

# Register Allocation for General Purpose Architectures

*Peter Keyngnaert*

*Report CW 281, March 2000*



Katholieke Universiteit Leuven  
Department of Computer Science  
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Register Allocation for General Purpose Architectures

*Peter Keyngnaert*

*Report CW 281, March 2000*

Department of Computer Science, K.U.Leuven

## **Abstract**

Over the years, register allocation has been recognized as one of the vital problems to solve with respect to compiler design. Putting critical data values and/or addresses into the small but very fast memories that registers are, is essential for achieving high-speed code. This document gives an overview of the most important work in the field, detailing some breakthrough approaches while briefly mentioning a lot of results that were derived from them.

# Register Allocation for General Purpose Architectures

Peter Keyngnaert  
Department of Computer Science  
Katholieke Universiteit Leuven  
B-3001 Heverlee, Belgium  
peter.keyngnaert@cs.kuleuven.ac.be

## Abstract

Over the years, register allocation has been recognized as one of the vital problems to solve with respect to compiler design. Putting critical data values and/or addresses into the small but very fast memories that registers are, is essential for achieving high-speed code. This document gives an overview of the most important work in the field, detailing some breakthrough approaches while briefly mentioning a lot of results that were derived from them.

## 1 Introduction

A good register allocator is an important part of today's compilers. This section illustrates why this is so and introduces some vital terminology the reader should be familiar with.

### 1.1 Goals and terminology

**the need for good register use** The hierarchical structure of the memory model of a load/store architecture as shown in figure 1 plays a crucial role in the execution speed of a program run on such a computer. As the capacity of the different memories increases, so decreases their speed. This implies that generating optimal code (where optimality is defined with respect to the execution time of the program) requires optimal use of the fastest memory type which, in the case of the type of architectures considered here, is the register set. Current techniques for automatic register allocation are far from generating optimal code in the strictest sense of the word (since the register allocation problem is known to be NP-complete, see e.g. [14] or [59] where it is stated that “given a program and an integer  $k$ , determining if the program can be computed using  $k$  registers is polynomial complete” and “register allocation belongs to a class of problems for which there exists no nonenumerative solution” respectively), but it should be obvious that even so, heuristics for good use of the register set are key elements when optimizing code for speed. Both CISC and RISC machines benefit from a good register allocation: in the instruction set of the former, instructions that operate on registers are faster than the ones that do not, while the latter has a lot of operations that can only be performed by instructions where all operands are in registers.

**goals** The goal of register allocation is to *try to minimize to number of cpu accesses to the cache or to the main memory by putting as much data as possible in registers*. Alternatively, this can be restated as follows: *register allocation tries to maximize instruction level parallelism by putting operands of instructions that can be executed concurrently, into registers*.

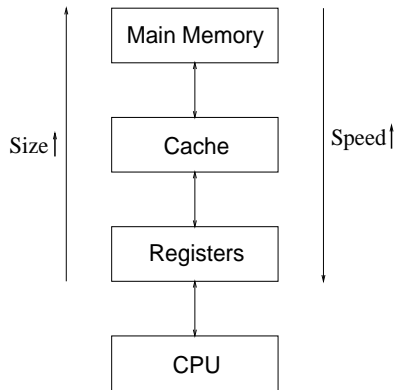


Figure 1: Memory size and speed hierarchy of a typical load/store architecture.

**liveness** *Liveness* is an important concept within the context of register allocation. Intuitively, a variable is *live* if it holds a value that may be needed in the future. As such, a variable  $v$  is live at a program point  $p$  if all of these conditions hold:

- $v$  has been defined in a statement that precedes  $p$  in any path
- $v$  may be used by a statement  $s$  and there is a path from  $p$  to  $s$
- $v$  is not killed between  $p$  and  $s$

The *live range* of  $v$  is the interval  $[p_l, p_d]$  with  $v$  being defined by statement  $p_l$  and  $v$  last used in statement  $p_d$ .

**domain** Since `load` and `store` operations, as well as address calculations, need to be explicitly modeled, the domain to work in will be the one of either low-level intermediate code or assembly language.

**register allocation vs. register assignment** Also, note the difference between register allocation and register assignment. Register allocation is the selection process of those variables that will reside in registers at a certain point in time during the execution of the program. The subsequent register assignment phase then picks the specific register that an allocated variable will reside in. This separation of concerns is based on what John Backus stated during the development of the first Fortran compiler, i.e. that *the optimization of subscript expressions should be handled separately from the allocating of index registers*. *Register assignment* can be trivial in some architectures, but in others it is not since there may be different register sets for e.g. integers and floating point numbers.

## 1.2 Overview

Section 2 of this document gives an overview of the existing methods that handle register allocation at the different levels of abstraction of a program where the technique is relevant. In section 3 a brief introduction to instruction scheduling shows its connectivity with register allocation. Also, a summary of various efforts to integrate both techniques is presented and register windows are mentioned. In section 4 some pointers to optimal register allocation techniques are given. As a general note for this document: while the basic methods for doing register allocation are presented up to a certain detail, the variations upon them are mentioned only very briefly.

## 2 Register allocation at different levels

Register allocation can be applied at various levels during code generation: at the expression level, at the local level (i.e. for a basic block<sup>1</sup>), at the global level (for a procedure) and finally at the interprocedural level (for the entire program).

### 2.1 Register allocation at the expression and/or basic block level

Given an expression<sup>2</sup>, how can it be evaluated as fast as possible? If the assumption is made that each instruction takes 1 cpu cycle, the quest is to find the shortest sequence of instructions that does the job. To access data in memory, `load` and `store` instructions exist. Since the goal is to minimize the number of instructions of the expression evaluation routine and most binary mathematical operations have either two registers or a register and an address in memory as their source operands, this really is a register allocation problem also: the more values that are kept in registers, the less instructions will be generated and hence the faster the code will be.

#### 2.1.1 For DAGs that are trees

In [60] a now classic algorithm is given that solves the above problem for expressions without common subexpressions. The algorithm is designed for a machine with an unlimited storage capacity and  $N \geq 1$  register(s) which has the following 4 commands:

1.  $C(\text{storage}) \rightarrow C(\text{register})$
2.  $C(\text{register}) \rightarrow C(\text{storage})$
3.  $OP[C(\text{register}), C(\text{storage})] \rightarrow C(\text{register})$
4.  $OP[C(\text{register}), C(\text{register})] \rightarrow C(\text{register})$

where the first one is a `load`, the second one a `store`, the third one a binary operation where the first operand is in a register, the second operand resides in memory and the result goes into a register and the fourth is a binary operation where both operands are in registers and the result goes into a register also. A program  $P$  is a sequence of operations (instructions) which evaluate an expression for which the initial values are stored in specific locations in memory. Because of the assumption about the execution time of instructions (cfr. *supra*), the *cost of  $P$*  is the number of program steps (instructions) needed to evaluate the expression.  $P$  must terminate with the result of the expression's evaluation in some register. The basic idea is to label each node  $\eta$  in the binary graph  $G$  that represents the expression (as an example, the graph for the expression  $a/(b+c) - d * (e+f)$  is given in figure 2) and afterwards have an algorithm, that uses this labeled tree as its input, produce a minimal sequence of instructions. A node's label then, is the minimal number of registers required to evaluate that node (i.e. generate code to evaluate the expression the binary tree with this node as its root stands for) without the use of any `store` instructions. In fact, not 1 but 3 algorithms are given in the paper, as the initial algorithm only works for

---

<sup>1</sup>[2] defines a basic block as a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end

<sup>2</sup>In this context, an expression is a sequence of binary operations on arguments where the arguments can either be initial (defined externally) or intermediate (defined by operations on other arguments). Expressions can be represented by binary trees where the leaf nodes correspond to initial values and the interior nodes correspond to intermediate values, their left and right descendants respectively representing the first and second operand of the operation.

non-commutative operations and is then extended, first to also allow commutative operations and finally to a version that handles (a form of) associativity as well. Also, the extra assumption is made that there are no non-trivial relations between operations and the elements they operate on. For simplicity/clarity, only the basic algorithm will be presented here.

## 1. LABELING PHASE

$\forall \eta \in G$  : define label  $L(\eta)$  from the bottom up, according to these rules :

L1 If  $\eta$  is both a leaf node and a left descendant of it's ancestor then  $L(\eta) = 1$ . If  $\eta$  is both a leaf node and a right descendant of it's ancestor then  $L(\eta) = 0$ .

