# A Survey on Register Allocation

Fernando Magno Quintão Pereira

October 12, 2008

**Abstract**

Register allocation is the problem of mapping program variables to either machine registers or memory addresses. A good compiler should strive to allocate as many variables in registers as possible, as they provide faster access time; however, registers exist in limited number, as some variables might have to be sent to memory. These are called spilled variables. Register allocation is one of the most important problems in compiler optimization, as a good allocator can improve a naive algorithm in over 250%. It is also one of the most studied problems in compiler theory, and a vast number of different algorithms exist to solve it. In this survey we describe different strategies that compilers use to perform register allocation, and we study the different tradeoffs involved in each algorithm. The first half of the survey touches register allocation from a general perspective, whereas the second half gives special attention to the recent breakthroughs in the field of SSA-based register allocation.

## 1 Introduction

Computer architectures rely on a memory hierarchy to store the data that is manipulated by programs. Figure 1 shows the storage pyramid that is typically observed on a ordinary computer. At the very top of the pyramid we have registers, which provide to the CPU the fastest access to data. Operations of reading and writing to registers in a modern computer take no more than one cycle of the CPU clock. All this velocity comes with a cost: the register file is very small. For instance, the 32-bit x86 chip contains only eight general purpose registers, the 16-bit x86 chip contains 16 and the ARM and the PowerPC chips contain only 32 integer registers. In comparison, its is fairly common to find 200G hard disks in current computers - a difference of almost 33 orders of magnitude!

Because registers are so fast and so few, one of the greatest challenges of compiler writers is to design algorithms that keep the most used program variables in registers, while relegating the least used variables to memory if necessary. Register allocation is thus the problem of mapping variables to registers or memory. This is one of the most important compiler optimizations. As an example, experiments show that an optimal allocator can produce code that is over 250% faster than the code produced by a naive algorithm that maps all the variables to memory. We will be using the program in Figure 2 to illustrate the main concepts related to register allocation.

A program can be described by its *control flow graph*. The control flow graph is formed by *basic blocks*. Each basic block has a unique label, and a list of *instructions*. Instructions are the primary constituents of programs. Each instruction may apply an operation on some variables, which are called the *used variables*. An instruction may define one or more variables; these are called *defined variables*. Different computer architectures, e.g x86, PowerPC, ARM, etc, provide different sets of instructions, but all these sets are Turing Complete. The program in Figure 2 contains four basic blocks and 14 instructions. For simplicity we will be dealing with only four types of instructions: assignments, branches, jumps and joins. The first instruction of basic block $L_2$ is an assignment that uses the variable $a$ and defines the variable $c$. We use assignments to abstract instructions that use or define variables. The actual operation that the instruction applies on its operands is not important for our purposes, and if the instruction has no operand on either the left or right side, we will represent this with a $\bullet$ symbol. The other kinds of instructions model the shape of the control flow graph. Branches finish basic blocks with multiple successors. Jumps finish basic blocks

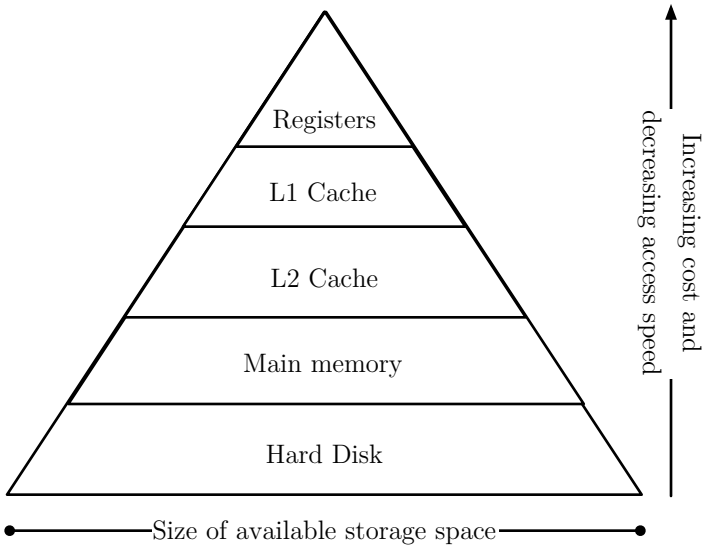



Figure 1: Memory hierarchy in a typical computer architecture.

with a single successor and joins start basic blocks with multiple predecessors. The only operands of these instructions are basic block labels.

We call a *program point* the point between two consecutive instructions. The program in Figure 2 contains 16 program points named $p_1$ to $p_{16}$. We say that a variable $v$ is *alive* at a program point $p$ if there is a path from $p$ to an instruction that uses $v$ where $v$ is not re-defined by any instruction. The collection of program points where a variable is alive is called its *live range*. For instance, the live range of variable $B$ is $\{p_2, p_3, p_4, p_9\}$, whereas the live range of variable $a$ includes all the program points but $p_{11}, p_{12}$ and $p_{16}$. Notice that $a$ is not alive at program points $p_{11}$ and $p_{12}$ because this variable is redefined by the instruction $a = f$, and its old value is not necessary after the instruction $f = a$. A simple algorithm to compute liveness information is given by Appel and Palsberg [3, p.206].

We say that two variables *interfere* if the intersection between their live ranges is non-empty. In this case, we also say that their live ranges overlap. For instance, variables $c$ and $d$ interfere, because their live ranges overlap at program point $p_5$; however, variables $c$ and $E$ do not intefere. This concept is very important for register allocation, because two variables that do not interfere can be stored in the same register.

## 1.1 Irregular Architectures

Modern computer architectures present *irregular* register banks. The two most common sources of irregularities are *pre-colored* registers and *aliasing* [42, 60, 61].

## 1.2 Pre-coloring

Pre-coloring is a very common phenomenon that forces some variables to be assigned to particular machine registers. A typical example is parameter passing in function calls. Architectures such as PowerPC and StrongARM use registers to pass arguments to functions. For instance, a two argument function call in PowerPC is written in assembly in a way similar to the code strip below:

```
r0 = arg0 ; the first argument must be passed in r0
r1 = arg1 ; the second argument must be passed in r1
```

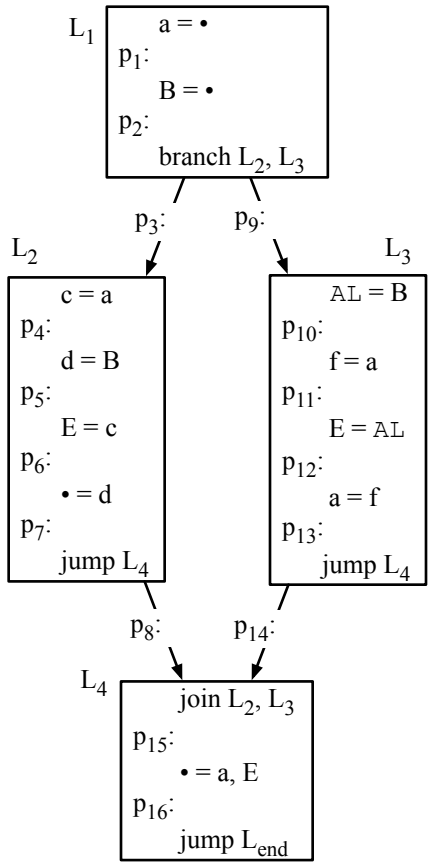



Figure 2: An example program.

`bl foo ; branch and link, e.g, calls the function foo`
The variables `arg0` and `arg1` can be stored in any register, but the variables `r0` and `r1` cannot: they are already allocated to registers `r0` and `r1`. Many other common examples of pre-coloring are found in x86. For instance, in that architecture, the results of a division operation must be placed in the registers `edx` and `eax`. As a convention, we name a pre-colored variable with the name of its pre-coloring register. For instance, the variable `AL` in block $L_3$ of Figure 2 is pre-colored with register `AL`.

