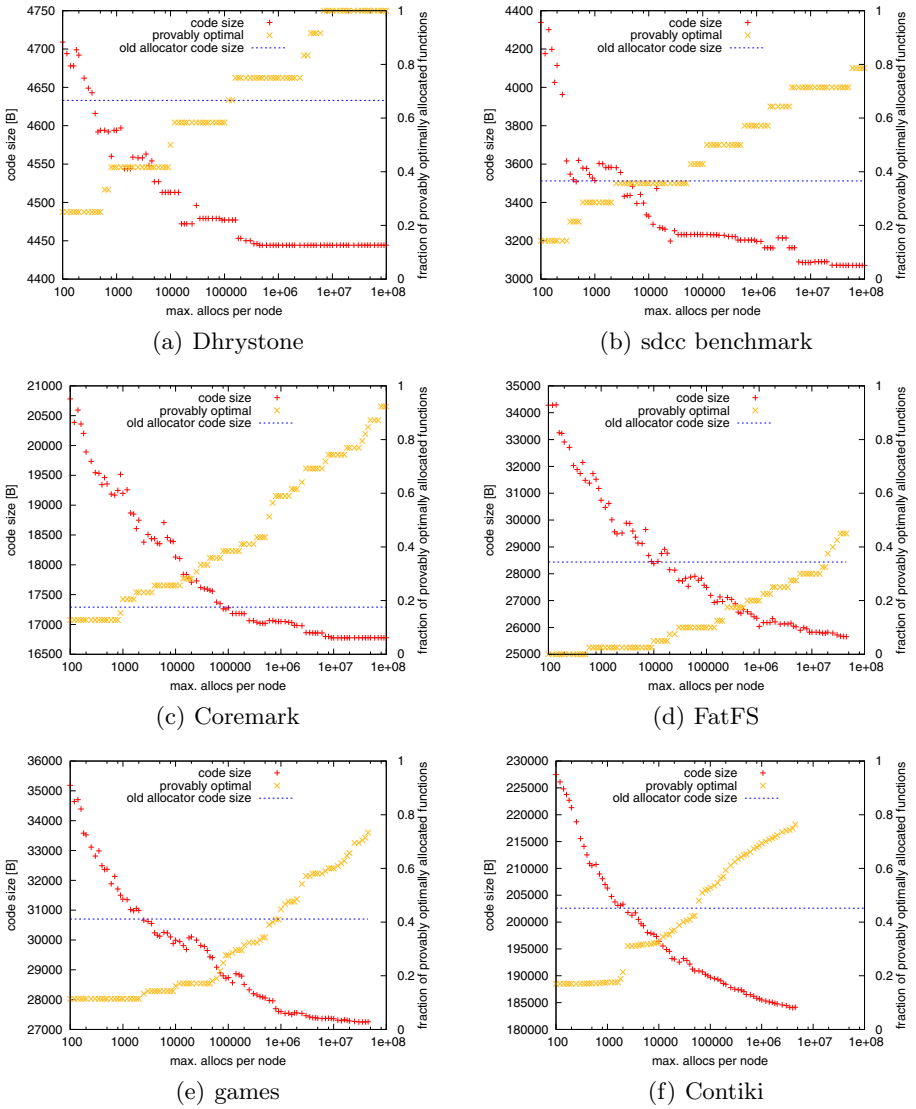


(e) games



(f) Contiki

Fig. 3. Experimental Results (Z80, with peephole optimizer)