L2 If  $\eta$  has descendants with labels  $l_{left}$  and  $l_{right}$  then  $L(\eta) = \max(l_{left}, l_{right})$  if  $l_{left} \neq l_{right}$  and  $L(\eta) = l_{left} + 1$  if  $l_{left} = l_{right}$ .

## 2. CODE GENERATION PHASE

Let there be  $N$  registers  $r_1, \dots, r_N$ . The following algorithm is applied to the root node of  $G$ :

- Let the current node be  $\eta$  with  $L(\eta) > 0$  and registers  $r_m, \dots, r_N$  available ( $1 \leq m \leq N$ ).

$L(\eta) = 1$  **If  $\eta$  is a leaf node** then it must be a left descendant so generate  $r_m = \eta$ <sup>3</sup> so we generate

$C(\text{left descendant of } \eta) \rightarrow C(r_m)$

**If  $\eta$  is not a leaf node**, it's right descendant must be a leaf node (otherwise  $L(\eta) \geq 2$  would hold whereas here we have  $L(\eta) = 1$ ) so we assign to  $r_m$  the resulting value of the algorithm applied to the binary tree with  $\eta$  as its root node and generate

$OP(C(r_m), C(\text{right descendant of } \eta)) \rightarrow C(r_m)$

$L(\eta) > 1$  **If  $l_1 \geq N, l_2 \geq N$**  then apply the algorithm to the right descendant of  $\eta$  with registers  $r_m, \dots, r_N$  available. Generate a store of the value of the right descendant of  $\eta$

$C(\text{storage}) \rightarrow C(\text{storage})$

Assign the result of applying the algorithm to the left descendant of  $\eta$  to  $r_m$  with registers  $r_m, \dots, r_N$  available and generate

$OP(C(r_m), C(\text{storage})) \rightarrow C(r_m)$

**If  $l_1 \neq l_2$  and at least one of  $l_1, l_2 \leq N$**  then store the resulting value of applying the algorithm to the descendant of  $\eta$  with the highest value in  $r_m$  (with  $r_m, \dots, r_N$  available), store the resulting value of applying the algorithm to the descendant of  $\eta$  with the lowest value in  $r_{m+1}$  (with  $r_{m+1}, \dots, r_N$  available) and generate

$OP(C(r_m), C(r_{m+1})) \rightarrow C(r_m)$

There's an exception to this rule: if the smallest label is 0, do not apply the algorithm because if the label is 0 then the value resides in main memory (storage)!

Here's an example that shows the expression tree (figure 2) and the generated code for the expression  $a/(b+c) - d * (e+f)$ . The machine the code was written for has only 2 registers,  $r_1$  and  $r_2$ . To evaluate the given expression, only one store (line 5) of a value into main memory (at

---

<sup>3</sup>What we mean by  $r_i = \eta$  is that the value at node  $\eta$  is assigned to register  $r_i$ .

location T) was needed:

- 1)  $d \rightarrow r_1$
- 2)  $e \rightarrow r_2$
- 3)  $r_2 + f \rightarrow r_2$
- 4)  $r_2 * r_1 \rightarrow r_1$
- 5)  $r_1 \rightarrow T$
- 6)  $a \rightarrow r_1$
- 7)  $b \rightarrow r_2$
- 8)  $r_2 + c \rightarrow r_2$
- 9)  $r_1 / r_2 \rightarrow r_1$
- 10)  $r_1 - T \rightarrow r_1$

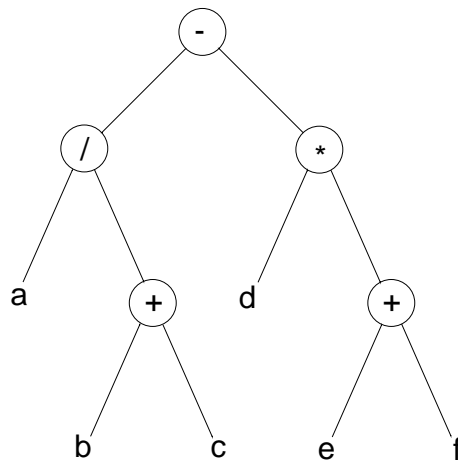


Figure 2: Expression graph for  $a/(b+c) - d*(e+f)$

In brief, the way commutative operators are handled is to select that member of the family of trees that compute the same value that minimizes:

- the highest label in the tree
- the number of major nodes
- the number of minor nodes

where a node is a *major node* if both of its descendants have labels  $\geq N$  and a *minor node* if it's a leaf node as well as the left descendant of its immediate ancestor node.

Instead of just do register allocation for an expression, it can also be done for an entire basic block. Farach and Liberatore ([24], [40]) show that this problem is NP-complete also.

### 2.1.2 For DAGs that are not trees

[59] states that the problem of local register allocation is NP-complete when the DAG (*Directed Acyclic Graph*) that represents the expression is no longer a tree. As such, no single algorithm exists that efficiently solves this problem. In [34], [32] and [33] an approach is illustrated that tackles this conclusion by the observation that several algorithms exist that perform well for some

particular kind of DAGs. Their heuristic executes all these algorithms in parallel on the DAG subject to register allocation. The best result is then picked out of all the different solutions. As the DAGs typically encountered in real programs belong to a few simple classes, in practice this approach works quite well. Apart from some variations on *depth-first* strategies, they also introduce a *randomized* evaluation strategy: a random set of bit-vectors representing different orderings of the instructions in the DAG are generated and evaluated and the best one of them is used for allocation. The paper shows that this random approach has a very good chance of generating a solution that is close to optimal.

A DAG  $G(V, E)$  is *contiguous* if, when evaluating it, the following holds:

$$\forall v \in V : w_i \in V \text{ is a predecessor of } v \text{ and } ord(v_1) < ord(v_2) \rightarrow ord(w_1) < ord(w_2)$$

where  $v_1, v_2$  are children of  $v$ ,  $w_1, w_2$  are children of  $w$  and  $ord(x) < ord(y)$  means that node  $x$  is visited before node  $y$ . In other words, a contiguous evaluation of a node  $v$  first visits the complete subDAG of a child of that node before it visits the rest of the subDAG with  $v$  as a root. [36] and [35] note that almost all DAGs encountered in real programs are contiguous. The algorithm works by splitting the DAGs into trees and perform register allocation on these trees.

## 2.2 Register allocation at the procedure level

### 2.2.1 Chaitin's now classic approach: register allocation as graph coloring

The still widely used method of using graph coloring to perform global register allocation across an entire procedure was first presented in [17], although the idea was not entirely new as it was already hinted at in [3], [66] and [58]. The essential benefit of this technique is the uniform way in which all machine idiosyncracies are integrated into the data structure and the systematic way of dealing with them. The problem of graph coloring can be stated as follows: given a graph  $G$  and  $n > 2$ , is there a way to color the nodes of this graph using at most  $n$  colors and having the restriction that no two nodes that have an edge between them have the same color. [1] and [25] shows that the decision whether such a coloring is possible is NP-complete. The implication of this result is that there will always be certain programs for which there will be serious coloring problems. The basic structure of Chaitin's so called *Yorktown allocator* is illustrated in figure 3.

**The interference graph** For each procedure in the program, a (*register*) *interference graph* is constructed and a coloring is obtained for it. Each node in the graph corresponds to a *name*. Normally, an unlimited number of symbolic registers is used in code generation before the actual register allocation is done. Each symbolic register is split into the connected components of its *def-use* chains<sup>4</sup> and these components are called *names*. The benefit of this is that this splitting generates an additional coloring freedom since distant regions of the program are now uncoupled. Therefore, the mapping of symbolic registers onto names now transforms register allocation from a *one-to-many* mapping into a *many-to-many* mapping: first, we mapped a single, physical register onto a number of different symbolic registers. Now, we may map more than one physical register onto the same symbolic register (but in different parts of the code). Chaitin also uses a special notion of liveness:

---

<sup>4</sup>the *def-use chain* for a certain def(inition) of a variable is the list of all possible uses of that def(inition)

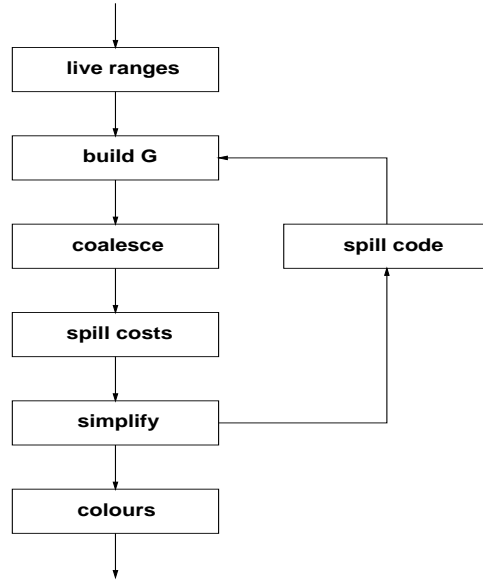


Figure 3: Chaitin's *Yorktown* allocator

A name  $X$  is live at point  $L$  in program  $P \Leftrightarrow \exists$  a control flow path from the entry point of  $P$  to a definition of  $X$  and then through  $L$  to a use of  $X$  at point  $U$

which means that there is no redefinition of  $X$  on the path between  $L$  and the use of  $X$  at  $U$ : a name is live if it's computed and used before it's recomputed. As for the notion of *interference*, two nodes are said to *interfere* if their corresponding names are ever live simultaneously.