## 1.3 Aliasing

We say that an architecture contains aliasing when an assignment to one register name can affect the value of another [61]. For example, Figure 3 shows the set of general purpose registers used in the x86 architecture. The x86 architecture has four general purpose 16-bit registers that can also be used as eight 8-bit registers. Each 8-bit register is called a *low* address or a *high* address. The initial bits of a 16-bit register must be *aligned* with the initial bits of a *low*-address 8-bit register. The x86 architecture allows the combination of two 8-bit registers into one 16 bit register. Another example of aliased registers is the combination of two aligned single precision floating-point registers to form one double-precision register. Examples of architectures with such aliased registers include early versions of HP PA-RISC, the Sun SPARC, and the ARM processors. For a different kind of architecture, Scholz and Eckstein [60] describe experiments with the Carmel 20xxDSP

Core, which has six 40 bit accumulators that can also be used as six 32-bit registers or as twelve 16-bit aligned registers. As a convention, along this dissertation we will use lower case names to denote values that fit in one single register, and upper case names to denote values that must fit in one register pair. Thus, in Figure 2 variables $a$, $c$ and $d$ fit in one register, whereas variables $B$ and $E$ fit in a register pair.

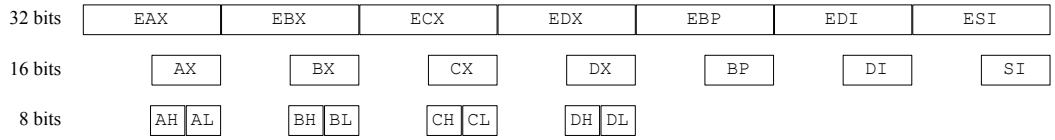



Figure 3: General purpose registers from the x86 architecture. This Figure was taken from [55].

### 1.4 Some Register Allocatin Jargon

**Spilling** Because registers exist in limited number, they may not be enough to store all the variables in the source program. If that is the case, then some variables must be mapped to memory. The act of storing a variable into memory is called *spilling*. Spill is normally undesirable because it forces the register allocator to insert special instructions, that we will call spill code, in the target program to access values stored in memory. An instruction that copies a value from a register to a memory address is called a store. The opposite instruction, which copies a value from memory into a register, is called a load. These instructions tend to be slow compared to operations that do not access memory; thus, one of the objectives of a register allocator is to avoid inserting such instructions in the code that it produces.

**Coalescing** If two variables $v_1$ and $v_2$ do not interfere, and they are related by a copy instruction, that is, the source program contains an instruction such as $v_1 = v_2$, then it is desirable that these variables be allocated into the same register $r$. In this case, we will have the copy instruction $r = r$, which is redundant and can safely be removed from the target program. *Coalescing* is the act of mapping two non-interfering variables that are related by a copy instruction to the same register. For instance, in the program in Figure 2, the instructions $f = a$ and $a = f$ can be eliminated from the program if variables $a$ and $f$ are assigned to the same register. Therefore, a good register allocator should not only assign different registers to interfering variables, but also try to assign the same register to variables related by copies.

**Live Range Splitting** This concept is the inverse of coalescing. Whereas coalescing join the live ranges of variables by removing copies from the source program, live range splitting divides the live range of variables by adding copies to the program and renaming variables. The splitting of live ranges tends to reduce the interferences between variables; thus, it might minimize the number of registers required by programs. Figure 4 shows an example of live range splitting.

## 2 Different Register Allocation Approaches

Register allocation is possibly the compilation problem with the greatest number of different algorithms already described in the literature. In the remainder of this section we will be describing several approaches to register allocation, using the program in Figure 2 as a running example.

### 2.1 Register Allocation via Graph Coloring

Graph coloring is the most used approach to solve register allocation. The *Interference Graph* of a program is the intersection graph of the live ranges of the variables in the program. That is, given a program $P$, its interference graph $G = (V, E)$ contains a vertex for each variable $v$ of $P$. An edge $(u, v)$ is in $E$ if the

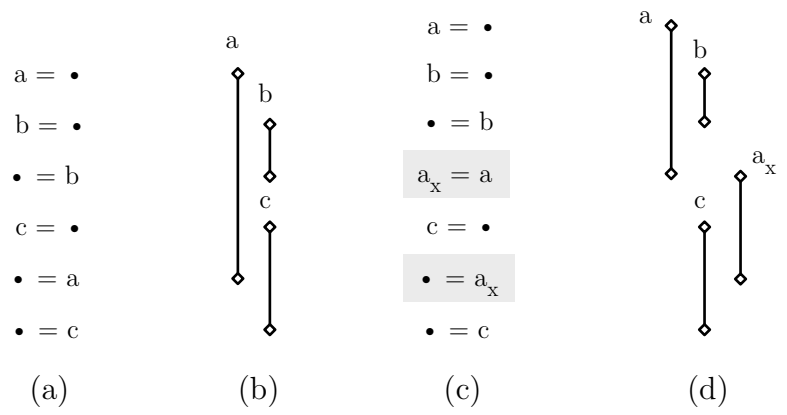



Figure 4: (a) Example program. (b) Live ranges represented as intervals. (c) Program after live range of variable $a$ is split. (d) New interval representation.

intersection of the live ranges of variables $v$ and $u$ is non-empty. Figure 5 shows the interference graph for the Program in Figure 2, as well as a valid register assignment using three x86 registers: `AX`, `BX` and `CX`.

The problem of assigning registers to variables can thus be approximated by coloring the interference graph of the source program. Each color corresponds to a register, and interfering variables will be given different colors, given that they are adjacent on the interference graph. One of the first and most celebrated graph coloring based register allocators was described by Chaitin *et al.* [15, 16]. The algorithm described in [15] laid the foundations of practically all the graph coloring based register allocators that were introduced later. The core of Chaitin's algorithm is Kempe's coloring scheme [39]. Basically, a node $v$ in the interference graph $G$ can be colored if it has less than $K$ neighbors, where $K$ is the number of colors available. In this case, the node $v$ can be safely removed from $G$, and placed on a stack of nodes that are guaranteed to be colorable. This process, called simplification, iterates until there is no more nodes to remove from $G$.

Two aspects of the register allocation problem complicate this technique: *spilling* and *coalescing*. Spilling is the act of mapping a variable to memory because there is no more registers available to hold its value. Coalescing is the act of mapping two non-interfering variables that are related by a copy instruction to the same register. For instance, in the program in Figure 2, the instructions $f = a$ and $a = f$ can be eliminated from the program if variables $a$ and $f$ are assigned to the same register. Therefore, a good graph coloring based register allocator should not only assign different colors to interfering variables, but also try to assign the same color to variables related by copies. Due to spilling and coalescing, Chaitin *et al.* proposed an iterative algorithm, illustrated in Figure 6. This algorithm has the following phases:

1. **Renumber**: discover live range information in the source program.
2. **Build**: build the interference graph.
3. **Coalesce**: merge the live ranges of non-interfering variables related by copy instructions.
4. **Spill cost**: compute the spill cost of each variable. That is a measure of the impact of mapping a variable to memory on the speed of the final program.
5. **Simplify**: Kempe's coloring method.
6. **Spill Code**: insert spill instructions, i.e loads and stores to commute values between registers and memory.

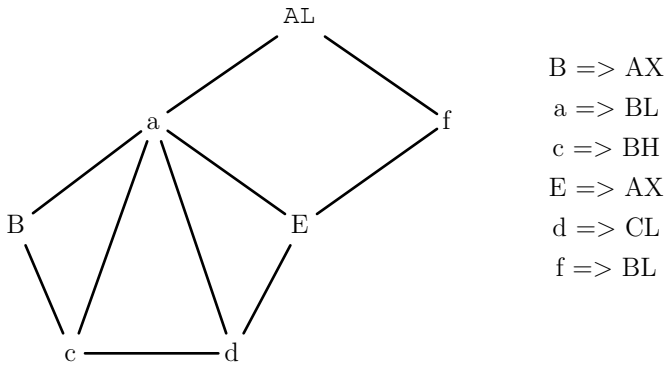



Figure 5: Interference graph for the Program in Figure 2.

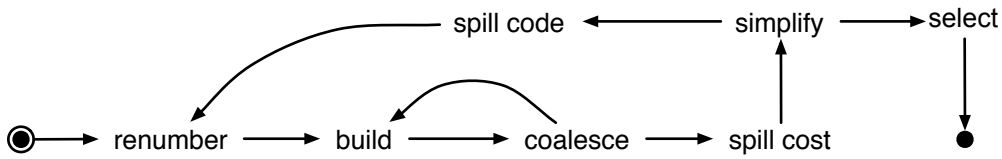



Figure 6: Chaitin *et al.*'s iterative graph coloring based register allocator. This Figure was taken from [11].