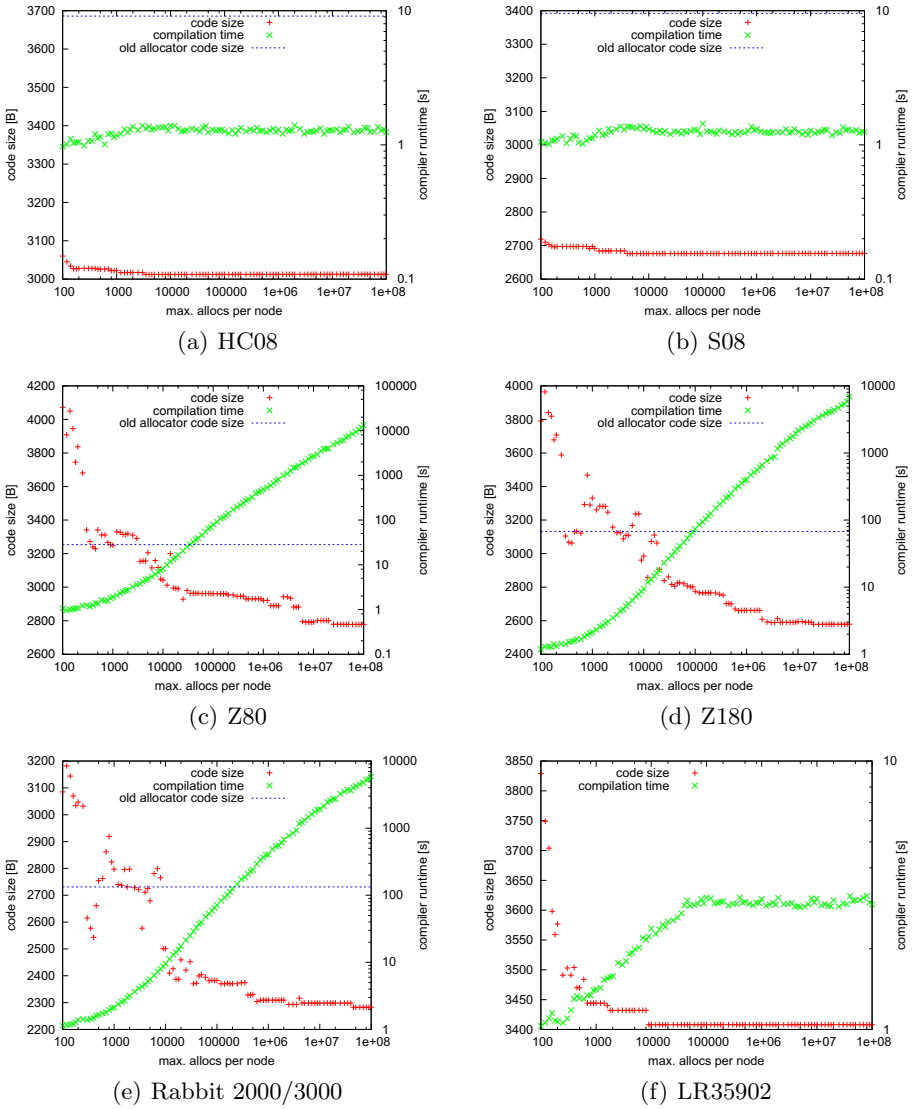


(a) Dhrystone (b) sdcc benchmark

(c) Coremark (d) FatFS

(e) games (f) Contiki

Fig. 4. Experimental Results (Z80, without peephole optimizer)



(a) HC08

(b) S08

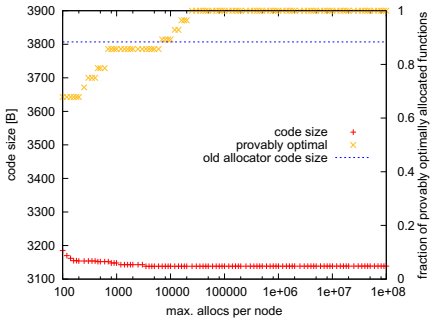
(c) Z80

(d) Z180

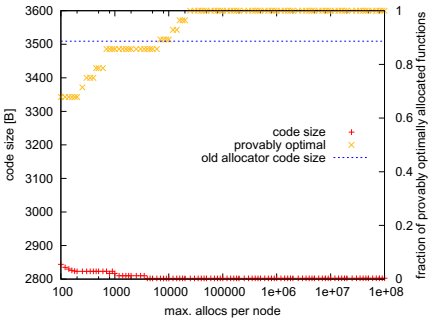
(e) Rabbit 2000/3000

(f) LR35902

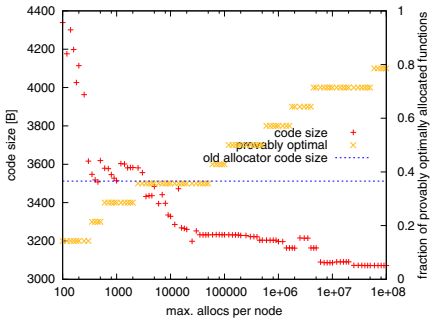
Fig. 5. Experimental Results (sdcc benchmark, with peephole optimizer)