Now:

**IF** at a point in  $P$ :  $\exists k$  live names  $N_i$  ( $1 \leq i \leq k$ ) **THEN**  $k(k-1)/2$  edges are added to the interference graph  $G$

Adding so many edges is not necessary though:

**IF**  $k$  names  $N_i$  ( $1 \leq i \leq k$ ) are live at the definition point of another name  $N'$  **THEN** we add the  $k$  interferences  $(N_i, N')$  to the graph.

so the actual notion of interference is: *two names interfere if one of them is live at a definition point of the other*. This is a better definition for 2 reasons: first, building the interference graph is less work now (we only add  $k$  edges instead of  $k(k-1)/2$ ) and second, there exist programs for which the resulting interference graph has a smaller chromatic number<sup>5</sup>.

The notion of interference as it is defined here indeed makes it possible to handle machine idiosyncracies with respect to register allocation in a uniform way: e.g. if register `R0` is not allowed to be the base register in a `load` instruction, just have all names that are used as a `load` base

<sup>5</sup>The chromatic number  $\chi$  of a graph is the minimal number of colors needed to color that graph.

register interfere with the node that corresponds to `R0`. Also, if, as a side effect, a call destroys the contents of certain machine registers, make all names that are live across the call interfere with all registers whose contents are destroyed <sup>6</sup>.

**Processing the interference graph** Processing the interference graph happens in 3 stages:

1. build the interference graph  $G$  cfr. supra
2. coalesce the nodes in  $G$  (i.e. force some nodes to have the same color and hence be assigned the same register later on) to create a new interference graph  $G'$
3. attempt to construct an  $n$ -coloring of  $G'$

The final step could be done using backtracking to find an  $n$ -coloring if one exists, but this might result in using exponential amounts of time so it is best to limit this to a fixed (and small) number of iterations.

**Node coalescing** Node coalescing (which is sometimes also called *subsumption*) is the process of taking 2 nodes  $n_i$  and  $n_j$  that don't interfere and replacing them by 1 node  $n_k$  that interferes with all nodes  $n_i$  and  $n_j$  interfered with. E.g. register subsumption may benefit from this: a `move R0,R0` instruction can be omitted from the code, so if a `move Rx,Ry` <sup>7</sup> instruction is present, we may try to coalesce the nodes corresponding to `Rx` and `Ry` and eliminate the instruction. However, this (again) requires the definition of interference to be altered/refined: the target of the `move` instruction doesn't necessarily have to be allocated to a different register than it's source. Therefore, a `move Rx,Ry` instruction at a point where `Rx` and  $k$  names  $N_i$  are live only yields  $k$  interferences (i.e. edges in  $G$ ) of the form  $(T, N_i)$ : an edge  $(T, S)$  is not added to  $G$ . Coalescing makes it possible to reduce the coloring of a graph  $G$  to the coloring of a simpler graph  $G'$ .

**Representation of the interference graph** Three operations are defined on the interference graph:

1. building the graph
2. coalescing nodes
3. coloring the graph

This implies that we need to access the nodes in the graph both randomly (determine whether 2 names interfere) and sequentially (go through a list of names that interfere with a given name). Operation 1 needs random access, operation 3 needs sequential access and operation 2 needs both. To satisfy both needs, 2 datastructures are typically used to represent the graph. Random access is very well supported by a *bit matrix* <sup>8</sup> while sequential access is very easy when *adjacency lists* <sup>9</sup> are used.

---

<sup>6</sup>The terms *register*, *node* and *name* are used loosely in this context to avoid the overuse of "corresponding to".

<sup>7</sup>`Ry` is the target register, `Rx` is the source register

<sup>8</sup>A bit matrix for a graph with  $n$  nodes  $N_i (1 \leq i \leq n)$  is an  $N \times N$  matrix  $M$  where  $M(i, j) = 1 \Leftrightarrow$  nodes  $N_i$  and  $N_j$  interfere and  $M(i, j) = 0 \Leftrightarrow$  nodes  $N_i$  and  $N_j$  do not interfere.

<sup>9</sup>for every node there is a linked list of all the nodes it interferes with.

**Spilling** Let  $\chi$  be the *chromatic number* of  $G$ . If  $\chi > n$  then the graph can't be colored with  $n$  colors. With respect to register allocation, this means that it is not possible to generate code for the given program using only  $n$  registers while every name is allocated to a register: some names will not be allocated to a register and for these, `store` and `load` instructions will have to be added to the code. The refusal of allocating a register to a name is referred to as *spilling* that name. The added `store` and `load` instructions are called *spill code*. When spilling is needed, the major issue to be resolved is deciding which name should be spilled. The heuristic used in [17] uses the concept of *register pressure*: the pressure on the registers at a point  $U$  in the program  $P$  = the number of live names at  $U$  + the number of machine registers unavailable at  $U$ .

Adding spill code should decrease register pressure. Two rules are given according to which spill code is added:

**Rule 1** if a name is spilled, insert a `store` at each of its definition points

**Rule 2** pass-throughs<sup>10</sup> are reloaded by putting a `load` at the entry of each basic block  $B$  for every name that is live at the entry to  $B$  and is not spilled within  $B$ , but is spilled in some basic block that is an immediate predecessor of  $B$ .

These simple rules sometimes insert unnecessary spill code, but this is later on eliminated by a dead code elimination process.

Another interesting idea is *rematerialization* which uses recomputation as an alternative to the insertion of spill code: `load` instructions that can't be coalesced can be replaced by a recomputation that directly leaves the result of the computation in the desired register. The main rationale behind this is that sometimes it's cheaper to redo a computation than to store its value in memory and retrieve it again when needed.

**An example** Figure 4 shows how Chaitin's technique basically works. No spilling is needed in this example and a coloring with 3 colors (which is a minimal coloring) is found. In the figure, the target number of registers (colors) is represented by  $k$  with  $k = 3$ . In the coloring step (represented in the right part of the picture) the 3 colors are indicated by the labels `c1`, `c2` and `c3` next to the nodes. Figure 5 illustrates that the technique can not handle all graphs: although in practice the coloring results are very good, even some simple graphs can not be colored in an optimal way.

### 2.2.2 Chow and Hennessy's priority based graph coloring algorithm

In [18], [19] another approach to graph coloring is presented, based on the observation that a good allocation should use a cost/benefit analysis, hence the name *priority based*. During the allocation process, the *unconstrained live ranges* are never considered: live ranges that interfere with less other live ranges than there are registers will ultimately always be colorable. Thus, the algorithm needs only to be concerned with *constrained live ranges*. The 3 steps of the algorithm are repeatedly applied, coloring one live range with each step, until either there are no more uncolorable live ranges left or there's no color left that can be assigned to a still uncolored live range:

1. calculate the priority function  $P(lr) = \frac{S(lr)}{N}$  (with  $lr$  an unconstrained live range) if it was not calculated before. In this definition of  $P(lr)$ ,  $S(lr)$  is defined as follows:  $S(lr) = \sum_{i \in lr} s_i \times w_i$  and  $s_i = LODSAVE \times u + STRSAVE \times d + MOV COST \times n$  with  $i$  one of the  $N$  live range

---

<sup>10</sup>A *pass through* is a computation which is live at the entry to a code interval but which is neither *used* nor *defined* within it.