7. **Select**: assign a register to each variable.

One of the early achievements of Chaitin *et al.* [16] was to show that spill free register allocation is a NP-complete problem. Basically, Chaitin *et al.* proved that any graph is the interference graph of some program. For instance, to represent $C_4$, the cycle with four nodes, the NP-completeness proof would produce the program in Figure 7. The minimal coloring of such graph can be trivially mapped to a minimal coloring of $C_4$, by simply deleting node $x$. The NP-completeness result comes from the fact that finding a minimal coloring to a graph is NP-complete, as shown by Richard Karp in its seminal work [38].

Chaitin's algorithm had a few deficiencies that were improved by later works. One of the problems of that allocator was the aggressive coalescing policy. Merging live ranges of variables has the undesirable effect of increasing the degree of vertices in the interference graph, and thus it might cause spilling. In order to solve this omission, Briggs *et al.* [11] introduced the concept of *conservative coalescing*. This is an extra criterion to decide when two live ranges can be merged. Thus, in addition of the non-interfering requirement, two variables can only be coalesced if their merging will not cause further spilling in the interference graph. Another improvement brought up by Briggs *et al.* was *biased coloring*: the select phase tries to assign the same color to variables that are copy related. Briggs *et al.* point that the combination of conservative coalescing and biased coloring could remove most of the copy instructions in the original program before register allocation. Finally, Briggs *et al.* introduced the concept of *optimistic coloring*: instead of spilling away variables that could not be simplified using Kempe's technique, Briggs *et al.* defer this decision to the simplification phase. Many times two or more of the neighbors of a vertex $v$ will be assigned the same color, and $v$ will be colorable even if it has a number of neighbors that is larger than the number of available colors. The modified version of Chaitin's algorithm, as described by Briggs *et al.* [11], is illustrated in Figure 8.

A criticism of Briggs allocator is that it is too conservative with regard to the coalescing policy, and thus it misses some copy instructions that could be removed. The coalescing criterion used by Briggs *et al.* [11]

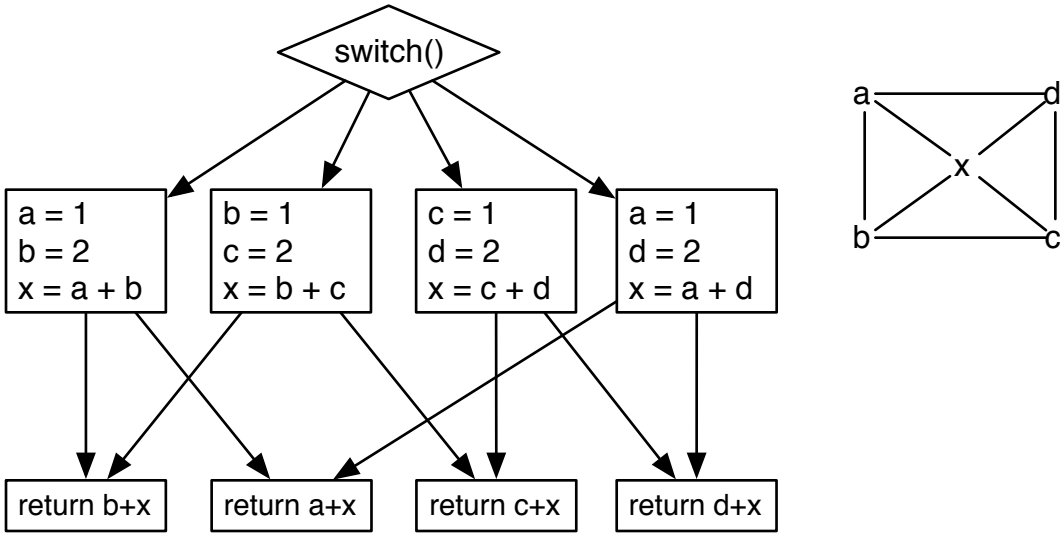



Figure 7: Chaitin *et al.*'s program to represent $C_4$. This example was taken from [54].

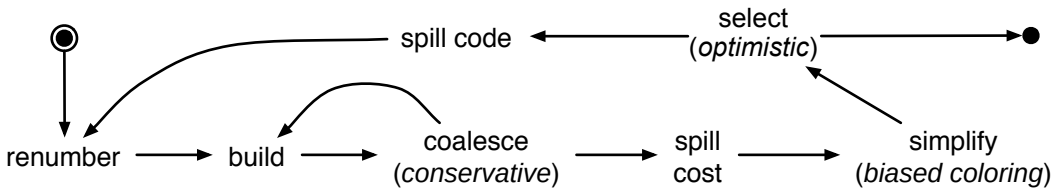



Figure 8: Briggs *et al.* graph coloring based register allocator.

is described as follows: nodes $a$ and $b$ can be coalesced if the node that results from their merging has less than $K$ neighbors of *significant* degree, where a node has significant degree if it has $K$ or more neighbors. Subsequently, George and Appel [30] showed that this criterion could be relaxed to allow more aggressive coalescing without introducing extra spilling. They proposed Iterated Register Coalescing, a graph coloring based register allocator [30] that remains today, almost 15 years after its first release, the base algorithm taught in many compiler courses [3] and used as baseline in many research projects. Figure 9 illustrates the several phases of this algorithm.

An important addition of Iterated Register Coalescing over the previous allocators was a *Freeze* phase. If neither simplify nor coalesce could remove any node from the interference graph, the freeze step would mark a copy related node, so that it would no longer be considered for coalescing.

### 2.1.1 A Taxonomy of Coalescing Approaches

One quarter century after Chaitin's seminal paper, coalescing has been one of the main forces pushing new variations in graph coloring register allocation. Bouchez *et al.* [9] summarizes some of the best known approaches for performing register coalescing:

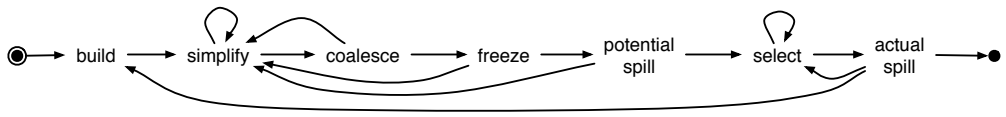



Figure 9: Iterated Register Coalescing. This Figure was taken from [3].

- *Aggressive Coalescing* [16, 15]: merge move-related vertices, regardless of the colorability of the interference graph after the merging.
- *Conservative Coalescing* [11]: merge moves if, and only if, it does not compromise the colorability of the interference graph.
- *Optimistic Coalescing* [51, 52]: coalesce moves aggressively, and if it compromises the colorability of the graph, then give up as few moves as possible.
- *Incremental Conservative Coalescing* [30]: remove one particular move instruction, while keeping the colorability of the graph.

Bouchez *et al.* [7, 9] have shown, by means of an ingenious sequence of reductions, that all these different realizations of the register coalescing problem are NP-complete for general interference graphs.

## 2.2 Linear Scan Register Allocation

Graph coloring is the most popular approach for register allocation, but it is not the only one, and it has been losing ground to a different, simpler approach called *Linear Scan*. Variations of this technique are adopted in many modern compilers, including LLVM [24] and the Java HotSpot client compiler [65]. In this section we describe Linear Scan and some of its more important variations.

Linear scan is a greedy algorithm introduced by Poletto and Sarkar on the late nineties [56]. It simplifies register allocation by reducing it to the problem of assigning colors to an ordered sequence of intervals. It is well know that ordered intervals can be colored optimally by a simple greedy algorithm [28]. However, linear scan is not optimal: it uses the optimal algorithm as an approximation to solve the register allocation problem. The algorithm starts by *linearizing* the basic blocks of the source program, that is, arranging these blocks in a linear sequence; the exact ordering chosen is not important and will not compromise the correctness of the results. Given this linearization, linear scan replaces the live range of each variable with a contiguous interval, called the variable *lifeline*, and then proceeds to color these intervals. Figure 10 illustrates this algorithm being applied on the program in Figure 2. In this case, register allocation amounts to coloring the seven intervals defined by variables $a, B, c, d, E, f$ and `AL`.

