Intermediate Program Representations for Register Allocation

Patryk Zadarnowski

University of New South Wales
<patrykz@cse.unsw.edu.au>

Abstract. The recent advent of the load/store architectures, combined with the growing disproportion between the register and memory access times, has exerted an increasing pressure on compiler designers to expend substantial effort on the register allocation phase of compilation, promoting it from its original role as an optional optimizing pass to an intrinsic part of the compilation process and an important design factor in determining the global structure of a compiler. There is a strong tendency for modern compilers to be clearly divided into a portable symbolic phase encompassing the bulk of transformations, and an unportable register-allocated phase comprising of architecture specific transformations such as instruction scheduling and cache-related optimizations, and connected to the portable phase through register allocation. In this report, we review the past and present register allocation methodologies employed by compiler designers, with an emphasis on their influence on the design of intermediate program representations.

1 Introduction

Register allocation refers to the general problem of an effective utilization of the limited resources provided by the CPU register files for storage of the unbounded set of variables referenced in a program.

Originally, register allocation has been viewed as an optional optimizing transformation intended to reduce the number of instructions while possibly offering some limited performance improvements over the the canonical representation of the program in which every variable was allocated a fixed location in memory, usually in the stack frame of the current proceduredure. [3] However, with the advent of load/store architectures such as *Reduced Instruction Set Computing* (RISC) [31], this approach ceased to be feasible. There is a marked tendency in modern instruction set architectures to depricate or completely elimiate computational instructions on values stored in memory, forcing variables on the stack to be loaded into a register files before being operated on, and therefore requiring a pair of additional load and store instructions for every variable access. Under these conditions, executing a register-unallocated program could result in tripling of the code size and a potential ten-fold degradition of performance as the access times to main memory the CPU registers continue to diverge. [20]

Further, the large register files of modern processors have made it feasible to associate variables with registers by default and to fall back to the data stack only as a last-resort alternative. This, in turn has prompted compiler designers to focus development on optimizations designed to operated on a symbolic representations of the program which assume an infinite number of available registers, effectively dividing the compiler into symbolic and register-allocated phases, with register allocation promoted to the role of an intrinsic structural element linking the two stages of compilation. Today, we can indentify three clearly-separated stages in most modern compilers:

1. *Language-specific transformations*
   First, the compiler performs a number of transformations specific to the source language, usually operating on a high-level symbolic intermediate representation closely resembling the abstract syntax tree of the input program. These transformations include, for example deforestration and unboxing in compilers for lazy functional languages such as Haskell [23, 34, 11]
2. *Portable symbolic transformations*
   The bulk of optimizing transformations is typically performed on a quasi-portable low-level representation. These representations tend to vary a lot in format and the level of abstraction, but one commonality is that, by the end of the phase, the program is transformed into a normalized low-level representation in which all intermediate results of computation are assigned to named temporary variables. Typically, the final assembly instructions are selected at the end of this phase, and the only difference between that representation and the final assembly output is the presumption of availability of an infinite number of registers to hold the variables and temporaries of the program.
3. *Architecture specific transformations*
   In the final phase of compilation, a limited number of transformations such as instruction scheduling [28] and instruction-cache optimizations [27] are performed on the assembly representation of the program. Since these transformations are highly sensitive to the details of the target machine, they must be performed with a detailed knowledge of the registers selected for storage of variables during calculation, and therefore assume a representation of the program that offers little abstraction over the final machine language.

Ocassionally, compilers interleave transformations from the language-specific and portable categories, repeatadly translating the program between the high-level and low-level symbolic representations, but, with only few notable exceptions architecture-specific transformations are never mixed with portable passes. [29] This restriction is dictated by the sensitivity of transformations such as instruction scheduling to even a slightest change of the program: converting the program into a portable representation would almost certainly nullify any benefits previously gained through these transformations.

It is entirely possible to envisage a compiler design in which register allocation is performed after the code is transformed into the final representation (initially

with all variables allocated in memory) and this used to be the standard design decision in compilers implemented before the advent of RISC architectures. [26] This design is distinguished by the fact that the input to the register allocator is a sub-optimal register-allocated program, transformed by the allocator by shifting some variables from memory into registers, making the design a favourite for local or semi-local register allocators, in which register allocation is guided by the loop structure of the program.

With the advent of graph coloring techniques [9, 8], this approach has lost its appeal in favour of global register allocators, which, by design, allocate variables to registers, given an unallocated symbolic representation of the program. For that reason, global register allocators are best placed in the compiler structure as a connecting element between the second and third groups of transformations, transforming the program from its portable to the final representation. Because of their efficiency and effectiveness, global register allocators are now the standard approach in modern compilers.

The reminder of this report is structured as follows: in Section 1.1, we give a quick overview of the local register allocation approaches, followed by a brief introduction to the graph coloring approach to register allocation. In Section 2, we expand on the details of the graph coloring algorithms, and include a brief discussion of the recently-proposed optimal graph coloring algorithm. Section 3 discusses a number of key additional issues that must be handled by a register allocator. The techniques are brought together in Section 4, where we discuss register allocation as it applies to four major types of program representation, and its influence on these representation. Finally, we conclude the paper in Section 5, with a brief comparison of these representations.

1.1 Overview of Register Allocation

There are two predominant approaches to register allocation: *local approaches*, which rely on the structure of the program to supply heuristics for the choice of variables to be shifted from memory to registers in the most performance critical sections of a suboptimally-allocated program, and *global approaches*, which collect and analyze information on the interference between all symbolic variables in a procedure, and attempt to allocate all (or as many as possible) of these variables to registers. In the first approach, all variables are stored in memory by default, while in the second, all variables are assumed to be stored in registers, and shifted to memory only temporarily whenever no free register is available. [29]

In both approaches, register allocation can be separated into two stages:

1. **Register allocation**, which selects the set of variables to reside in registers at a particular point in the program, and
2. **Register assignment**, which selects physical registers to hold values of variables during evaluation.