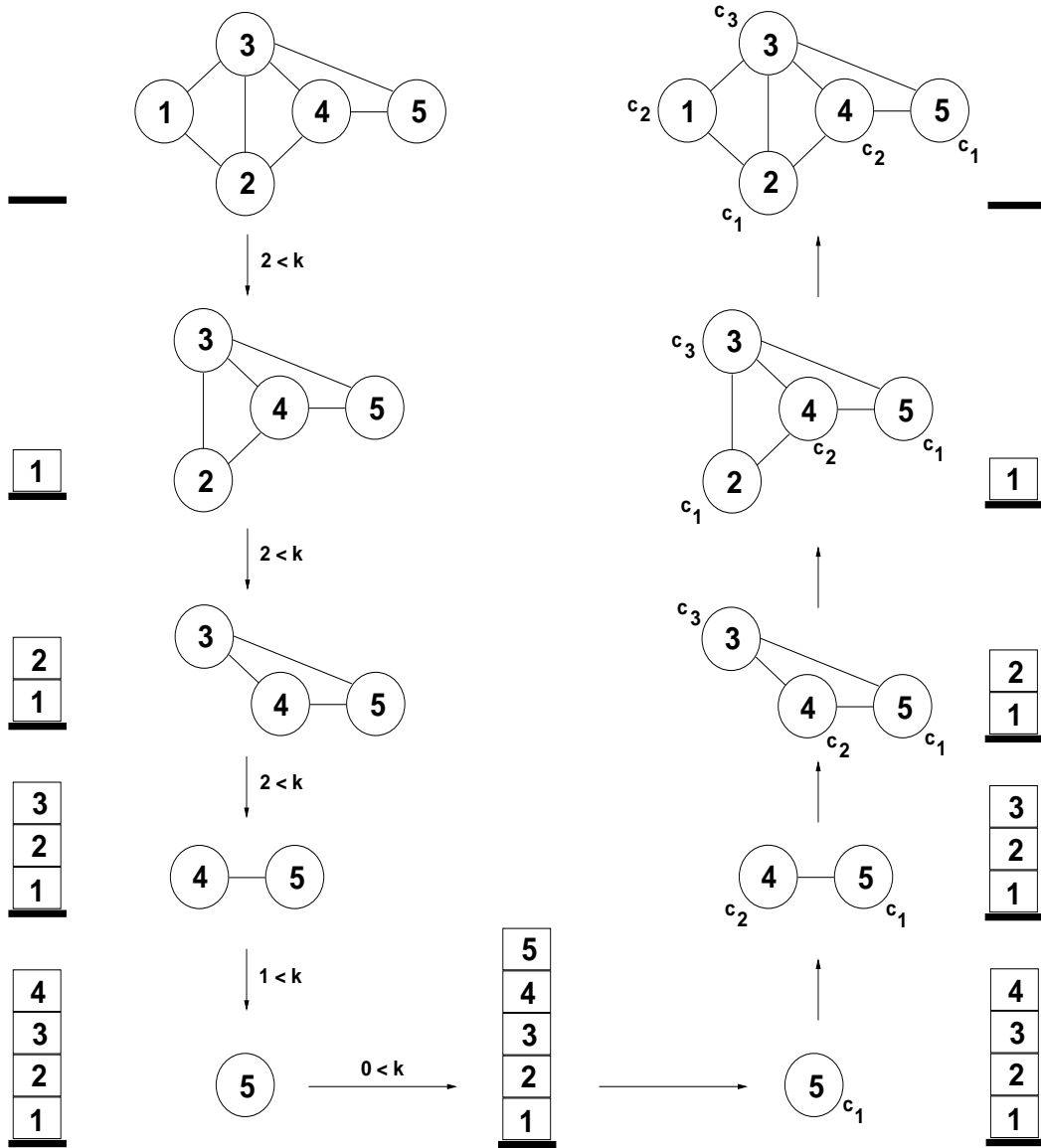


Figure 4: Expression graph for  $a/(b+c) - d*(e+f)$

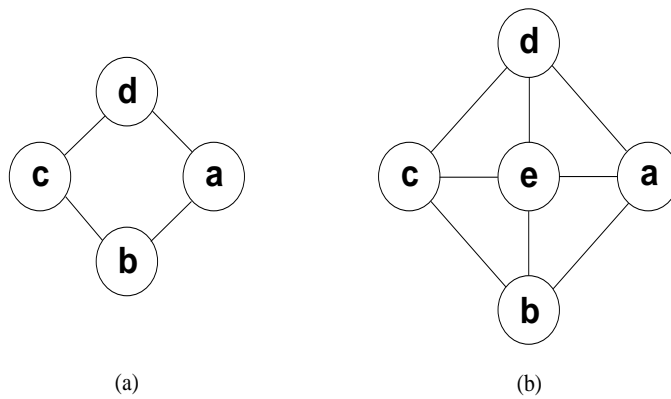


Figure 5: Graphs that can be problematic for Chaitin's heuristic

units of  $lr$ ,  $u$  the number of uses of  $i$ ,  $d$  the number of defines of  $i$ ,  $n$  the number of register moves needed and *LODSAVE*, *STRSAVE* and *MOV COST* the saves/costs in terms of cpu cycles. If  $P(lr) < 0$  or  $lr$  is uncolorable, mark  $lr$  as a variable that will not reside in a register.  $lr$  is then deleted from the conflict graph.

2. Find  $lr^*$  being the live range with the highest priority function value. Assign a valid color to  $lr^*$ .
3. For each live range  $lr$  interfering with  $lr^*$ , check whether it needs to be split or not.  $lr$  must be split if all valid colors for  $lr$  are already assigned to one or more of its neighbours.

### 2.2.3 Improvements and variations on the graph coloring approach

**node coalescing** Graph coloring removes copies by coalescing nodes: if the source and target of a copy instruction do not interfere in the conflict graph, these nodes can be treated as one single node. The coalescing heuristic in [17], [16] could make a graph uncolorable: every pair of nodes that is not connected by an edge in the interference graph is coalesced. Unfortunately, the new node being introduced is often so constrained that coloring of the resulting graph after coalescing is no longer possible, hence introducing spill code. [13] improves upon that heuristic by introducing a conservative approach that preserves graph colorability: if the node being coalesced has fewer than  $n$  neighbours with degree  $\geq n$  then coalescing is guaranteed not to turn a  $n$ -colorable graph into a non- $n$ -colorable one. However, this approach is too conservative and leaves many opportunities for coalescing open. [26] introduces a more aggressive approach since the graph is still  $n$ -colorable when a node has more than  $n$  neighbours with degree  $\geq n$ : 2 or more neighbours may have the same color. The idea is to interleave the simplification heuristic from [17] and [16] with the conservative approach of [13]. The *simplify* and *coalesce* routines of the Yorktown allocator scheme are called in a loop, hence the name *iterated coalescing*. The authors show that over 60% of all moves are removed by their coalescing scheme. [54] introduces a technique called *optimistic coalescing* which is based upon the *optimistic coloring* approach from [10]: coalesce nodes and when the coalescing decision turns out to be troublesome during coloring, split up the coalesced node into the set of original nodes again. This technique reduces the overall spill cost since nodes have a higher chance of being colored.

**live range splitting** The basic algorithm assigns a variable to a register for that variable's entire lifetime. By splitting the variable's live range into 2 or more parts (and hence creating 2 or more variables out of the given one), some of these parts may be assigned a register while others may not. Possibly, all parts may be assigned a register, introducing only some register moves (which are cheap in terms of cpu cycle usage) as overhead. Live range splitting was described in e.g. [20], [18] and [37].

**graph splitting** [28] describes an approach that partitions the interference graph into subgraphs that are colored individually and later merged. Separation of the graph happens along *maximal cliques*. A maximal clique is the the maximal graph for which every node is connected directly to every other node. Cliques are trivially colored: the required number of colors is the size of the clique, i.e. its number of nodes.

**rematerialization** In an attempt to improve the spill code, a technique called *rematerialization* was suggested by [17]. In [12] and [13] this is extended and handled more in depth. *Rematerializa-*

tion has to do with the observation that, when choosing the least expensive way to accomplish a spill, it is sometimes cheaper to do a recomputation of the value than to store it in main memory and load it afterwards. Rematerialization is not just one technique but a set of techniques that accomplish the same goal mentioned above. These are some practical opportunities for rematerialization:

- immediate loads of integer<sup>11</sup> constants
- computing a constant offset from the frame pointer or the static data-area pointer
- loads from a known constant location in either the frame area or the static data-area
- loading non-local frame pointers from a display<sup>12</sup>

The key to recomputation is that it should be possible to recompute the desired value in a cheap way from values and/or operands available throughout the procedure. While the approach in [17], [16] can only handle live ranges with a single value, [13] introduces a technique to handle live ranges with multiple values<sup>13</sup> by splitting each live range into its component values (by constructing each procedure’s static single assignment (SSA) graph), tagging each value with rematerialization information and finally form new live ranges from connected values that have identical tags (after some form of constant propagation has been used to propagate the rematerialization tags).