The main appeal of linear scan is the allocation speed. This fact makes linear scan an attractive option to Just in Time (JIT) compilers, like Java HotSpot and LLVM. Timing comparisons between graph coloring and linear scan span a wide spectrum [55]. The original linear scan paper [56] suggests that graph coloring is about twice as slow as linear scan. These numbers are corroborated by Sagonas and Stenman [58]. Traub *et al.* [64] gives an slowdown of up to 3.5x for large programs, and Sarkar and Barik [59] suggest a 20x slowdown. On the other hand, this speed pays a price in terms of quality of the code produced. Going back to our example in Figure 10, the original linear scan algorithm [56] would not be able to allocate variables $B$ and $E$ into the same register, even though these variables do not interfere, because their lifelines overlap. This omission is due to the original algorithm not handling *holes* in the live ranges of variables.

Later improvements on linear scan are able to handle holes in the live ranges of variables. An important extension is due to Traub *et al.* [64]. Traub's algorithm introduces the *binpacking* model: the machine registers are viewed as bins into which variable lifetimes can be packed. Thus, linear scan can assign two non-overlapping lifetimes to the same bin, or it can assign two lifetimes into the same bin if the live range

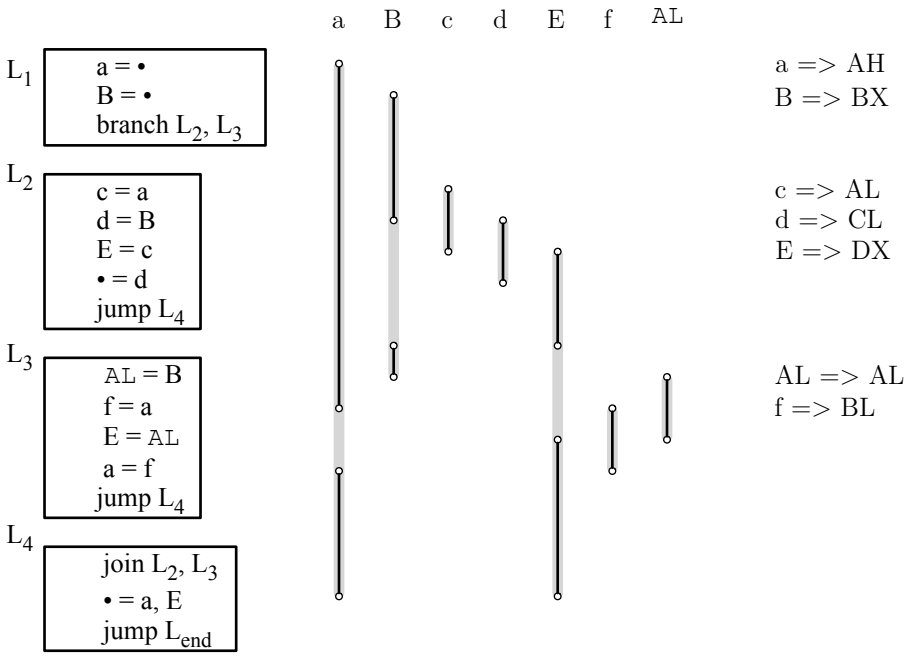



Figure 10: Linear Scan register allocation.

that constitutes one of the lifetimes is completely contained in holes in the live range that constitutes the other lifetime. This model was later used by Mössenböck and Pfeiffer [47] in an algorithm that performs register allocation in programs in Static Single Assignment form.

Wimmer and Mössenböck have introduced several optimizations to linear scan [65]. Their algorithm handles holes in live ranges, but their most innovative extensions are optimal split positions, register hints and spill store elimination. Optimal split position is a technique to minimize the effects of spill code in the final program produced after register allocation. This optimization allows to move loads outside loops, for instance. Register hint is a simplified coalescing approach for linear scan. It is similar to biased coloring [11] in the sense that, when choosing a color to an interval $i_1$, if $i_1$ is connected to another interval $i_2$ via a move instruction, then the allocator attempts to assign to $i_1$ the color previously assigned to $i_2$. Spill store elimination is an optimization used to remove from the target program some store instructions that can be proved statically to be useless. A common example is multiple stores of a variable that is only defined once. In this case, all the store instructions can be replaced with a single store after the definition point of the variable.

The most recent addition to the family of linear scan algorithms is Sarkar and Barik [59] new method, called *Extended Linear Scan*. By inserting copy and swap instructions along the source program, this version of linear scan guarantees to use the minimal number of registers necessary to compile the source program. Sarkar's algorithm diverges from previous implementations because it has a preference towards inserting extra move instructions to avoid inserting spill code.

## 2.3 Register allocation via Integer Linear Programming

Quoting Dasgupta *et al.* [18], linear programming, together with dynamic programming, are the two sledgehammers of the algorithm craft. The linear programming framework fits a vast number of different optimization problems, and a subclass of this model, the 0-1 integer programming, has been used to solve register

allocation. The basic idea of this approach is to model the interactions between registers and variables as constraints in a system of integer linear equations. For instance, given a register $AX$ we define the following variables:

- $AX_r(v, p)$ is one if variable $v$ reaches and leaves program point $p$ allocated to register $AX$ and is zero otherwise.
- $AX_l(v, p)$ is one if variable $v$ reaches program point $p$ in memory, but leaves it allocated to register $AX$. It is zero otherwise. The letter $l$ indicates a load.
- $AX_s(v, p)$ is one if variable $v$ reaches program point $p$ allocated to register $AX$, but leaves it in memory. It is zero otherwise. The letter $s$ indicates a store.

The constraints must guarantee that only one variable will be allocated into register $AX$ at any time; thus, we add to the linear system the constraint:

$$\forall(v, p), \sum(AX_r(v, p) + AX_s(v, p) + AX_l(v, p)) \leq 1$$

Goodwin and Wilken [32] gave the first formulation of register allocation as a 0-1-integer programming problem. Their constraint set could handle several facets of register allocation, such as coalescing, spilling, rematerialization [10] and aliasing. Goodwin's register allocator produces code of very good quality; however, as integer linear programming is NP-complete [38], it presents a worst case exponential running time, and can take hours to find an optimal solution.

Two years after Goodwin's work, Kong and Wilken described a constraint set broad enough to encompass all the irregularities of the x86 architecture [42]. Posteriorly, Appel and George [2] introduced a different approach for IP-based register allocation: the separation of phases between spilling and register assignment. The constraint solver is responsible for defining which variables stay in registers and which variables are spilled. The non-spilled variables are mapped to registers in the next phase. This approach is faster than the previous IP-based algorithms; however, it may produce an undesirably large number of move instructions transferring values between registers. Appel and George use a variation of the optimistic coalescing of Park and Moon [52], which is not optimal, to remove redundant copies.

Finally, Fu and Wilken [27] presented a new IP formulation that keeps the optimal guarantees of the initial algorithm, e.g [32], but is 150 times faster. It is important to point that this "fast" algorithm is still much slower than traditional allocators such as linear scan, and even graph coloring. Therefore, IP-based register allocation has not yet seen use in industrial strength compilers. Nevertheless, it has been successfully used in research in embedded systems [48, 50], and in bit-width aware register allocation [4].

## 2.4 Register allocation via Partitioned Quadratic Programming

The Partitioned Boolean Quadratic Problem (PBQP) is a kind of Quadratic Assignment Problem (QAP) [14]. PBQP is NP-complete; however, a subclass of these problems can be solved in polynomial time. Quoting Hames and Scholtz [36], the input for PBQP is a set of discrete variables $X = \{x_1, \ldots, x_n\}$ and their finite domains $\{D_1, \ldots, D_n\}$, where $m_i = |D_i|$. A solution of PBQP is a function $h : X \rightarrow D$, where $D = D_1 \cup D_2 \cup \ldots \cup D_n$. For each variable $x_i$, $h$ finds an element $d_i \in D_i$. The challenge of PBQP is to find a function $h$ of minimal cost, where the cost $c$ is controlled by two sets of terms:

- the cost of assigning variable $x_i$ to $d_i$. This cost is measured by a *local* cost function $l(x_i, d_i)$;
- the cost given by the interactions between two variables. This is the cost of assigning variable $x_i$ to $d_i$ and assigning variable $x_j$ to $d_j$. This cost is measured by a *related* cost function $r(x_i, x_j, d_i, d_j)$.