As always, multi-stage algorithms introduce a phase ordering problem: information about the selection of physical registers obtained during register assignment may aid the register allocation phase. In the rest of this section, we give an overview of the two predominant approaches, and discuss the benefits of the global approach as it relates to resolving the above phase ordering problem.

1.2 Local Approaches

The local approaches, popular in compilers written before the advent of the graph coloring techniques, are concerned predominantly with the register allocation stage (selection of variables to be stored in registers.) Typically, some fixed number of registers is reserved to hold the most acive values in each inner loop, while always using registers to store values local to a basic block. This methodology has been pioneered by Lowry and Medlock [26] and implemented in their Fortran H compiler for the IBM-360 series machines. Its primary advantage is simplicity, and yet, the approach tends to provide reasonable results when combined with efficient heuristics for variable selection.

In the Lowry and Medlock approach, we commence the register allocation in each innermost loop of the procedure, and progress to the outer loops as long as some available registers remain unused.

For each loop, we typically collect usage count information for every variables, and pick the most frequently used variables for storage in registers for the duration of the loop.

Specifically, we begin by generating code in which all variables are stored in memory between basic blocks, but in which all operations within a basic block are performed on values stored in registers. A basic block may be split if it uses more variables than the number of available registers. Variables are loaded into register upon entry into the basic block and stored back in memory on exit.

For each loop in the program (beginning with the innermost loop) we then proceed to estimate the savings to be realized by moving each variable into a register for the duration of the loop. Specifically, we count one unit of credit to a variable for each use of that variable within the loop that is not preceeded by an assignment to the variable in the same block. In addition, we count two units of savings for each block containing an assignment to the variable, for which the variable is live on exit from the block. For each loop, we select the variables which offer the most savings for storage in registers. Note that, using this algorithm, most variables will still be loaded from memory on entry into the loop and stored back on exit.

Despite its simplicity, this algorithm tends to perform very well in practice, and is still of some interest in environments where the compilation speed is important, such as optimizing just-in-time compilers. However, in most compilers, the efficiency of the global techniques described in the remainder of the paper, and the importance of effective register allocation on modern RISC architectures, more than amply outweight the penalty in compilation speed, and therefore local techniques have now been all but forgotten by compiler designers.

1.3 Global Approaches — Register Allocation as Graph Coloring

While the local approach described above is clearly pessimistic in nature, assuming that a variable cannot be stored in a register unless proven otherwise, global approaches tend to be optimistic, focusing on the problem of sharing registers for storage of different variables. This approach relies on an observation that, while there is almost always more variables to allocate than the number of registers in the processor, the lifespans of these variables rarely overlap, and therefore, more often than not, multiple variables can share the same register in a procedure.

Specifically, two variables can be allocated to the same register precisely if their lifespans do not overlap. A lifespan of a variable extends from its first assignment to its last use. If we construct a graph in which each node corresponds to a variable, with an edge between two variables precisely when their lifespans overlap, we can reduce the problem of register allocation to the well-understood problem of *graph coloring.* [9, 8] Such a graph is called the *variable interference graph* of the program.

Graph coloring refers to the problem of coloring nodes in a graph with k distinct colors, such that no two adjacent nodes are assigned the same color in the graph. In general, graph coloring in known to be NP-hard, but in practice, a number of very efficient algorithms exist for finding good approximation to the solution in linear or quadratic time.

To refolmulate the register allocation problem as graph coloring, we take the variable interference graph for a procedure and attempt to colour its notes with k colours, where k is the number of available registers. If k-coloring cannot be achieved on the original procedure, we move (*spill*) some variables into memory for a portion of their lifespan, and attempt k-coloring on the modified graph. As will be demonstrated in the next section, it is possible to perform k-coloring incrementally in such a way as to avoid a complete recomputation of the graph after each spill.

One advantage of the graph coloring approach is that it is trivial to extend it to accomodate architectures with non-uniform register files. Occasionaly, processors place restrictions on registers which may be used as operands for certain instructions. For example, on most architectures, floating point operations are performed on dedicated FPU registers, and boolean values obtained from integer test instructions are stored in a limited number of condition flags with severe restrictions placed on their usage. The graph coloring approach can be extended to handle this scenarios by adding nodes for each physical register to the interference graph, and drawing edges between these registers and variables that cannot be stored in them.

Most register allocators used in practice are based on graph coloring. We devote the next section to finding an approximate solution to the graph coloring problem for a register interference graph, and delay discussion of alternative global techniques until Section 5 of this paper.

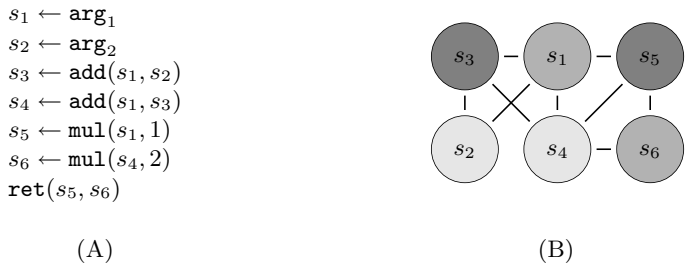


Fig. 1. Register allocation by graph coloring

2 Implementation of Graph Coloring Allocators

The connection between register allocation and graph coloring has been made by Cocke [4], and the first register allocator based on this result has been published by Chaitin [9] and implemented in the experimental IBM 370 PL/I compiler, soon followed by a number of papers proposing numerous improvements to the original approach, the most important being the introduction of *priority-based graph coloring* by Chow and Hennessy. [10]

To this day, virtually all successful graph coloring techniques are based on one important result known as *the degree $<$ k rule.* Since optimal graph coloring is known to be NP-hard, these allocators compute an *approximation* of the optimal solution, meaning that they may fail to deliver perfect allocation for some program, and therefore introduce unnecessary spills. In practice these situations are rare, and recently Andersson [6] proposed an alternative optimal graph coloring algorithm which operates in linear time for all practical input programs.