**optimistic coloring** In [10] and [13] a modification to the coloring algorithm of [17] is introduced, which is called *optimistic coloring*. Contrary to the approach in [17], nodes that are removed from the conflict graph and are assumed uncolorable are not spilled at once. Instead, they are put on the coloring stack just as the nodes for which coloring is possible. The rationale behind this is that some of the neighbours of this node may end up having the same color, making the over-constrained node colorable anyway and avoiding a needless spill code insertion. This optimistic approach is illustrated on an example that can not be colored successfully by [17] in figure 6. An early version of such an optimistic variation on the standard coloring algorithm was already presented in [45], but that algorithm only finds a coloring: there is no notion of finding a  $k$ -coloring for a given  $k$  and no mechanism for inserting spill code.

**Coloring register pairs** Some architectures require e.g. double-precision floating point values to be kept in adjacent (and sometimes also aligned<sup>14</sup>) register pairs. [17] suggests that such a machine idiosyncrasy can be handled by the graph coloring scheme it presents. However, in [11] it is noted that the standard coloring heuristic in [17] often over-estimates the register demand, resulting in more spilling than required. Limited ways to handle this problem were described in [30] (introducing a method to handle the register pair constraints that arise in the shift instructions on the ROMP microprocessor) and [50] (a method for allocating structures into an aggregate set of adjacent registers that produces good results when combined with the allocator described in [10]). In [11] it is shown that the optimistic coloring approach naturally avoids the over-spilling problem: a node is only spilled when no register pair is available when selecting a color for the node under consideration.

---

<sup>11</sup>or, on some machines, floating point

<sup>12</sup>the list of stack frame pointers the CPU copies into the new stack frame from the preceding frame is sometimes called the display. The first word of the display is a pointer to the last stack frame.

<sup>13</sup>i.e. a live range with multiple definitions of the same variable: there is a clear distinction to be made between a *value* and a *live range*

<sup>14</sup>the first register of an adjacent pair of registers should have an even number

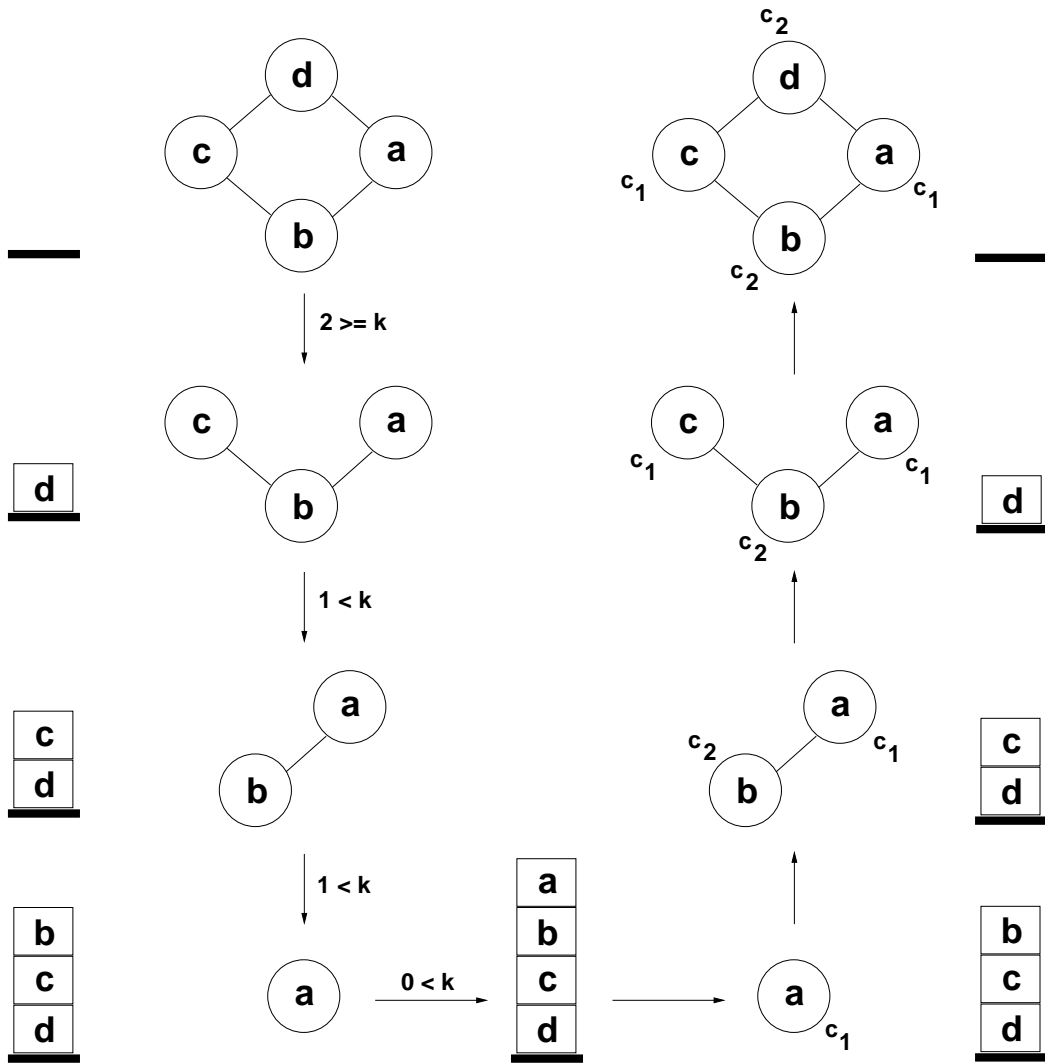


Figure 6: Brigg's heuristic applied to a graph Chaitin's heuristic can't color with 2 colors

## 2.3 Register allocation at the program level

Contrary to other register allocation methods, global methods take the control flow of the target program into account.

### 2.3.1 Register allocation by graph fusion

Every Yorktown allocator variant treats live range splitting, live range spilling and register assignment during different phases of the process and does this repeatedly since it uses procedures as its objects to work on, not considering what happens to live ranges over procedural boundaries. By building interference graphs for all regions (a region can be any combination of basic blocks) and then fusing these graphs into one global interference graph, a better allocation can be achieved, especially since either static estimation or profiling is used to guide the decisions. This guidance can help integrate register allocation into the total optimisation framework: it is now known where the most critical/optimized parts of the program are, so register allocation may be treated differently there to get better performance, not letting decisions from less important parts spoil the decisions in the critical part of the code. The idea is to delay coloring of conflict graphs of a procedure until more of the program is known, i.e. until the global conflict graph has been constructed. This is exploited in [44], the idea of *regions* was introduced in [29]. In the *graph fusion* approach the partial conflict graphs are merged along control flow edges by means of a *fusion operator*. This fusion operator maintains the invariant that the resulting graph is simplifiable (i.e. colorable) if the merged graphs are simplifiable also. Spilling is delayed until this merging phase, where gradually more live ranges are spilled as more graphs are merged and hence more information becomes available.

### 2.3.2 Demand driven register allocation

In this technique illustrated in [57], graph coloring is used to allocate registers but not to assign them, i.e. graph coloring decides which values get assigned a register but not which register specifically. To decide which values get assigned a register, some measurement is done regarding the costs and the benefits of that decision, as well as the estimate that, given an instruction and a variable that is assigned a register before that instruction, that variable will still be in a register after the instruction was executed. Local allocation is done first, since restricting the decision to a small part of the program enables a rigorous search for the optimal allocation. Afterwards, global allocation is done starting from the deepest nested loops and using both the cost/benefit and the estimate heuristics as a guide. The technique is promising but when very few registers are available, it can not be shown that the approach consistently beats the classic approach in [17]. An approach that uses a similar sequence of phases, is described in [38]. That approach produces code that is 25% slower compared to Chaitin-style allocation but the allocation process itself is almost 2.5 times faster. After the greedy local allocation phase, register assignment is done globally using a priority function: based on their use frequency as well as their place in the program, the candidates that were selected to be kept in a register in phase one are now assigned a specific register.