Thus, the total cost of an assignment is:

$$c = \sum_{1 \leq i \leq n} l(x_i, h(x_i)) + \sum_{1 \leq i < j \leq n} r(x_i, x_j, h(x_i), h_(x_j))$$

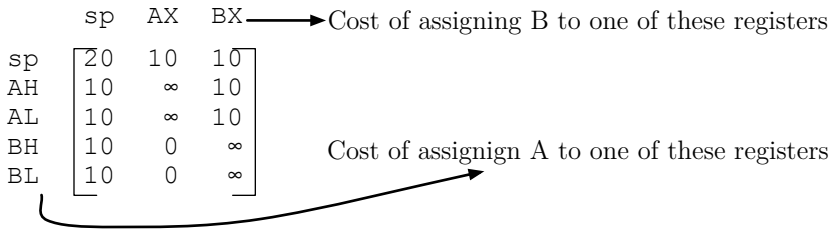



Figure 11: Some example cost matrices for the program in Figure 2.

Although PBQP is NP-complete in general, there is a class of assignments that can be solved in polynomial time. The algorithm introduced in [60] is able to identify these problems, and solve them in $O(nm^3)$, where $n$ is the number of variables and $m$ is the maximum size of any domain.

PBQP has seen use in two different compiler related problems: instruction selection [20, 22, 21] and register allocation [36, 37, 60]. For register allocation, the solver introduced by Scholz *et al.* [60] associates a *cost matrix* $C$ to each edge of the interference graph of the source program. Each cost matrix $C_{uv}$ describes the tradeoffs of assigning different registers to variables $u$ and $v$. As an example, lets build the cost matrix for variables $a$ and $B$ in the program of Figure 2, assuming a bank of registers with two registers only, $AX$ and $BX$. We assume that each of these two registers have two disjoint aliases, in the same configuration found in x86, that is, $AX$ alias $AH$ and $AL$, and $BX$ alias $BH$ and $BL$. Figure 11 shows the cost matrix.

The complexity of PBQP for register allocation is $O(|V|K^3)$, where $|V|$ is the number of variables in the source program, and $K$ is the number of registers in the target architecture. Experiments performed by Hames *et al.* [36] showed that their implementation of a PBQP solver could find the optimal register allocation in 97.4% of the functions in SPEC CPU 2000. For the cases that no solution could be proved to be optimal, a branch-and-bound heuristics was used to find a satisfactory register mapping.

## 2.5 Register allocation via Multi-Flow of Commodities

An interesting way to see register allocation is as a *Multi-Flow of Commodities* (MFC) problem. This idea was introduced by Koes and Goldstein [40] in order to perform local register allocation. Local allocation is the version of register allocation that is restricted to basic blocks only, in contrast to global register allocation, that is concerned about the whole program. The same authors later extended their previous work to incorporate global allocation into the initial model [41]. Multi-flow of commodities has a close relation with register allocation. For instance, it was the starting point of Lee *et al.*'s proof that aliasing register allocation is NP-complete [44]. MCF is a NP-complete problem, as shown by Even *et al.* [23], but it is possible to find solutions that, although non-optimal, are satisfactory enough, via heuristics. Koes and Goldstein have refined the heuristic approach with a progressive algorithm: their allocator uses a simple heuristics to find an initial allocation, and, if it is given extra time, it can improve this solution until reaching the optimal.

In the MCF approach, a program is seem as a collection of $K$ pipes, thru which the allocator must pass a number of indivisible commodities. Each pipe corresponds to a physical location, either register or memory, and each commodity corresponds to a variable. Thus, a flow of a commodity represents the detailed allocation of the variable that the commodity encodes. The multi-commodite network flow naturally models several aspected of register allocation, including live-range splitting, rematerialization [10] and pre-colored registers, although it is unclear how this technique would model register banks with aliasing. Koes and Goldstein have optimized their progressive allocator to reduce the size of the target programs, and they have shown that this method is consistently able to produce code of smaller size than a graph coloring based allocator [41].

# 3 SSA based register allocation

An important breakthrough in register allocation happened in 2005, when three different research groups [7, 13, 35] proved independently that the interference graphs of programs in Static Single Assignment (SSA) form are chordal. This result is important because chordal graphs can be colored in polynomial time [28]. In this section we describe the main developments in the area of SSA-based register allocation.

**SSA form.** The *Static Single Assignment* (SSA) form is an intermediate representation in which each variable is defined at most once in the program code [17, 57]. Many industrial compilers use the SSA form as an intermediate representation: Gcc 4.0 [33], Sun's HotSpot JVM [62], IBM's Java Jikes RVM [63] and LLVM [43]. However, these compilers perform register allocation in *Post-SSA programs*, that is, programs in which $\phi$-functions have been replaced with copy instructions, as shown in Figure 12(a). In SSA-based register allocators [7, 12, 35, 55] we observe an inversion of phases: $\phi$-functions are replaced after registers have been assigned to variables, as illustrated in Figure 12(b). We will use *colored-SSA-form* to denote the variation of SSA-form in which each variable is associated with a physical location, which can be a register or a memory address. Figure 13 shows an SSA-form program and a corresponding colored-SSA-form program.

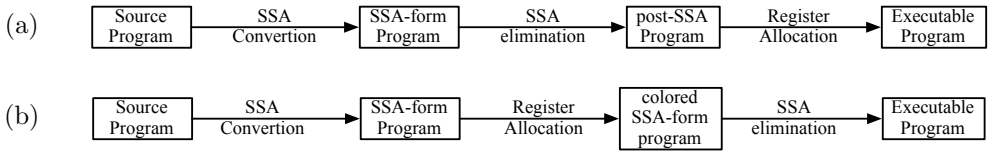



Figure 12: (a) Traditional register allocation, (b) SSA-based register allocation.

**$\phi$-functions** SSA form uses $\phi$-functions to join the live ranges of different names that represent the same value. We will describe the syntax and semantics of $\phi$-functions using the matrix notation introduced by Hack et al. [35]. Figure 13 (a) outlines a $\phi$-matrix. An equation such as $V = \phi M$, where $V$ is a $n$-dimensional vector, and $M$ is a $n \times m$ matrix, contains $n$ $\phi$-functions and $m$ *parallel copies*, as outlined in Figure 14. Columns in the matrix correspond to control flow paths. The $\phi$ symbol works as a multiplexer. It will assign to each element $v_i$ of $V$ an element $v_{ij}$ of $M$, where $j$ is determined by the actual path taken during the program's execution. The semantics of $\phi$-functions have been nicely described in [1]. The parameters of a $\phi$-function are evaluated simultaneously, at the beginning of the basic block where the $\phi$-function is defined. Thus, a $\phi$-equation $V = \phi M$, where $M$ has $n$ columns encodes $n$ parallel copies. If the path leading to column $j$ is taken during program execution, all the elements in that column are copied to $V$ in parallel.

**Chordal Graphs** A graph is chordal if every cycle with four or more edges has a *chord*, that is, an edge which is not part of the cycle but which connects two vertices on the cycle. (Chordal graphs are also known as 'triangulated', 'rigid-circuit', 'monotone transitive', and 'perfect elimination' graphs.) The graph in Figure 15(a) is chordal because the edge *ac* is a chord in the cycle *abcda*. The graph in Figure 15(b) is non-chordal because the cycle *abcda* is chordless. Finally, the graph in Figure 15(c) is non-chordal because the cycle *abcda* is chordless, just like in Figure 15(b).

Chordal graphs have several useful properties. Problems such as *minimum coloring*, *maximum clique*, *maximum independent set* and *minimum covering by cliques*, which are NP-complete in general, can be solved in polynomial time for chordal graphs [28]. In particular, optimal coloring of a chordal graph $G = (V, E)$ can be done in $O(|E| + |V|)$ time.