2.1 The *degree* $< k$ Rule

While optimal graph coloring is known to be NP-hard, a remarkable result known as the *degree* $< k$ rule permits us to efficiently approximate k-coloring of most graphs encountered in practice with a very good precision.

The rule states that a graph containing a node with less than k neighbours is k-colourable precisely when the graph without that node is k-colourable. Further, a graph with only one node is trivially k-colourable for any k. Accordingly, we can obtain an approximation of k-coloring for a graph by recursively removing any nodes with less than k neighbours from the graph. Once we reach a graph with only one node, we colour it with an arbitriary colour, and proceed by adding the removed nodes back to the graph in the order that is a reversal of that in which they have been removed. As each node is added, it has, by construction, less than k neighbours in the partially-reconstructed graph, always leaving us with an available colour distinct from any of its neighbours.

(A) (B)

Fig. 2. Graphs non-colourable using the $degree < k$ rule

Consider, for example, the code fragment in Figure 1(A), with the corresponding interference graph given in Figure 1(B). Assume that our target machine has three registers available for storing variables ($k = 3$.) We can remove the nodes from the graph in the order s_2, s_3, s_1, s_4. We colour the remaining node (s_6), and proceed by adding s_4 through to s_2 back to the graph, arriving at the colouring in Figure 1(B).

2.2 Recovery Strategies

Of course, it is entirely possible for the above approach to arrive at a graph in which no node has fewer than k neighbours. The graph may simply be not k-colourable: for example, the graph in Figure 2(A) cannot be coloured for $k = 3$ (it requires a minimum of four registers for perfect colouring.) However, it is also possible for the $degree < k$ rule to fail on a k-colourable graph, such as the one in Figure 2(B). Specifically, this happens when the graph contains a *clique* of k nodes (a clique is a subgraph in which every node is pairwise adjacent.) Often, such graphs are still k-colourable, but the colouring cannot be obtained using the simple algorithm described in the previous section. In Figure 2(B), there are four cliques of three nodes: $\{s_1, s_2, s_5\}$, $\{s_2, s_3, s_5\}$, $\{s_3, s_4, s_5\}$ and $\{s_4, s_1, s_5\}$, but the graph is still 3-colourable. Both situations occur frequently in practice, and therefore it is essential for a graph-coloring register allocator to deal with them efficiently.

Surprisingly, with a simple optimistic modification, the $degree < k$ algorithm described earlier can recover gracefully from the above situations. When faced with a graph in which no node has fewer than k neighbours, we simply pick the node with fewest neighbours and remove it from the graph as normal, delaying the spill decisions to the second stage of the algorithm, when nodes are added back to the graph and coloured. For example, when colouring the graph in Figure 2(B), we would remove nodes in the order s_1, s_2, s_3 and s_4. Notice how, after optimistic removal of s_1, the remainder of the graph did not require further optimistic decisions and was trivially 3-colourable. This is very typical of the interference graphs encoutered in practice.

If we permit removal of nodes with k or more neighbours during the register allocation stage of the algorithm, it is no longer true that, when the nodes are reinserted during register assignment, they will always have fewer than k neighbours in the partially reconstructed graph. The reinsertion step has to be modified to account for three cases:

- If the reinserted node has fewer than k neighbours in the partially reconstructed graph, we can always find an available colour for it, so we proceed with the reinsertion as before.
- Otherwise, it is possible that some of the neighbours of the new node have been assigned the same colour. If all the neighbours use less than k colours, we can still colour the node without modifying the graph. Note that this permits to colour some graphs that were not colourable using the simple $degree < k$ algorithm, provided that we've picked a "fortunate" colouring for the previously-reinserted nodes (this was, for example, the case for the graph in Figure 2(B).) Therefore, delaying spilling until the reinsertion stage improves the ability of the algorithm to colour some graphs, but does not always achieve optimal colouring.
- Finally, if there is no colour available for the reinserted node, we must somehow modify the interference graph of the procedure before we can colour it. There are a number of options available. Most of these attempt to undo the effects of those optimizing transformations performed previously which are likely to have increased the register pressure of the code. Some possibilities, in the increasing order of runtime overhead [22], are:
  1. We can attempt to move some instructions closer to their use (thus undoing loop invariant code motion.) This has the effect of shrinking the lifespan of some variables, thus decreasing the likelihood of it interfering with another variable.
  2. We can clone a calculation to avoid keeping its result in a register (undoing common subexpression elimination.) Since many subexpressions recognized by the CSE transformations are trivial to recalculate, it is usually more efficient to recompute the value than store at several places in the program than store it in a temporary register at an expense of causing a spill (and therefore a memory access) elsewhere in the program.
  3. We can collapsing unrolled loops. Each iteration of a loop that has been unrolled usually requires additional registers so this technique can have a dramatic effect on the interference graph, but it is difficult to implement in practice and therefore none of the register allocators described in literature implement it.
  4. Finally, if all other approaches fail, the allocator is left with the recourse to spilling of some variables into memory as described in Section 2.3 below.

  Undoing optimizing transformations becomes critical to effective register allocation on modern super-scalar architectures, as these optimizations continue to be designed to extract more and more instruction-level parallelism

out of the programs, dramatically increasing register pressure in the process. However, most registers allocators still implement only the final alternative, with the notable exception of Johnson's VSDG approach [22] which is discussed in Section 4 below.