### 2.3.3 Linear scan register allocation

An intraprocedural linear register allocation algorithm was proposed by [15]. Having a linear register allocator is interesting for compiler response issues, since typically graph coloring has a  $O(n^2)$  complexity with  $n$  the number of live ranges due to the construction of successive graphs. Linear scan register allocators have been introduced by both [56] and [63] (based on the binpacking algorithm described in [9]: registers are viewed as bins into which temporary life ranges are packed,

taking into account the restriction that a bin can only contain one live range at a time) that are interprocedural. Although they do an only slightly worse allocating job than the register coloring variants, they are much faster as soon as  $n$  starts growing. A Linear scan allocator views liveness as a *lifetime interval* and visits each lifetime interval in turn (according to its occurrence in the static linear code order). It considers how many other lifetime intervals are currently live also: that number represents the register pressure at that point, i.e. the need for registers. If the register pressure is bigger than the amount of registers available, a heuristic spills one (or more, if needed) of them to memory.

### 2.3.4 Register allocation at link-time

As said, global program optimization is concerned with optimizing the entire program, across procedure boundaries. Large programs usually have their procedures structured in a collection of several modules and libraries that are linked together to create an executable. If the source code of every module and library is available to the compiler, optimizing an entire program can be done with (extensions of) the usual techniques. However, the source code is not always available (commercial libraries are typically already compiled when they're delivered) or different modules might be written in different languages. The first time all the information available comes together is at link-time. Several link-time optimizers exist (e.g. [61], [49]) and their results show that having another optimization pass at link-time is very beneficial. Register allocation during this phase is also interesting, since calls from a procedure in one module to a procedure in another module prohibit a good allocation if both modules are not available to the compiler at the same time. If the compiler has to do register allocation for separate modules, two registers might use different registers for the same global, or the same register for different locals. Both of these problems disappear if registers can be assigned at link-time over the entire program. In [64] a technique is presented that uses compiler annotations as well as profile information to achieve this goal. The annotations the compiler makes allow for register allocation to be treated as a form of relocation: the linker rewrites each module based on the allocation. Given are the following annotations (for a three-operand instruction set):

- **REMOVE.name**: remove the annotated instruction if *name* is assigned to a register
- **OP1.name**: replace the first operand by the register *name* is assigned to
- **OP2.name**: replace the second operand by the register *name* is assigned to
- **RESULT.name**: replace the result operand by the register *name* is assigned to
- **LOAD.name**: replace the `load` instruction of this annotation by a `move` instruction that copies *name* from the register it's currently assigned to, to a temporary register
- **STORE.name**: replace the `store` instruction of this annotation by a `move` instruction that copies *name* from the temporary register it's currently in, to the register it was assigned to

The first 4 annotations are straightforward, the latter 2 are somewhat more complicated. An example with the code generated for the C assignment  $x = y++ + z$  illustrates the use of *LOAD.name* (as well as that of some other annotations): it is needed when the value of that name changes, while its original value is used further in the code.

R1 := load y	LOAD.y
R2 := R1 + 1	RESULT.y
store y := 2	REMOVE.y
R2 := load z	REMOVE.z
R3 := R1 + R2	OP2.z RESULT.x
store x := R3	REMOVE.x

which, if  $x$ ,  $y$  and  $z$  are all assigned to registers, becomes:

```

R1 := y
y := R1 + 1
x := R1 + z

```

How is the code annotated? The code is seen as a sequence of commands. Each command is one of three types:

1. **leaf**: evaluate a single variable
2. **assign**: assign the value of a previous command to a variable
3. **operation**: perform an operation using the values of previous commands and produce a new result value

Some commands must be marked as time critical ([62] presents an algorithm for doing so): the value used there is not always the current value of the variable. Once this marking is done, code can be annotated as follows:

**Case 1:** if the command is a *leaf* for some variable  $v$  **then** generate a **load** into a temporary register <sup>15</sup>. **If** the leaf is marked *time critical* **then** annotate the instruction with **LOAD.v** **else** annotate it with **REMOVE.v**.

**Case 2:** if the command is a *operation*, generate the instructions to perform it (either 1 instruction or a sequence of instructions). **If** the result value of this operation is only used once (by an assignment command  $v :=$ , **then** annotate the instruction with **REMOVE.v**.

**Case 3:** if the command is an *assignment* to some variable  $v$ , generate a **store**. **If** the operand is a leaf command **or** if the operand is used in another command also, **then** annotate the instruction with **STORE.v** **else** annotate it with **REMOVE.v**.

For each instruction, if an operand is a leaf for some variable  $v$  and it is not marked as *time critical*, annotate all instructions that use that operand with either **OP1.v** or **OP2.v** (whether it is the first or the second operand).

This technique gives remarkably good results: it does register allocation substantially better than the known compile-time algorithms.

### 3 Register allocation and its relation to other optimization techniques

Register allocation is just one optimization technique within a large set of others that are applied one after the other. These techniques are not always independent, so the influence of other techniques on register allocation is an important research topic. For an overview of most of these techniques, excellent references are [2] and, more recently, [48].

---

<sup>15</sup>Some registers are reserved for temporary use and as such can not be used for holding variables.

## 3.1 Instruction Scheduling

Instruction scheduling is concerned with the reordering of program instructions to improve performance. This reordering can e.g. enhance ILP (instruction level parallelism) or reduce the amount of conflicts in the pipeline.

### 3.1.1 A brief introduction to instruction scheduling

The most basic instruction scheduling techniques are *branch scheduling* and *list scheduling* which will be presented here as a brief introduction to the field.

**branch scheduling** Branch scheduling incorporates 2 related aspects:

- filling the *delay slot(s)* after a branch instruction with useful instructions
- covering the delay between performing a compare and being able to branch based on its result

Many architectures have delayed branch or call instructions, i.e. one or more slots after the branch or call instruction can contain instructions that are executed while the branch/call is handled. This way, pipeline stalls are avoided because of the typical longer execution time taken by calls or branches. As whether a branch is taken or not often depends on a condition, the instructions in the delay slots are not always executed: the concept of *nullification* makes sure they are only effectively executed when the branch is taken and are not executed otherwise. Jumps, calls and branch always instructions (may) have delay slots that are always executed. Branch scheduling tries to fit appropriate instructions into these delay slots to maximize program performance.

Some machines require some cycles to have passed between a condition-determining instruction and a branch instruction that depends on the outcome of the condition. If the required time has not passed by the time the branch is executed, the processor stalls at the branch instruction for the remainder of the number of cycles that must pass. By placing the compare as early in the schedule as possible, the potential number of cycles wasted can be minimized.

**list scheduling** List scheduling is a *basic block scheduling* technique, i.e. its goals are to reorder the instructions within a basic block in such a way that the execution time of the basic block is minimal while it still produces the same result as originally intended. The instructions in the basic block under consideration are put into a DAG (direct acyclic graph). List scheduling initially traverses this DAG from the leaves towards the roots, labeling each node with the maximum possible delay from that node to the end of the basic block. In a second step, the DAG is traversed from the roots towards the leaves while selecting nodes to schedule and keeping track of the current time and the earliest time each node should be scheduled to avoid stalls. Since the problem is NP-complete, heuristics are applied to select the best instruction from the set of candidates for each time slot.

### 3.1.2 Integrating register allocation and instruction scheduling

Register allocation and instruction scheduling are closely related since they both have an influence on each other's results if one is done after the other. If register allocation is done first, instruction scheduling becomes a bit more constrained: you can not (always) reschedule instructions in such a way that previously non-overlapping live ranges suddenly overlap, because they may be assigned the same register. Vice versa, if instruction scheduling is done first, register allocation has a harder job to tackle as there may be more overlap between live ranges. As the question which phase

must be done before the other has no answer, a lot of attempts have been made to integrate the 2 consecutive phases into 1 single phase that handles both problems or at least to make the first phase aware of the one to follow it.

In [6] URSA, the *Unified Resource Allocator*, is presented which allocates both functional units as well as registers to a VLIW (*Very Large Instruction Word*) architecture. Aware of the phase ordering problems described above, this approach comes up with a new set of phases where each phase has a minimal impact on the subsequent one. These phases are (1) the computation of resource requirements, (2) the identification of regions with excessive requirements, (3) code motion to reduce the (excessive) requirements and (4) resource assignment. In [7] and [8] this framework is described in more detail with respect to instruction scheduling vs. register allocation. In [5] URSA has evolved into GURRR, the *Global Unified Resource Requirements Representation* where the program dependence graph is augmented with resource requirement information.

[51] avoids the increasing complexity of a full integration of the 2 phases by modifying the global register allocation phase, which comes first, to become *scheduler sensitive*. Since scheduling comes after register allocation, the inserted spill code will also be scheduled nicely. [52], by the same authors, inverts the order of that approach, making the scheduler *register allocation sensitive*. This way, it is hoped that the amount of spill code to be inserted will be minimal. The authors describe their further experiences with and conclusions from these schemes in [53].