**The Dominator Tree** A basic block $L_i$ *dominates* another basic block $L_j$ if very path from the start of the program to $L_j$ goes through $L_i$ [3, p.379]. A basic block $L_d$ is the *immediate dominator* of another block $L$ if $L_d$ dominates $L$, and for all other basic blocks $L_i$ in the program, if $L_i$ dominates $L$, then $L_i$ also

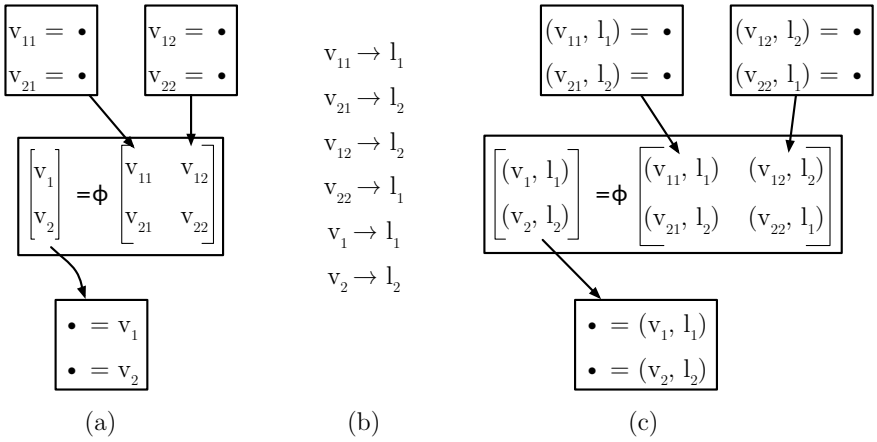



Figure 13: (a) SSA-form program. (b) Register assignment. (c) colored-SSA-form program.

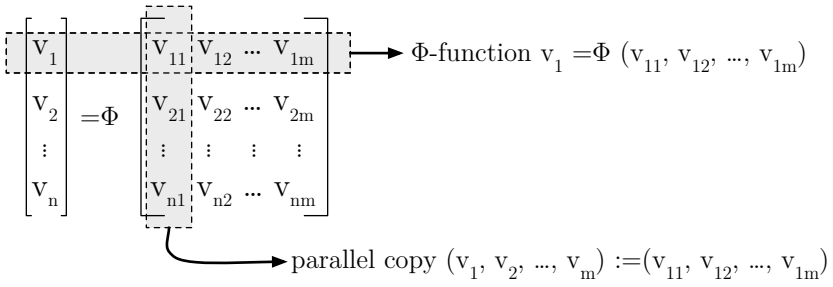


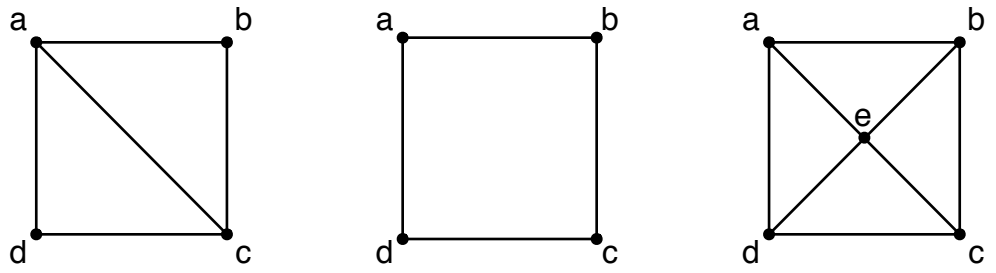

Figure 14: The $\phi$-matrix.

dominates $L_d$. Every basic block in a program, except its entry point, has an immediate dominator, and this immediate dominator is unique. Thus, this relation of immediate dominance allows us to build a tree $T = (V, E)$ whose vertices are the basic blocks of a program, and an edge $(L_s, L_t)$ is in $E$ if basic block $L_s$ is the immediate dominator of basic block $L_t$. This tree is called the *Dominator Tree* of the program.

The key insight to understand why SSA-form programs have chordal interference graphs lays on a well known characterization of chordal graphs: these are the intersection graphs of subtrees on a tree, as shown by Gavril in 1974 [29]. The live ranges of a SSA-form program are subtrees of the *dominator tree* of the program. By applying Gavril's result to the dominator trees of programs, Bouchez *et al.* [7], Brisk *et al.* [13] and Hack *et al.* [35] proved that the interference graph of a SSA-form program is chordal. The opposite direction is also true: a chordal graph is the interference graph of some SSA-form program [55].

## 3.1 The Advantages of SSA-Based Register Allocation

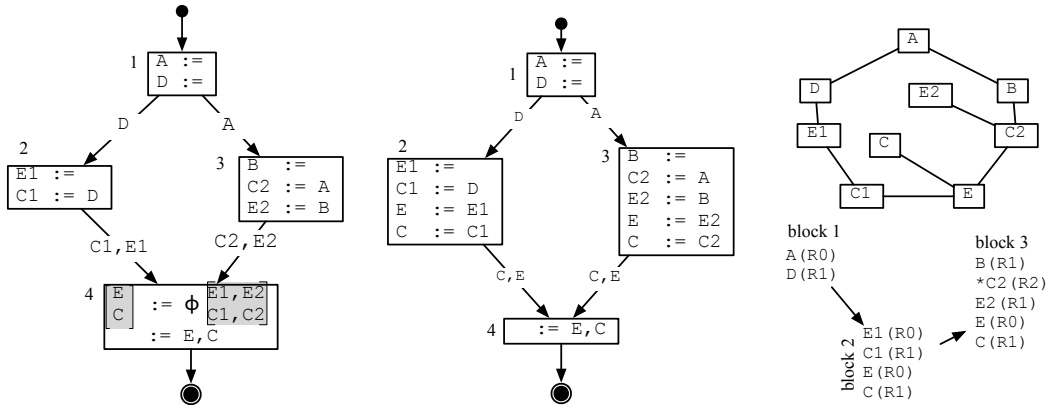
We illustrate the simplicity and elegance of SSA-based register allocation with an example. The program in Figure 16 was taken from [53]. Figure 16 (b) is the same program in post-SSA form, that is, after $\phi$-functions have been replaced using the algorithm described by Appel and Palsberg [3]. Figure 16 (c) shows the sequence of assignments that would be performed by a linear scan algorithm that handles holes in live ranges of variables [65]. This algorithm would traverse blocks 1, 2, 4(where it executes no action), and finally block 3. When allocating registers in block 3, the allocator has to deal with temporaries `C` and `E` that have




(a) (b) (c)

Figure 15: (a) A chordal graph. (b-c) Two non-chordal graphs.

already been assigned machine registers. The graph in Figure 16 (c) cannot be colored with two colors. Its chromatic number is 3.

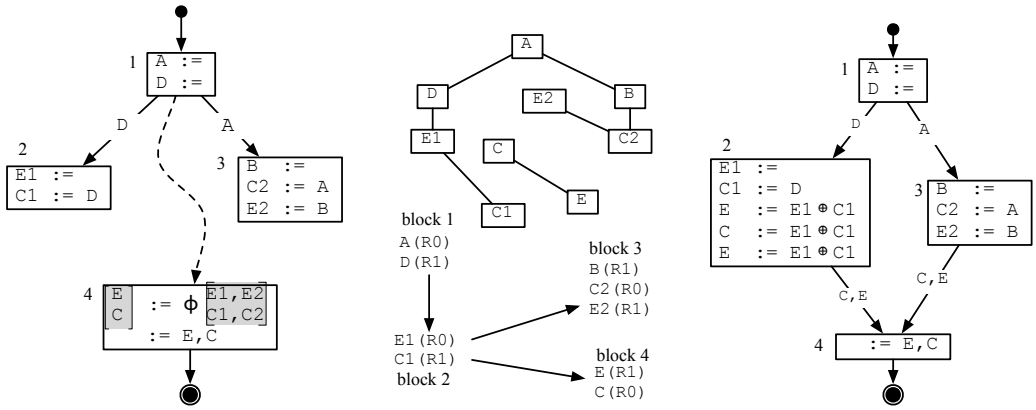



(a) (b) (c)

Figure 16: (a) SSA-form program taken from Pereira and Palsberg [53]. (b) Program after the SSA elimination phase. (c) Interference graph and sequence of register assignments.