2.3 Variable Spilling

Spilling refers to the process of moving a variable into memory for some duration of its lifetime, effectively splitting it into two separate unconnected nodes in the interference graph. This requires storing the variable in memory (usually on the stack frame of the corresponding procedure) after its definition and loading the value back before its use. Typically, register allocation algorithms insert the spilling instructions indiscremenantly, relying on subsequent peephole optimizations to move the `store` instructions to the most efficient position in the instruction stream, and remove redundant `load` instructions after register assignment has been completed. [29] While spilling is usually the easiest recovery option for graphs that cannot be coloured otherwise, it can have a dramatic negative effect on performance by increasing memory traffic. [20] For that reason, it is advisable for a register allocator to minimize the number of spills in the program.

When using the graph coloring algorithm describe above, only interference graph nodes optimistically removed from the graph during register allocation may require spilling decisions to be made when reinserted into the graph during register assignment. Therefore, register allocators typically estimate the spilling costs for every variable in the program, before attempting register allocation, and use these spill costs to guide optimistic removal of nodes from the interference graph. When faced with the need to remove a node optimistically, the node with the minimal spill cost is removed first. This ensures that, during register assignment, that node will be considered for spilling first, hopefully rendering the reminder of the graph colourable and therefore avoiding further spills of more expensive nodes.

The estimate of spill costs for a single variable should take into account:

1. A spill performed in an inner loops of the program will be executed more often at runtime, and is therefore exponentially more expensive than that performed in an outer loop.
2. If a variable may be recomputed more efficiently than reloaded from memory, the cost of recomputing should be used instead.
3. Spilling the source or target of a copy instruction makes that instruction redundant, so these are the preferred candidates for spilling. Other instructions may behave similarly on certain architectures, so the spill costs should be weighted by instruction type.
4. If a spilled value is used several times in the same basic block it does not have to be loaded more than once in that block.

Accordingly, the cost of spilling a variable defined by the set of instructions D, used by the set of instructions U and copied by the set of instructions C can be estimated by the following equation:

$$\sum_{i \in D} w_i 10^{d_i} + \sum_{i \in U} w_i 10^{d_i} - \sum_{i \in C} w_i 10^{d_i}$$

where d_i represents the nesting depth of the instruction i within the loop structure of the procedure, and w_i represents the weighting assigned to its instruction type. The algorithm for computing the cost of spilling each variable is described by Chaitin.[9, 8] Other techniques for minimizing register spills on super-scalar architectures by tailoring the instruction sequence generated by the compiler are presented by Govindarajan. [18]

2.4 Optimal Graph Coloring

The graph coloring algorithm described above is only an approximation of the exact coloring, and may fail to colour some colourable graphs. Nevertheless, it works surprisingly well in practice, which suggests that interference graphs have some special properties that simplify their colouring.

After an investigation of 27,921 real-life interference graphs, Andersson [6] found that all interference graphs produced by a compiler belong to the class of *1-perfect* graphs. A graph G is said to be *1-perfect* if its *chromatic number* $\chi(G)$ (the smallest k for which G is k-colourable) is equal to its *clique number* $\omega(G)$, the number of nodes in the largest clique in the graph (a *clique* is a subgraph in which all nodes are pairwise adjacent.) Note that, while, for general graphs, the differene between the chromatic number $\chi(G)$ and the clique number $\omega(G)$ is large, Andersson was unable to construct an interference graph that was not 1-perfect, despite numerous attempts, which suggests that non-1-perfect interference graphs do not occur in practice. One possible explanation of this remarkable fact is the uneven density of interference graphs, which tend to be sparse in general with a small number of localised dense regions.

Andersson proposed to utilize these properties to design an efficient optimal graph coloring algorithm for interference graphs. The algorithm begins by searching the graph for the largest clique and colouring this clique with $\omega(G)$ colours, obtaining a partial colouring of the interference graph, which covers its "most difficult" portion. The algorithm then proceeds with the search for the optimal colouring by enumerating all possible colourings. Although this is exponential in the in the number of nodes in the graph, the search can be aborted when it finds a colouring using $\omega(G)$ colours. For 1-perfect graphs, once the largest clique has been coloured, the remainder of the graph is trivially colourable and the algorithm completes the search in linear time.

Andersson has found that his algorithm is, in practice faster than the approximate heuristics-based algorithm designed by Appel and George for use in their compiler for Standard ML of New Jersey. [17] However, Andersson's algorithm does not, as yet, perform actual register allocation, as it finds the smallest k for

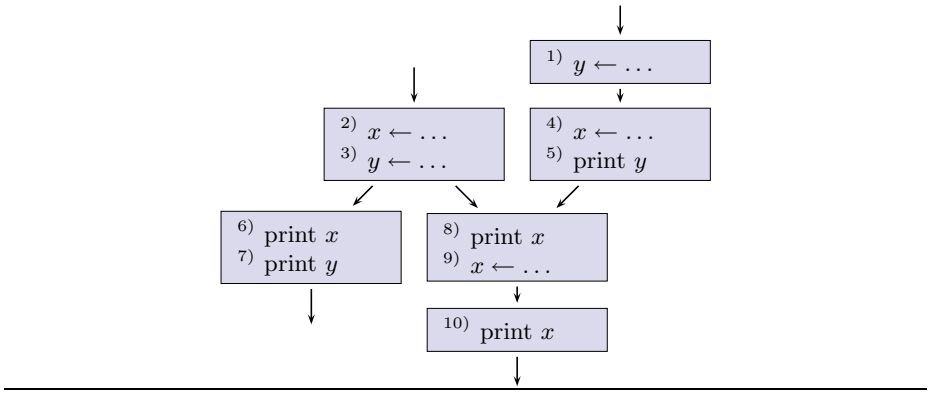


Fig. 3. Identifying Allocatable Objects

which the interference graph is colourable, without modifying the interference graph in situations when k is larger than the number of available registers.