In [55] register allocation is done based on the coloring of the *parallellizable interference graph*. This representation ensures no false dependencies are introduced and hence all the options for parallellism remain available to the scheduler. Heuristics are introduced to make a trade-off between the costs of register spilling versus the loss of instruction level parallellism.

[47] shows that, for a basic block, the integration of register allocation and instruction scheduling into one problem (*CRISP: Combined Register Allocation and Instruction Scheduling*) is solved easier *approximately* than when using the classic graph coloring approach. An algorithm based on a heuristic is introduced, that would have been impossible to formulate outside of such a full integration. The algorithm is called the  $(\alpha, \beta)$  – *Combined Heuristic* where  $\alpha$  is a measure of the register pressure and  $\beta$  is a measure of the instruction level parallellism.

### 3.1.3 Register allocation for loops

Loops are considered a special case with respect to register allocation since live ranges may be spread over different, successive iterations of the loop. Since loops are often those parts of the program where most of the time is spent, both register allocation as well as instruction scheduling for loops are critical.

**software pipelining** This is a loop scheduling technique that overlaps the execution of several consecutive iterations of the loop in an attempt to enhance parallellism in the loop body: instructions can be scheduled in such a way that instruction level parallellism is maximal. Obviously, some instructions have to be set up before and after the rescheduled loop body to ensure correctness. Unfortunately, software pipelining often increases register pressure, making register allocation even harder or less succesful (see [43], [41], [42], [65]).

**the Meeting graph** In [22], [23], [21] and [39] the *meeting graph* is introduced as an alternative for the conflict graph, which can be a circular-arc graph for loops. Two live ranges “meet” if one of them ends at the time the other one starts. Figure 7 illustrates the difference between the conflict graph and the meeting graph of a simple example. The labels next to the nodes of the meeting

graph are the length of the live ranges they represent, expressed as the number of cycles. The meeting graph incorporates a notion of *time* which is not in a normal conflict graph and which is obviously needed when treating a loop body as live ranges can span more than one iteration of the loop.

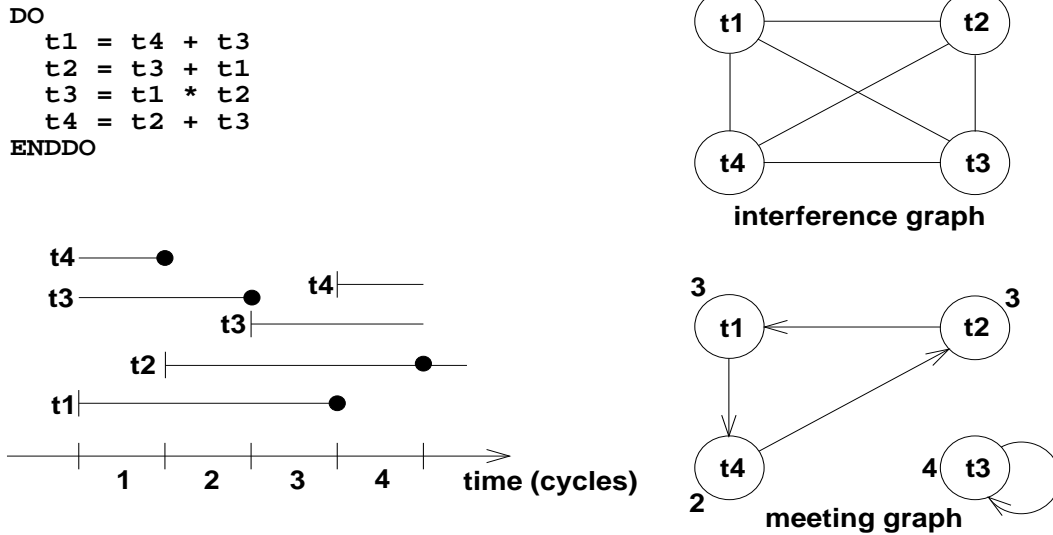


Figure 7: Both the conflict graph and the meeting graph for a small example

### 3.2 Register allocation vs register windows

While the assignment of variables to registers in register allocation is handled by the compiler and is thus the responsibility of the software, there also exists a hardware mechanism that tries to achieve the same goal. If the hardware makes use of *register windows*, the register set (which is typically larger than the register set when there is no hardware support, i.e. when register allocation is used) is a circular register buffer. If a call occurs, the tail pointer is moved to allow a number (a *window*) of registers (either a fixed number or a number dependent on what is needed, this varies according to different architectures) to be used for the locals in the called procedure. If an overflow occurs, enough registers are spilled from the head of the buffer to create a new window at the tail. In [62] a study is presented that compares both approaches by measuring how well either register management scheme removes `load` and `store` instructions. Their actual measure is  $MR = (M_1 + S)/M_0$  where

- $M_0$  is the number of `load` and `store` instructions used to move variables between main memory and the register set when no register management scheme is used.
- $M_1$  is the number of `load` and `store` instructions used for these variables when a register management scheme is in use.
- $S$  is the number of `load` and `store` instructions that were added due to the use of the register management scheme.

Note that  $MR = 0$  is optimal while  $MR > 1$  means the applied register management scheme actually made things worse. Also, some more traditional schemes are included in the tests.

Their conclusion is that *register allocation at link – time* (if combined with profiling information) outperforms all the other software based methods and was sometimes even better than the *register windows* approach. *Register windows* actually does a slightly better job in allocating, but since the method is hardware based this results in the need of sacrificing some chip real estate (for the extra registers) and a slight increase of the cycle time (because of the overhead). The article also mentions the difference between *cooperative register allocation* and *selfish register allocation*. In the former, each procedure sometimes doesn't get all the registers it needs for its local variables, but procedures in the same call chain use different registers so spills and reloads are only needed for recursive calls or indirect calls (through procedure variables). In the latter, registers are allocated for the procedures separately so each procedure is able to use all the registers. Unfortunately, this means that registers used by the procedure must be spilled at its entry and must be reloaded when exiting the procedure.

## 4 Optimal register allocation?

In spite of the NP-completeness of the register allocation problem, several attempts to achieve an optimal solution within a reasonable time have been made. Among the approaches used are *dynamic programming* ([46], focusing on optimal instruction scheduling but integrating some aspects of register allocation into that framework) and *integer linear programming*<sup>16</sup> ([4], [27], [31]). Unfortunately, all of them are painstakingly slow. Even the most recent attempt in [4], where optimal register allocation is split into 2 phases (optimal spill code placement followed by optimal coalescing) is not satisfyingly fast: an efficient algorithm for the former phase is presented, an efficient algorithm for the latter phase is left as an open question (although both an inefficient optimal algorithm as well as an efficient suboptimal one are given).

## References

- [1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, USA, 1974.
- [2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [3] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19:137–147, 1976.
- [4] A. W. Appel and L. George. Optimal spilling for cisc machines with few registers. Technical Report TR-630-00, Princeton University, November 2000. To appear in ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation , June 2001.
- [5] D. Berson, R. Gupta, and M. Soffa. Gurr: A global unified resource requirements representation. In *ACM SIGPLAN Workshop on Intermediate Representations, Sigplan Notices, Vol. 30*, pages 23–34, April 1995.
- [6] D. A. Berson, R. Gupta, and M. L. Soffa. Ursa: A unified resource allocator for registers and functional units in vliw architectures. In *Proceedings of the IFIP Working Conference on*

---

<sup>16</sup>which is linear programming where all coefficients are integers

*Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, Orlando, Florida, January 1993.*