Figure 17 (a) outlines the dominator tree of our example program. Figure 17 (b) shows the allocation produced by the same algorithm used in Figure 16. The basic blocks are visited in a pre-order traversal of the dominator tree. This way to assign registers to variables is called *tree-scan*, to distinguish it from the linear-scan strategy. The interference graph of the SSA-form program can be compiled with two registers: one register less than the minimum necessary in the post-SSA-form program. Figure 17 (c) shows the final program, after one swap sequence has been inserted in order to copy the values of `C1` and `E1` into `C` and `E`. The ability to swap registers is necessary in order to keep the register pressure low, as described by Bouchez *et al.* [8]. In this example, swaps are implemented using three xor operations.

Any of the register allocation models described in Section 2 can be adapted to run on SSA-form programs. The chordal-based register allocator described by Pereira and Palsberg [53] follows the graph coloring model. The SSA-based allocator described by Grund *et al.* [34] uses integer linear programming, and the puzzle solving algorithm introduced by Pereira and Palsberg [55] is based on the tree-scan model. Register allocators can benefit from chordality of SSA-form programs in three main ways: (i) lower register pressure; (ii)




(a) (b) (c)

Figure 17: (a) Dominator tree of the example program. (b) Interference graph of the SSA-form program, and assignment sequence. (c) Final program after SSA-based register allocation.

separation between spilling and register assignment. (iii) simpler register assignment algorithms.

First, the SSA-form program never requires more registers than the original program, and often it will require less, as we showed in the previous example. The register pressure at any program point is the difference between the number of variables alive at that point and the number of registers available to accommodate those variables [26]. The total number of registers necessary to allocate all the variables in a SSA-form program $P$ equals the maximum register pressure at any point of $P$ [35]. This value is equivalent to the size of the largest clique in the interference graph of $P$, and it can be determined in time proportional to the number of edges of this graph [28]. This relation is valid for register banks with no aliasing. The problem of determining the maximal register pressure for an architecture with aliased registers is NP-complete for SSA-form programs [44].

Another advantage of SSA-based register allocators is the potential separation of phases between spilling, register assignment and coalescing. The register pressure at any point of a SSA-form program is known locally. This fact allows the register allocator to remove live ranges from the program until the register pressure equals the number of available registers. Thus, the allocator is able to take spill decisions without having to actually assign registers to variables. In a subsequent phase, registers are assigned to variables, and the SSA properties guarantee that no further spilling will happen. Once registers are assigned to variables, a third phase takes charge of improving the initial register assignment, so that variables related by copies are given the same register. Figure 18 illustrates a typical SSA-based register allocator. This algorithm has five phases:

**Build** builds the interference graph using liveness analysis information.

**Spill** remove live ranges if the register pressure is greater than the number of available registers.

**MCS** finds an ordering of the nodes of the graph that can be optimally colored by a greedy algorithm using the Maximum Cardinality Search algorithm [5].

**Color** assign registers to variables using a trivial greedy coloring algorithm.

**Coalesce** exchange registers between variables in order to maximize the number of variables related by copy instructions that are given the same register.

Figure 18 sheds light on a third advantage of SSA-based register allocation: simplicity. The iterations between the spilling and the register assignment phases complicate the design of register allocators in general.

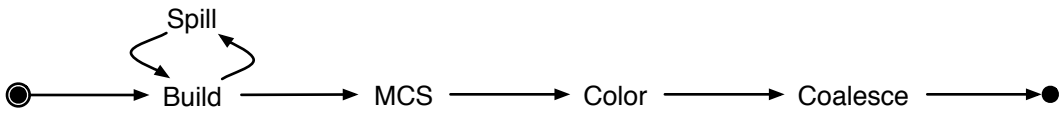



Figure 18: A graph coloring SSA-based register allocator.

This problem is particularly evident in graph coloring based allocators, as a comparison between Figure 18 and Figures 6, 8, 9 would reveal. Because spill decisions are taken independently of assignment decisions in a SSA-based allocator, the implementation of these algorithms tend to be simpler.

# 4 NP-complete Results

Spill Free Register Allocation has polynomial time solution for SSA-form programs [7, 13, 35], but it is NP-complete for programs in general [16]. One point that must be emphasized is that these two problems are obviously non-equivalent. Any program can be converted into SSA-form via a polynomial time transformation [17]. However, a register assignment for a SSA-form program cannot be converted back to an optimal register assignment of the original program in polynomial time unless P=NP.

An illustrative analogy is between the coloring of interval graphs and the coloring of circular-arc graphs. An interval graph is built on the following way: given a number of intervals on a line, assign a vertex to each interval. If two lines overlap, then connect their corresponding vertices. Circular-arc graphs are defined in a similar way, but using a circle instead of a line, and arcs on the circle instead of intervals. Finding the chromatic number of a circular-arc graph is a NP-complete problem [45], whereas the same problem for interval graphs has polynomial time solution [31]. To close our analogy, lets imagine that we can "cut" the base circle and all the arcs of a given circular-arc graph $G_c$ on a given point. This cut effectively changes $G_c$ into an interval graph $G_i$. We can find the chromatic number of $G_i$ in polynomial time, and this value will never be larger than the chromatic number of $G_c$; however, finding the chromatic number of $G_c$ is a NP-complete problem. Transforming a circular-arc graph to an interval graph in this way is analogous to converting a program into SSA-form.

Unfortunately, many register allocation related problems are still NP-complete even for SSA-form programs. Because SSA-form programs are a subset of general programs, these problems are, naturally, also NP-complete in general. We describe a number of these problems in this section.

**Spilling** The first of our NP-complete problems is *spill minimization*. When spills happen, loads and stores are inserted into the source program to transfer values to and from memory. If we assume that each load and store has a cost, then the problem of minimizing the total cost added by spill instructions is NP-complete, even for basic blocks in SSA-form, as shown by Farach and Liberatore [25]. If the cost of loads and stores is not taken into consideration, then a simplified version of the spilling problem is to determine the minimum number of variables that must be removed from the source program so that the program can be allocated with $K$ registers. This problem is equivalent to determining if a graph $G$ has a $K$-colorable induced subgraph, which is NP-complete for chordal graphs, but has polynomial time solution for interval graphs, as demonstrated by Yannakakis and Gavril [66].

**Coalescing** Coalescing is another part of register allocation that remains NP-complete even for SSA-form programs [19]. Ferriere *et al.* have proved that aggressive coalescing is NP-complete for SSA-form programs when the size of the $\phi$-functions is unbounded [19]. Bouchez *et al.* [9] took Ferriere's study to the extreme, proving that all the best known variations of the coalescing problem, which are described in Section 2.1.1, are NP-complete. The proofs used in [9] have the extra appeal of relying on bounded structures such as $\phi$-functions of size at most three.

**Live range splitting** SSA-based register allocators rely on the ability to swap the live ranges of variables without using extra registers to store temporary values. For instance, the live ranges of variables `E1` and `C1` are swapped in the basic block four of Figure 17 (c). Pereira and Palsberg [54] have proved that if swaps are not used by the register allocator, then the problem of deciding if a SSA-form program can be compiled with $K$ registers is NP-complete, although the problem of deciding if a program can be compiled with $K-1$ registers has polynomial time solution, as long as the $K$-th register is used as a temporary storage location. The proof in [54] considers that live ranges of variables can be split only at the end of basic blocks. Bouchez *et al.* [8] later refined this proof to show that the problem remains NP-complete even if live ranges are allowed to be split at any program point.

**Aliasing** Another factor that complicates register allocation is aliasing, described in Section 1.1. The problem of finding an optimal register assignment for a target architecture that allows registers to alias is NP-complete even for basic blocks in SSA-form as proved by Lee *et al.*

**Pre-coloring** Register allocation with pre-coloring is equivalent to the *pre-coloring extension problem* for graphs. In this problem we are given a graph $G$, an integer $K$ and a partial function $\varphi$ that associates some vertices of $G$ to colors. The challenge is to extend $\varphi$ to a total function $\varphi'$ such that (1) $\varphi'$ is a valid coloring of $G$ and (2) $\varphi'$ uses less than $K$ colors. Biró *et al.* [6] have shown that pre-coloring extension is NP-complete for interval graphs, and thus, register assignment for basic blocks in SSA-form with pre-colored registers is also NP-complete. Interestingly, pre-coloring extension is NP-complete even for unit interval graphs [46], that is, interval graphs in which each interval has the same size.