3 Preparatory Transformations

A number of issues must be considered before we can attempt register allocation on the input program. First of all, we must decide what constitutes objects that we are trying to allocate: in compilers for imperative languages such as C, this choice is far from trivial. Typically, once the allocatable objects have been identified and the interference graph constructed for the program, compilers perform a simple but powerful *register coalescing* transformation to simplify several stages of the compiler design while significantly improving efficiency of the generated code. [7] In this section, we briefly discuss each of these issues, as they will impact the designs of intermediate representations discussed in Section 4.

3.1 Identifying Allocatable Objects

Although it is convinient to talk about variables in the context of register allocation, it is not, in general, a good idea to use source variables as input objects to the register allocator. In most programming communities, it is a common practice to use certain variable names such as i or x for predefined purposes, resulting in these source variables being used frequently throughout the program usually in unrelated context. For example, in C programs i is commonly chosen as the name of the index variable in loops and, as a result, is one of the most commonly-occuring identifier in a typical C program. [3] An interference graph constructed with a node for each source variable would contain many more edges than one in which, for example, every loop index was treated as a separate node.

These false interferences are a common problem, and therefore most modern intermediate program representations such as the static single assignment

(SSA) form [5, 14] discussed in Section 4.2 or the value state dependance graphs (VSDG) [22] from Section 4.3 are designed to assign a unique name to each definition point (assignment) in the program. However, using definition points as allocatable objects has its drawbacks, too, since two variables defined on separate control-flow paths may have a common use once the merge of the paths. For example, if, in the code:

```
if ... then
   x ← ...
else
   x ← ...
print x
```

the two definitions of x are assigned to different registers, than we would have to introduce copy instructions to move them to the location expected by the `print` statement.

To solve these problems, register allocators operate on *webs* of values instead of simple definition point, when a web is defined as a maximal union of all def-use chains that have a use in common. For example, in the code in Figure 3 there are four webs consisting of the following instructions: $\{1, 5\}$, $\{2, 4, 6, 8\}$, $\{3, 7\}$ and $\{9, 10\}$.

An important benefit of using webs for nodes in the interference graph is that it forces the register allocation to allocate results of computations performed in different branches of the control flow graph in the locations where they are expected by all subsequent uses of that value, including the cases when these registers are passed as parameters to function calls. This eliminates the need for copying parameter values to the registers allocated for parameters by the function call protocol of the target instruction set architecture. [25]

3.2 Register Coalescing

Most register allocators perform a simple, but powerful optimization called *register coalescing* or *subsumption* before attempting register allocation proper. Register coalescing is a specialized form of copy propagation which eliminates assignments of the form $x \leftarrow y$ when x and y do not interfere, by replacing all occurances of x with y. Note that this only requires the information stored in the variable interference graph, making this optimization an ideal candidate for integration into a graph coloring register allocator.

As pointed out by Chaitin [9, 8], register coalescing is a powerful transformation that simplifies many aspects of compiler design, allowing the compiler to capture complex requirements on register selection using simple redundant copy instructions inserted during earlier transformations and eliminated by the register allocation through register coalescing.

For example, removal of SSA ϕ nodes during conversion from the Static Single Assignment form discussed in Section 4.2 to the final machine code representation of the program can be implemented by converting ϕ nodes into redundant copy instructions to be eliminated by the register coalescing pass whenever feasible. [7]

Further, many instruction set architectures require that parameters to certain instructions are passed in a specific register, and most systems impose similar restrictions on calls to library functions through standardised function calling conventions. These restrictions are typically implemented through a redundant copy of the value into a temporary variable, and an introduction of additional interference edges into the variable interference graphs for the temporary variable. Register coalescing will eliminate the temporary variable and effectively propagate the corresponding interference information to all definition points of the correspoinding parameter. [25]

In its simplest form, register coalescing works by checking each copy instruction of the form $x \leftarrow y$ with the information stored in the interference graph. If x and y do not interfere, it replaces any instructions that wrote to y and modifies them to put their result in x instead. After register coalescing has been completed, a dead code elimination pass will remove the now-redundant assignments.

Note that the interference graph can be updated incrementally during the transformation simply by merging the interference graph nodes for x and y whenever register coalescing is performed on an assignment $x \leftarrow y$.

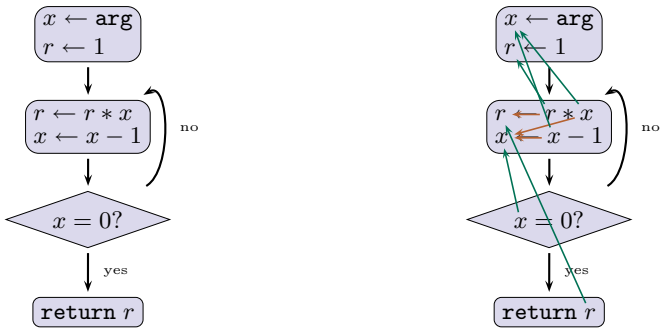
4 Bringing It All Together

While graph coloring gives with a systematic approach to register allocation, it does not, in itself, help to structure the compiler when combined with *ad hoc* web calculation and spilling algorithms. The choice of the intermediate representation for the program is still crucial to the overall structure of the compiler. In this section, we review the four most important alternatives available to the compiler designers.