- [7] D. A. Berson, R. Gupta, and M. L. Soffa. Resource spackling: A framework for integrating register allocation in local and global schedulers. In *PACT '94: International Conference on Parallel Architectures and Compilation Techniques, Montreal, Canada, August 1994.*
- [8] D. A. Berson, R. Gupta, and M. L. Soffa. Integrated instruction scheduling and register allocation techniques. In *Languages and Compilers for Parallel Computing*, pages 247–262, 1998.
- [9] D. S. Blickstein, P. W. Craig, C. S. Davidson, R. N. Faiman, K. D. Glossop, S. O. Hobbs, and W. B. Noyce. The gem optimizing compiler system. Technical Report 4, Digital, 1992.
- [10] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the SIGPLAN 89 Conference on Programming Language Design and Implementation, SIGPLAN Notices 24(7)*, pages 275–284, July 1989.
- [11] P. Briggs, K. D. Cooper, and L. Torczon. Coloring register pairs. *ACM Letters on Programming Languages and Systems, (LOPLAS)*, 1(1):3–13, March 1992.
- [12] P. Briggs, K. D. Cooper, and L. Torczon. Rematerialization. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pages 311–321, San Francisco, California, 17–19 June 1992. *SIGPLAN Notices 27(7)*, July 1992.
- [13] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, May 1994.
- [14] J. Bruno and R. Sethi. Code generation for a one-register machine. *J. ACM*, 23(3):502–510, July 1976.
- [15] R. G. Burger, O. Waddell, and R. K. Dybvig. Register allocation using lazy saves, eager restores, and greedy shuffling. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 130–138, June 1995.
- [16] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction*, pages 98–105, Austin, Texas, 30 Apr.–2 May 1982.
- [17] G. J. Chaitin, M. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.
- [18] F. C. Chow and J. L. Hennessy. The priority based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, 1984.
- [19] F. C. Chow and J. L. Hennessy. Register allocation by priority-based coloring. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 222–232, June 1984.
- [20] R. Cytron and J. Ferrante. What's in a name? the value of renaming for parallelism detection and storage allocation. In *Proceedings of the 1997 International Conference on Parallel Processing*, pages 19–27, August 1987.

- [21] C. Eisenbeis and S. Lelait. Lora: a package for loop optimal register allocation. In *Proceedings of the 3rd International Workshop on Code Generation for Embedded Processors, Witten, Germany*, March 1998.
- [22] C. Eisenbeis, S. Lelait, and B. Marmol. The meeting graph: a new model for loop cyclic register allocation. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT'95, Limassol, Cyprus*, June 1995.
- [23] C. Eisenbeis, S. Lelait, and B. Marmol. Circular-arc graph coloring and unrolling. In *Proceedings of the 5th Twente Workshop on Graphs and Combinatorial Optimization, Twente, The Netherlands*, pages 71–74, May 1997.
- [24] M. Farach and V. Liberatore. On local register allocation. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1998.
- [25] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., San Francisco, CA, 1979.
- [26] L. George and A. W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.
- [27] D. Goodwin and K. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software-Practice and Experience*, 26(8):929–965, 1996.
- [28] R. Gupta, M. Soffa, and T. Steele. Register allocation via clique separators. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 264–274, July 1989.
- [29] R. E. Hank, W. W. Hwu, and B. R. Rau. Region-based compilation: An introduction and motivation. *International Journal of Parallel Programming*, 25(2):113–146, April 1997.
- [30] M. E. Hopkins. Compiling for the rt pc romp. *IBM RT Personal Computer Technology*, pages 76–82, 1986.
- [31] D. Kästner and M. Langenbach. Integer linear programming vs. graph-based methods in code generation. Technical report, Saarland University, February 1998.
- [32] C. W. Kessler. Scheduling expression dags for minimal register need. In *Proceedings of the 8th Int. Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'96)*, pages 228–242, Aachen, Germany, September 1996. Springer.
- [33] C. W. Keßler. Scheduling Expression DAGs for Minimal Register Need. *Computer Languages*, 24(1):33–53, September 1998.
- [34] C. W. Keßler, W. J. Paul, and T. Rauber. A randomized heuristic approach to register allocation. In J. Maluszynski and M. Wirsing, editors, *Programming Language Implementation and Logic Programming: Proc. of the 3rd International Symposium PLILP'91, Passau*, pages 195–206, Berlin, Heidelberg, 1991. Springer.
- [35] C. W. Kessler and T. Rauber. Optimal contiguous expression dag evaluations. In *Proceedings of Int. Conf. on Fundamentals of Computation Theory (FCT'95), Dresden, Germany*. Springer, August 1995.

- [36] C. W. Keßler and T. Rauber. Generating optimal contiguous evaluations for expression DAGs. *Computer Languages*, 21(2):113–127, 1996.
- [37] P. Kolte and M. Harrold. Load/store range analysis for global register allocation. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 268–277, June 1993.
- [38] K. Krishna and S. M. Krishnamurthy. Register allocation sans coloring. Technical Report 94-04, University of New Mexico, Albuquerque, February 1994.
- [39] S. Lelait, G. R. Gao, and C. Eisenbeis. A new fast algorithm for optimal register allocation in modulo scheduled loops. In *Proceedings of the 1998 International Conference on Compiler Construction (CC'98), Lisbon, Portugal*, pages 204–218, April 1998.
- [40] V. Liberatore, M. Farach, and U. Kremer. Hardness and algorithms for local register allocation. Technical Report DCS-TR-332, Rutgers University, New Jersey, USA, July 1997.
- [41] J. Llosa, M. Valero, and E. Ayguadé. Bidirectional scheduling to minimize register requirements. In *Proceedings of the 5th International Workshop on Compilers for Parallel Computers, Malaga*, pages 534–554, June 1995.
- [42] J. Llosa, M. Valero, and E. Ayguadé. Heuristics for register-constrained software pipelining. In *Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29), Paris (France)*, pages 250–261, December 1996.
- [43] J. Llosa, M. Valero, E. Ayguadé, and J. Labarta. Register requirements of software pipelined loops and its effects on performance. In *Proceedings of the 2nd International Workshop on Massive Parallelism, Capri-Italy*, pages 173–189, October 1994.
- [44] G. Lueh, T. Gross, and A. Adl-Tabatabai. Global register allocation based on graph fusion. Technical Report CMU-CS-96-106, Carnegie Mellon University, Pittsburgh, USA, March 1996.
- [45] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering of graph coloring algorithms. *Journal of the ACM*, 30(3):417–427, July 1983.
- [46] W. M. Meleis and E. S. Davidson. Optimal local register allocation for a multiple-issue machine. In *Proceedings of the 8th conference on ACM International Conference on Supercomputing*, pages 107–116, July 1994.
- [47] R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen. Combining register allocation and instruction scheduling. Technical Report CS-TN-95-22, Stanford University, August 1995.
- [48] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman Publishers, 1997.
- [49] R. Muth, S. Debray, S. Watterson, and K. De Bosschere. `alto`: A link-time optimizer for the compaq alpha. *Software Practice and Experience*, 31:67–101, January 2001.
- [50] B. R. Nickerson. Graph coloring register allocation for processors with multi-register operands. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 40–52, June 1990.

- [51] C. Norris and L. Pollock. A scheduler-sensitive global register allocator. In *Proceedings of Supercomputing '93*, pages 804–813, 1993.
- [52] C. Norris and L. Pollock. Register allocation sensitive region scheduling. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, June 1995.
- [53] C. Norris and L. Pollock. Experiences with cooperating register allocation and instruction scheduling. *International Journal on Parallel Programming*, 26(3):241–284, September 1998.
- [54] J. Park and S.-M. Moon. Optimistic register coalescing. In *Parallel Architectures and Compilation Techniques (PACT '98)*, pages 196–204, 1998.
- [55] S. S. Pinter. Register allocation with instruction scheduling: a new approach. *ACM SIGPLAN Notices*, 28(6):248–257, 1993.
- [56] M. Poletto and V. Sarkar. Linear scan register allocation. *TOPLAS*, 21(5):895–913, Sept. 1999.
- [57] T. A. Proebsting and C. N. Fischer. Demand-driven register allocation. *ACM Transactions on Programming Languages and Systems*, 18(6):683–710, 1996.
- [58] J. T. Schwarz. On programming: An interim report on the setl project. Technical report, Courant Institute of Math. Sciences, New York University, 1973.
- [59] R. Sethi. Complete register allocation problems. *SIAM Journal of Computing*, 4(3):226–248, 1975.
- [60] R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *J. ACM*, 17(4):715–728, Oct. 1970.
- [61] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, pages 1–8, March 1993.
- [62] A. Srivastava and D. W. Wall. Link-time optimization of address calculation on a 64-bit architecture. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 49–60, June 1994.
- [63] O. Traub, G. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. *ACM SIGPLAN Notices*, 33(5):142–151, 1998.
- [64] D. W. Wall. Global register allocation at link time. In *Proceedings of the ACM SIGPLAN '86 Conference on Compiler Construction.*, pages 264–275, 1986.
- [65] J. Wang, A. Krall, M. Ertl, and C. Eisenbeis. Software pipelining with register allocation and spilling. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO27)*, pages 95–99, November 1994.
- [66] A. P. Yershov. *The Alpha Automatic Programming System*. Academic Press, London, 1971.