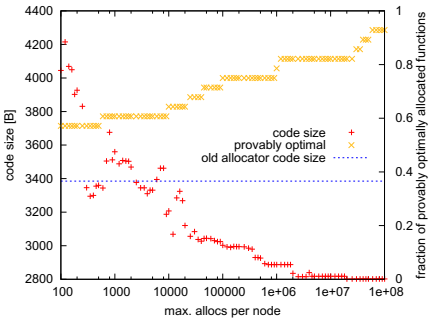
(a) HC08



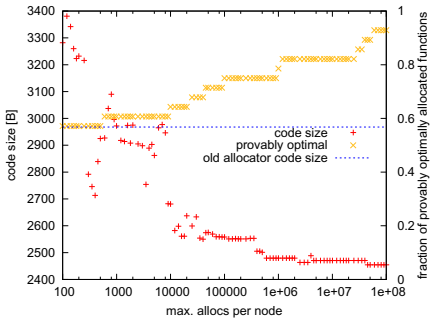
(b) S08



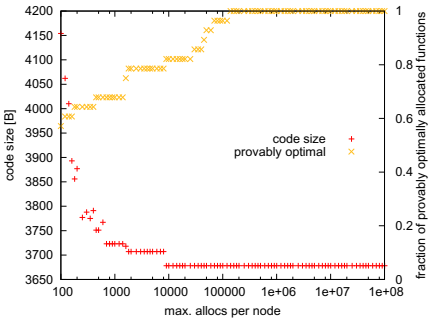
(c) Z80



(d) Z180



(e) Rabbit 2000/3000



(f) LR35902

Fig. 6. Experimental Results (sdcc benchmark, without peephole optimizer)

the competing Code Warrior and Cosmic C compilers. The Z180 and Rabbit 2000/3000 behave similar to the Z80, which was already discussed above. Our register allocator makes sdcc substantially better in generated code size than the competing z88dk, HITECH-C and CROSS-C compilers for these architectures. The LR35902 backend had been unmaintained in sdcc for some time, and was brought back to life after the 3.1.0 release, at which time it was not considered worth the effort to make the old register allocator work with it. There is no other current compiler for the LR35902.

8 Conclusion

We presented an optimal register allocator, that has polynomial runtime. Register allocation is one of the most important stages of a compiler. Thus the allocator is a major step towards improving compilers. The allocator can handle a variety of spill and rematerialization costs, register preferences and coalescing.

A prototype implementation shows the feasibility of the approach, and is already in use in a major cross-compiler targeting architectures found in embedded systems. Experiments show that it performs excellently for architectures with a small number of registers, as common in embedded systems.

Future research could go towards improving the runtime further, completing the prototype and creating a massive parallel implementation. This should make the approach feasible for a broader range of architectures.

References

1. Coremark, `http://www.coremark.org`
2. Guidelines for the Use of the C Language in Critical Systems (MISRA-C:2004, 2nd Edition). Technical report, MISRA (2008)
3. Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM Journal of Computation 25(6), 1305–1317 (1996)
4. Bodlaender, H.L., Gustedt, J., Telle, J.A.: Linear-Time Register Allocation for a Fixed Number of Registers. In: SODA 1998: Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 574–583. Society for Industrial and Applied Mathematics (1998)
5. Bouchez, F., Darte, A., Rastello, F.: On the Complexity of Register Coalescing. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO 2007, pp. 102–114. IEEE Computer Society (2007)
6. Burgstaller, B., Blieberger, J., Scholz, B.: On the Tree Width of Ada Programs. In: Llamosí, A., Strohmeier, A. (eds.) Ada-Europe 2004. LNCS, vol. 3063, pp. 78–90. Springer, Heidelberg (2004)
7. Chaitin, G.J.: Register allocation & spilling via graph coloring. In: SIGPLAN 1982: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, pp. 98–105. Association for Computing Machinery (1982)
8. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via coloring. Computer Languages 6, 47–57 (1981)
9. ChaN. Fatfs, `http://elm-chan.org/fsw/ff/00index_e.html`