Aliasing and pre-coloring cause SSA-based register allocation to be NP-complete; however, these problems have polynomial time solution if we use a program form more restrictive than SSA. This format is called *Elementary Form*, and it is the subject of our next section.

# References


[1] Andrew W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[2] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *PLDI*, pages 243–253. ACM Press, 2001.

[3] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.

[4] Rajkishore Barik, Christian Grothoff, Rahul Gupta, Vinayaka Pandit, and Raghavendra Udupa. Optimal bitwise register allocation using integer linear programming. In *LCPC*, volume 4382 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2006.

[5] Anne Berry, Jean Blair, Pinar Heggernes, and Barry Peyton. Maximum cardinality search for computing minimal triangulations of graphs. *Algorithmica*, 39(4):287 – 298, 2004.

[6] M Biró, M Hujter, and Zs Tuza. Precoloring extension. I:interval graphs. In *Discrete Mathematics*, pages 267 – 279. ACM Press, 1992. Special volume (part 1) to mark the centennial of Julius Petersen's "Die theorie der regularen graphs".

[7] Florent Bouchez. Allocation de registres et vidage en mémoire. Master's thesis, ENS Lyon, October 2005.

[8] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the NP-completeness proof of chaitin et al. really prove? In *5th Annual Workshop in Duplicating, Deconstructing, and Debunking*, 2006.

[9] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. In *CGO*, pages 102 – 104. IEEE, 2007.

[10] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *PLDI*, pages 311–321, 1992.

[11] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, 1994.

[12] Philip Brisk, Foad Dabiri, Roozbeh Jafari, and Majid Sarrafzadeh. Optimal register sharing for high-level synthesis of ssa form programs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(5):772–779, 2006.

[13] Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial-time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*. ACM Press, 2005.

[14] Rainer E. Burkard, Eranda Çela, Panos M. Pardalos, and Leonidas S. Pitsoulis. Quadratic assignment problems. *European Journal of Operational Research*, 15:283–289, 1984.

[15] G. J. Chaitin. Register allocation and spilling via graph coloring. *Symposium on Compiler Construction*, 17(6):98–105, 1982.

[16] Gregory J. Chaitin, Mark A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[17] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[18] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill Science/Engineering/Math, 2006.

[19] François de Ferriére, Christophe Guillon, and Fabrice Rastello. Optimizing the translation out-of-SSA with renaming constraints. *ST Journal of Research Processor Architecture and Compilation for Embedded Systems*, 1(2):81–96, 2004.

[20] Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. Generalized instruction selection using SSA-graphs. In *LCTES*, pages 31–40, 2008.

[21] Erik Eckstein, Oliver König, and Bernhard Scholz. Code instruction selection based on ssa-graphs. In *SCOPES*, pages 49–65, 2003.

[22] Erik Eckstein and Bernhard Scholz. Addressing mode selection. In *CGO*, pages 337–346, 2003.

[23] S Even, A Itai, and A Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM J. Computing*, 5(4):691 – 703, 1976.

[24] Alkis Evlogimenos. Improvements to linear scan register allocation. Technical report, University of Illinois, Urbana-Champaign, 2004.

[25] Martin Farach and Vincenzo Liberatore. On local register allocation. In *9th ACM-SIAM symposium on Discrete Algorithms*, pages 564–573. ACM Press, 1998.

[26] Martin Farach-colton and Vincenzo Liberatore. On local register allocation. *Journal of Algorithms*, 37(1):37–65, 2000.

[27] Changqing Fu and Ken D. Wilken. A faster optimal register allocator. In *Internation Symposium on Microarchitecture*, pages 245–256. ACM, 2002.

[28] Fanica Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SICOMP*, 1(2):180 – 187, 1972.

[29] Fanica Gavril. The intersection graphs of subtrees of a tree are exactly the chordal graphs. *Journal of Combinatoric*, B(16):46 – 56, 1974.

[30] Lal George and Andrew W. Appel. Iterated register coalescing. *Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):300–324, 1996.

[31] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Elsevier, 1st edition, 2004.

[32] David W. Goodwin and Ken D. Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *SPE*, 26(8):929–965, 1996.

[33] Brian J. Gough. *An Introduction to GCC*. Network Theory Ltd, 1st edition, 2005.

[34] Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. In *Compiler Construction*, volume 4420, pages 111–115. Springer, 2007.

[35] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *15th Conference on Compiler Construction*, pages 247–262. Springer-Verlag, 2006.

[36] Lang Hames and Bernhard Scholz. Nearly optimal register allocation with PBQP. In *JMLC*, pages 346–361. Springer, 2006.

[37] Ulrich Hirnschrott, Andreas Krall, and Bernhard Scholz. Graph coloring vs. optimal register allocation for optimizing compilers. In *JMLC*, pages 202–213, 2003.

[38] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum, 1972.

[39] A. B. Kempe. On the geographical problem of the four colours. *American Journal of Mathematics*, 2(3):193 – 200, 1879.

[40] David Koes and Seth Copen Goldstein. A progressive register allocator for irregular architectures. In *CGO*, pages 269–280, 2005.

[41] David Ryan Koes and Seth Copen Goldstein. A global progressive register allocator. In *PLDI*, pages 204–215, 2006.

[42] Timothy Kong and Kent D Wilken. Precise register allocation for irregular architectures. In *Internation Symposium on Microarchitecture*, pages 297–307. ACM, 1998.

[43] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[44] Jonathan K. Lee, Jens Palsberg, and Fernando M. Q. Pereira. Aliased register allocation. In *ICALP*, 2007.

[45] Daniel Marx. A short proof of the NP-completeness of circular arc coloring, 2003.

[46] Daniel Marx. Precoloring extension on unit interval graphs. *Discrete Applied Mathematics*, 154(6):995 – 1002, 2006.

[47] Hanspeter Mossenbock and Michael Pfeiffer. Linear scan register allocation in the context of SSA form and register constraints. In *CC*, pages 229–246. LNCS, 2002.

[48] Mayur Naik and Jens Palsberg. Compiling with code size constraints. *Transactions on Embedded Computing Systems*, 3(1):163–181, 2004.

[49] V. Krishna Nandivada, Fernando Pereira, and Jens Palsberg. A framework for end-to-end verification and evaluation of register allocators. In *Proceedings of SAS'07, International Static Analysis Symposium*, pages 153–169, Kongens Lyngby, Denmark, August 2007.

[50] Venkata Krishna Nandivada and Jens Palsberg. SARA: Combining stack allocation and register allocation. In *15th International Conference on Compiler Construction*. Springer-Verlag, 2006.

[51] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. In *IEEE PACT*, pages 196–204, 1998.

[52] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst.*, 26(4):735–765, 2004.

[53] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *APLAS*, pages 315–329. Springer, 2005.

[54] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation after classic SSA elimination is np-complete. In *Foundations of Software Science and Computation Structures*. Springer, 2006.

[55] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation by puzzle solving. In *PLDI*, pages 216–226, 2008.

[56] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.

[57] B. K. Rosen, F. K. Zadeck, and M. N. Wegman. Global value numbers and redundant computations. In *POPL*, pages 12–27. ACM Press, 1988.

[58] Konstantinos Sagonas and Erik Stenman. Experimental evaluation and improvements to linear scan register allocation. *Software, Practice and Experience*, 33:1003–1034, 2003.

[59] Vivek Sarkar and Rajkishore Barik. Extended linear scan: an alternate foundation for global register allocation. In *LCTES/CC*, pages 141–155. ACM, 2007.

[60] Bernhard Scholz and Erik Eckstein. Register allocation for irregular architectures. In *LCTES/SCOPES*, pages 139–148. ACM, 2002.

[61] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI*, pages 277–288, 2004.

[62] JVM Team. The java HotSpot virtual machine. Technical Report Technical White Paper, Sun Microsystems, 2006.

[63] The Jikes Team. Jikes RVM home page, 2007. http://jikesrvm.sourceforge.net/.

[64] Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 142–151, 1998.

[65] Christian Wimmer and Hanspeter Mossenbock. Optimized interval splitting in a linear scan register allocator. In *VEE*, pages 132–141. ACM, 2005.

[66] Mihalis Yannakakis and Fanica Gavril. The maximum k-colorable subgraph problem for chordal graphs. *Information Processing Letters*, 24(2):133 – 137, 1987.