4.1 Control Flow Graphs

Control flow graphs (CFG) are the oldest, and still most widely used program representation for use by optimizing compilers. [3, 29] Their orgins can be traced to Floyd's original flowchart semantics. [16] In this form, the instruction sequence is divided into segments called *basic blocks*, where each basic block begins with a label and ends with a control transfer instruction such as a jump or `return`, with no other control transfer instructions within the block. Each instruction closely resembles assembly instructions, in that it operates on and stores the result in symbolic variables. Control flow graphs are imperative programs in that variables can be redefined (updated), but, unlike the final assembler, the compiler assumes availability of an infinite number of registers for storage of values during execution.

Because of these similarities between symbolic variables and physical registers, the same CFG representation can accomodate both symbolic and register-allocated code, with the only difference between the two forms being the number and naming of variables referenced in the program.



(CFG) (DU chains)

Fig. 4. Control Flow Graphs and Def-Use Chains

CFG representations place the emphasis on the control flow information rather than data flow. [19] This is effective when used for control flow-based optimizations such as loop invariant code motion [12, 15, 24], but introduces problems when used for algorithms that rely heavily on data flow. Computation of webs for register allocation belongs to the later class of algorithms.

The explicit ordering of instructions imposed by CFGs introduces artificial dependencies. For example, in the following code:

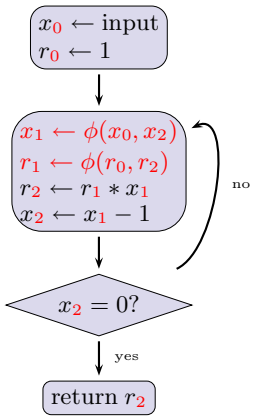
$a \leftarrow x + 1$
$b \leftarrow y + 2$
$c \leftarrow a + 3$
`return` $b + c$

all three variables interfere with each other. However, if the code was rearranged as follows:

$a \leftarrow x + 1$
$c \leftarrow a + 3$
$b \leftarrow y + 2$
`return` $b + c$

the interference between a and b is removed, improving the ability of the register allocator to colour the program. Normally, register allocators based on CFG representations do not attempt to address this problem.

Further, web calculations require detailed data flow information, which, in control flow graphs, are described by maintaining so-called *def-use chains*, representing the sequence of uses of each definition in the program. Def-use chains, represented by arrows in Figure 4, are a dense representation of the deta flow which is expensive to compute and represent. Mainly for those reasons, control



(From CFG to SSA)

```
proc fac(x) {
    goto L1
L1: {
       r0 ← φ(start:1, L1:r1)
       x0 ← φ(start:x, L1:x1)
       if x0 then goto L1 else ret r0
  L2:  r1 ← mul(r0, x0)
       x1 ← sub(x0, 1)
       goto L1
    }
}
...
```

(Factorial Function)

Fig. 5. Static Single Assignment Form

flow graphs are being progressively replaced by other intermediate representations, the most notable one being the *Static Single Assignment Form* discussed in the next section, although they have recently been revisited for use as a representation language in discrete event simulation. [13]

4.2 Static Single Assignment Form

The static single assignment (SSA) form [5, 14] is an imperative representation of programs which encodes data flow information explicitly by requiring that there be exactly one assignment for every variable. Figure 5 presents the factorial function in SSA form. Except for the two ϕ-functions, the program resembles standard three-address code: `fac` consists of three basic blocks connected by jumps, with the third (labelled L_2) constituting the main loop of the program.

The values of the variables x and r are updated in three places in `fac`. To achieve the single assignment property, we create a new variable for each update, splitting x into x, x_0 and x_1, and similarly for r. Since, the block L_1 can be reached either from the start block or from L_2, at the beginning of L_1 we must merge the two sources of values for x and r using the so-called ϕ-function, which selects a value based on the source block of the jump. Note that, in SSA, the single-assignment property is purely syntactic, since, in the loop L_1, variables are still updated at runtime on every iteration.

A procedure can be put in the SSA form by arranging the basic blocks into a *dominator tree*, where the parent of each node dominates its children. An assignment *dominates* an expression if every path to the expression from the start of the procedure includes that assignment. The SSA form has been popularised by Cytron et al. [14], who describe an algorithm for computing the dominance

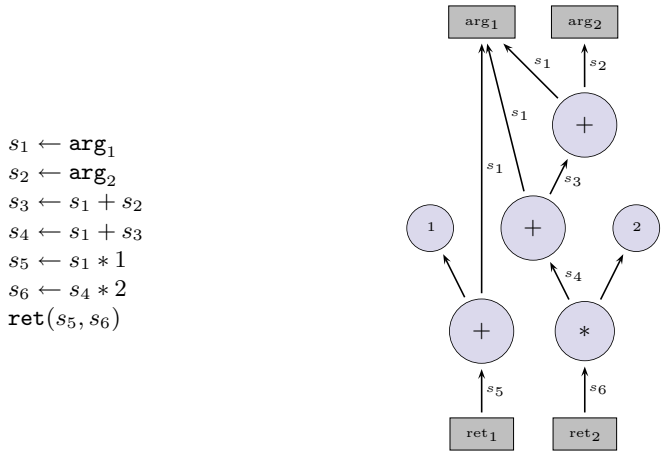


Fig. 6. Value State Dependance Graphs

information in linear time and demonstrate how SSA increases the power of many optimisations which require data-flow analysis.

Since an SSA variable uniquely defines the source of the corresponding value, the SSA form effectively annotates the program with an efficient sparse representation of the data flow information. Further, split variables tend to improve accuracy of data flow analysis by providing finer-grained notion of a variable.