10. Dendris, N.D., Kirousis, L.M., Thilikos, D.M.: Fugitive-search games on graphs and related parameters. Theoretical Computer Science 172(1-2), 233–254 (1997)
11. Dunkels, A., Grönvall, B., Voigt, T.: Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In: Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I) (November 2004)
12. Dutta, S.: Anatomy of a Compiler. Circuit Cellar 121, 30–35 (2000)
13. Evlogimenos, A.: Improvements to Linear Scan register allocation, Technical report, University of Illinois, Urbana-Champaign (2004)
14. Farach, M., Liberatore, V.: On local register allocation. In: Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1998, pp. 564–573. Society for Industrial and Applied Mathematics (1998)
15. Fu, C., Wilken, K.: A Faster Optimal Register Allocator. In: MICRO 35: Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture, pp. 245–256. IEEE Computer Society Press (2002)
16. Garey, M.R., Johnson, D.S., Miller, G.L., Papadimitriou, C.H.: The Complexity of Coloring Circular Arcs and Chords. SIAM Journal on Algebraic Discrete Methods 1(2), 216–227 (1980)
17. Goodwin, D.W., Wilken, K.D.: Optimal and near-optimal global register allocations using 0–1 integer programming. Software Practice & Experience 26(8), 929–965 (1996)
18. Guruswami, V., Sinop, A.K.: Improved Inapproximability Results for Maximum k-Colorable Subgraph. In: Dinur, I., Jansen, K., Naor, J., Rolim, J. (eds.) APPROX and RANDOM 2009. LNCS, vol. 5687, pp. 163–176. Springer, Heidelberg (2009)
19. Gustedt, J., Mæhle, O.A., Telle, J.A.: The Treewidth of Java Programs. In: Mount, D.M., Stein, C. (eds.) ALENEX 2002. LNCS, vol. 2409, pp. 86–97. Springer, Heidelberg (2002)
20. Hack, S., Grund, D., Goos, G.: Register Allocation for Programs in SSA-Form. In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 247–262. Springer, Heidelberg (2006)
21. Halin, R.: Zur Klassifikation der endlichen Graphen nach H. Hadwiger und K. Wagner. Mathematische Annalen 172(1), 46–78 (1967)
22. Hames, L., Scholz, B.: Nearly Optimal Register Allocation with PBQP. In: Lightfoot, D.E., Ren, X.-M. (eds.) JMLC 2006. LNCS, vol. 4228, pp. 346–361. Springer, Heidelberg (2006)
23. Kannan, S., Proebsting, T.: Register Allocation in Structured Programs. In: Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1995, pp. 360–368. Society for Industrial and Applied Mathematics (1995)
24. Karp, R.M.: On the Computational Complexity of Combinatorial Problems. Networks 5, 45–68 (1975)
25. Krause, P.K.: The Complexity of Register Allocation. To appear in the GROW 2011 special issue of Discrete Applied Mathematics (2011)
26. Lee, J.K., Palsberg, J., Pereira, F.M.Q.: Aliased register allocation for straight-line programs is NP-complete. Theoretical Computer Science 407(1-3), 258–273 (2008)
27. Pereira, F.M.Q.: Register Allocation Via Coloring of Chordal Graphs. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 315–329. Springer, Heidelberg (2005)
28. Poletto, M., Sarkar, V.: Linear scan register allocation. ACM Transactions on Programming Languages and Systems (TOPLAS) 21(5), 895–913 (1999)
29. Robertson, N., Seymour, P.D.: Graph Minors. I. Excluding a Forest. Journal of Combinatorial Theory, Series B 35(1), 39–61 (1983)
30. Scholz, B., Eckstein, E.: Register Allocation for Irregular Architectures. SIGPLAN Notices 37(7), 139–148 (2002)

31. Smith, M.D., Ramsey, N., Holloway, G.: A Generalized Algorithm for Graph-Coloring Register Allocation. In: PLDI 2004: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, pp. 277–288. Association for Computing Machinery (2004)
32. Thorup, M.: All Structured Programs Have Small Tree Width and Good Register Allocation. Information and Computation 142(2), 159–181 (1998)
33. Weicker, R.P.: Dhrystone: a synthetic systems programming benchmark. Communications of the ACM 27, 1013–1030 (1984)
34. Weicker, R.P.: Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules. SIGPLAN Notices 23, 49–62 (1988)
35. Yannakakis, M., Gavril, F.: The maximum k-colorable subgraph problem for chordal graphs. Information Processing Letters 24(2), 133–137 (1987)