The SSA form is the most popular representation used in compiles today. However, as for CFG representations, execessive control flow restrictions on instruction ordering remain in the way of register allocators (among other optimizations), and the ϕ nodes introduce new problems for register coalescing. Specifically, they translate to additional (and usually redundant) copy instructions when the program is translated to the final machine representation. Normally, these instructions would be removed during register coalescing, but, in the presence of ϕ nodes the computation of live ranges for variables (and, consequently, the variable interference graph) register coalescing is a difficult and relatively expensive operation, since the life-time of the variable which spans multiple blocks is obscured by the control flow information encoded in the ϕ nodes. Budimlić *et al.* [7] addresses most of the problems introduced by ϕ nodes by presenting an efficient register coalescing algorithm for removal of ϕ nodes during translation from an SSA form to a CFG representation.

4.3 Value State Dependence Graphs

To circumvent the major problem of artificial control flow restrictions imposed by both CFG and SSA forms when used for register allocation, Johnson [22] pro-

posed a new intermediate representation loosely based on the Gated Static Single Assignment form [32, 33], called the *Value State Dependance Graph* (VSDG.) When using the VSDG representation, register allocation is performed "in reverse": first, the program is restructed to make it colourable, and only then register allocation and assignment are performed on the program using standard graph coloring techniques.

In the VSDG form, a procedure is represented by a graph $\langle N, E_v, E_s, N_0, N_\infty \rangle$ where:

- N is a set of nodes
- E_v is a set of value dependence edges
- E_s is a set of state dependence edges
- N_0 is a unique entry node
- N_∞ is a unique exit node

The nodes in the graph can represent instructions (definitions) or the special structural nodes $\gamma(C, T, F)$ (lazy conditionals) and $\theta(C, I, R, L, X)$ (natural loops.) An example VSDG is presented in Figure 6. Except for the θ nodes, VSDG is an acyclic graph, representing all necessary data and state dependencies explicitly. Like the SSA form, it provides an efficient sparse representation of data flow, and, further, avoids imposing restrictions on instruction ordering except when required by program semantics. Further, as discussed by Johnson, VSDG have a fairly strong normalizing effect on programs, making them ideal for a number of optimizations such as common subexpression elimination. This is particularly useful if we want to perform common expression elimination during register allocation, thus avoiding the increase in register pressure commonly associated with that optimization.

The register allocation algorithm proceeds as follows:

1. First, we calculate the maximal distance from N_0 (DFR) for each node in the graph.
2. Next, we partition the graph into "cuts": sets of nodes with the same DFR.
3. Then, we calculate the set of live nodes for each cut: $S_{>d} \cap pred_v(S_{\leq d})$
4. Finally, we apply a forward snow-plough graph reshaping algorithm ensuring the size of each set is less than k. The plough proceeds from the bottom of the graph (the two `ret` nodes in Figure 6) up towards the N_0, ensuring that each cut contains no more nodes than the number of available registers, and inserting additional control dependence edges (representing register spills) to split large cuts. Details of the snow-plough algorithm are outlined in [22].

VSDG have been designed to represent both symbolic and register allocated code without loss of expressiveness. When using the VSDG representation, register allocation is the final compilation stage before code generation. All optimizations, including both portable and architecture-specific ones, are performed on the VSDG. Once the program has been optimized, redundant control-dependence edges are inserted into the VSDG thus converting the program into a standard CFG representation suitable for code generation.

4.4 Type-Theoretic Approaches

In order to formalize the role of register allocation in the compilation process, a number of people investigated application of type theory to register allocation. These can be classified into two broad categories: use of types as annotations expressing register requirements of expressions, and, more recently, proof-theoretic approaches which account for the entire process of register allocation and assignment, from liveness analysis to register assignment.

The first, more common approach, has been pioneered by Agat [1, 2]. In this approach, a standard call-by-value λ calculus is annoted with a type-effect system describing the register requirements of each subexpression. In addition to the location of function parameters and return values, the system maintains the so called *kill-set* of each subexpression, consisting of those registers that may be modified during evaluation of the subexpression, expressed as an *effect* in the type system. As example, the expression $3+5$ can be described in Agat's system with the following typing judgement:

$$\vdash \begin{pmatrix} \texttt{let}\ x\ \texttt{=}\ 3_{\mathrm{R8}}\ \texttt{in} \\ \texttt{let}\ y\ \texttt{=}\ 5_{\mathrm{R6}}\ \texttt{in} \\ \texttt{add}_{\mathrm{R8,R6,R2}}\ x\ y \end{pmatrix} : \textsc{Int}_{\mathrm{R2}}!\{\mathrm{R2}, \mathrm{R6}, \mathrm{R8}\}$$

which indicates that the expression returns its result in register R2 and modifies registers R2, R6 and R8 during evaluation. Agat's approach is particularly well suited for expressing flexible and adaptive function calling conventions, and therefore aids inter-procedural and inter-modular register allocation while also simplifying formal reasoning about correctness of the register allocation algorithm and its implementation. The annotations themselves are inserted for each procedure using standard graph coloring techniques.

Recently, Ohori [30] proposed a very different application of type theory to register allocation based on a proof-theoretic framework that accounts for the entire process of register allocation, designed to systematically combine all aspects of register allocation, from liveness analysis to register assignment, into a single framework. Ohori's approach is based on the *judgments as types* correspondence [21], according to which programs may be regarded as proofs of their types. If, following Agat's approach, we allow the types to express register requirements of expressions, then the process of register allocation may be formulated as a type-preserving program transformation performed using standard proof transformation techniques. This approach, unique in that it is not based on graph coloring techniques, can be easily incorporated into a static type system of the intermediate representation to unify pre- and post-allocation representations into a single intermediate language.

5 Conclusions

In recognition of its critical importance to performance of the compiled program, register allocation has progressively migrated from a minor optimizing

transformation to its current role as a central component of a modern compiler, influencing all aspects of compiler design. Most importantly, it is a major factor shaping modern intermediate program representations, with the predominant design goals being unification of pre- and post-allocation program representations, effectiveness, portability of register allocators and, more recently, considerations relating to formal verification of optimization algorithms and compiler implementations.

References

1. Agat, J., "A Typed Functional Language for Expressing Register Utilisation," Ph.D. thesis, Chalmers University of Technology and Göteborg University (1998).
2. Agat, J., *Types for register allocation*, Lecture Notes in Computer Science **1467** (1998), pp. 92–111.
3. Aho, A. V., R. Sethi and J. D. Ullman, "Compilers: Principles, Techniques and Tools," Addison-Wesley, 1986.
4. Allen, F. and J. Cocke, *A catalogue of optimizing transformations*, Design and Optimization of Compilers (1971), pp. 1–30.
5. Alpern, B., M. N. Wegman and F. K. Zadeck, *Detecting equality of variables in programs*, in: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1988), pp. 1–11.
6. Andersson, C., *Register allocation by optimal graph coloring*, Lecture Notes in Computer Science **2622** (2003), pp. 33–45.
7. Budimlić, Z., K. D. Cooper, T. J. Harvey, K. Kennedy, T. S. Oberg and S. W. Reeves, *Fast copy coalescing and live-range identification*, in: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (2002), pp. 25–32.
8. Chaitin, G. J., *Register allocation and spilling via graph coloring*, in: *Proceedings of the 1982 SIGPLAN symposium on Compiler construction* (1982), pp. 98–101.
9. Chaitin, G. J., M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins and P. W. Markstein, *Register allocation via coloring*, Computer Languages **6** (1981), pp. 47–57.
10. Chow, F. C. and J. L. Hennessy, *The priority-based coloring approach to register allocation*, ACM Transactions on Programming Languages and Systems (TOPLAS) **12** (1990), pp. 501–536.
11. Clack, C. and S. L. P. Jones, *Strictness analysis — a practical approach*, in: *Proceedings of a conference on Functional programming languages and computer architecture* (1985), pp. 35–49.
12. Click, C., *Global code motion/global value numbering*, in: *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation* (1995), pp. 246–257.
13. Cota, B. A., D. G. Fritz and R. G. Sargent, *Control flow graphs as a representation language*, in: *Proceedings of the 26th conference on Winter simulation* (1994), pp. 555–559.
14. Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman and K. F. Zadeck, *Efficiently computing static single assignment form and the control dependence graph*, ACM Transactions on Programming Languages and Systems (TOPLAS) **13** (1991), pp. 451–490.

15. Ferrante, J., K. J. Ottenstein and J. D. Warren, *The program dependence graph and its use in optimization*, ACM Trans. Program. Lang. Syst. **9** (1987), pp. 319–349.
16. Floyd, R. W., *Assigning meaning to programs*, in: J. T. Schwartz, editor, *Mathematical aspects of computer science: Proceedings of a Symposium in Applied Mathmematics of the American Mathematics Society* (1967), pp. 19–31.
17. George, L. and A. W. Appel, *Iterated register coalescing*, ACM Transactions on Programming Languages and Systems (TOPLAS) **18** (1996), pp. 300–324.
18. Govindarajan, R., H. Yang, J. N. Amaral, C. Zhang and G. R. Gao, *Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures*, EEE Transactions on Computers **52** (2003).
19. Hecht, M. S., "Flow Analysis of Computer Programs," Elsevier Science, 1977.
20. Hennessy, J. L., D. Goldberg and D. A. Petterson, "Computer Architecture: a quantitative approach," Morgan Kaufmann Publishers, Palo Alto, 1990.
21. Howard, W., *The formulae-as-types notion of construction*, in: J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, Academic Press, New York, NY, 1980 pp. 479–490.
22. Johnson, N. and A. Mycroft, *Combined code motion and register allocation using the value state dependence graph*, Lecture Notes in Computer Science **2622** (2003), pp. 1–16.
23. Jones, S. P., "Haskell 98 Language and Libraries: The Revised Report," Cambridge University Press, 2003.
24. Knoop, J., O. Rüthing and B. Steffen, *Lazy code motion*, in: *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation* (1992), pp. 224–234.
25. Kong, T. and K. D. Wilken, *Precise register allocation for irregular architectures*, in: *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture* (1998), pp. 297–307.
26. Lowry, E. S. and C. W. Medlock, *Object code optimization*, Communications of the ACM **12** (1969), pp. 13–22.
27. McFarling, S., *Program optimization for instruction caches*, in: *Proceedings of the third international conference on Architectural support for programming languages and operating systems* (1989), pp. 183–191.
28. Motwani, R., K. V. Palem, V. Sarkar and S. Reyen, *Combined register allocation and instruction scheduling*, Technical Report TR1995-698, Stanford University (1995).
29. Muchnick, S. S., "Advanced Compiler Design and Implementation," Morgan Kaufmann Publishers, 1997.
30. Ohori, A., *Register allocation by proof transformation*, Science of Computer Programming **50** (2004), pp. 161–187.
31. Patterson, D. A., *Reduced instruction set computers*, Communications of the ACM **28** (1985), pp. 8–21.
32. Tu, P. and D. Padua, *Efficient building and placing of gating functions*, in: *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation* (1995), pp. 47–55.
33. Tu, P. and D. Padua, *Gated ssa-based demand-driven symbolic analysis for parallelizing compilers*, in: *Proceedings of the 9th international conference on Supercomputing* (1995), pp. 414–423.
34. Wadler, P., *Deforestation: transforming programs to eliminate trees*, Theoretical Computer Science **73** (1990), pp. 231–248.