SOFSEM '92—Ždiar, Magura, Vysoké Tatry, Czechoslovakia, 22.11.–4.12.1992

Optimising compilation

Part I: classical imperative languages

Alan Mycroft and Arthur C. Norman

Computer Laboratory, Cambridge University
New Museums Site, Pembroke Street
Cambridge CB2 3QG, England
E-mail: `Alan.Mycroft@cl.cam.ac.uk, acn1@cam.ac.uk`

Abstract: This tutorial considers the design of modern machine-independent optimising compilers for classical (C-like) languages. We draw from two sources (1) the literature and lectures by the authors at Cambridge and (2) the 'Norcroft' compiler suite[1] jointly constructed by the authors.

Keywords: Optimising compilers, C, dataflow analysis, live variables, register allocation by colouring, available expressions, common sub-expression elimination, machine-parameterised and machine-independent optimisation.

1 Introduction

The classical imperative languages, from Fortran through to C and C++, all use a fundamental model of computation in which individual values are stored in variables, operated on and moved from place to place. The major differences between these languages are the syntactic constructs provided to allow users to structure code and the style and extent of compile-time type-checking. However, it is characteristic that type checking or inference determines a size in bytes for each variable and expression result (its 'machine type') together with a resolution of system- and user-defined operators into machine instructions or run-time library calls. In particular operators like "+" are mapped into their integer or floating-point versions or possibly a reference to a user-defined function. From now on we will assume the parse-tree has been so converted into a form which includes explicit typing of all expression nodes by machine types and of operators by machine-oriented operations.

Before going further it is perhaps worth briefly considering whether taking this approach is liable to limit the scope for optimisation in any important ways, or indeed if it will make a compiler harder to write or less efficient in operation. Provided the parse tree generated is an honest reflection of the user's code all that type-checking has done is to add information to it; this clearly cannot harm optimisation.[2] It is often sensible to arrange that a front-end simplifies certain expressions. Indeed the language often requires

[1] Commercial interests are referred to Codemist Ltd., "Alta", Horsecombe Vale, Combe Down, Bath BA2 5QG, England (fax +44 1225 837430)

[2] Note however that dumping the parse tree for languages such as Fortran and C++ in terms of C for input to a C compiler may well forestall optimisations as the original language may well provide restrictions which C does not. For example, two array formal parameters of a Fortran procedure cannot be written to if they overlap (and hence may be treated as non-overlapping) whereas this does not hold for C.

this, *e.g.* C array sizes. Moreover, we consider it desirable that semantically overlapping constructs (*e.g.* "++" and assignment in C) and constructs which exist in system-provided and user-defined forms (such as a built-in "`complex`" type in Fortran and a similar definition as a C++ class) should be normalised by mapping into abstract parse-trees whose constructs are reasonably orthogonal. This simplifies the work of the optimiser by exploiting source-to-source equivalences and encourages optimisation of general forms instead of special cases. In general later phases of the compiler will also need to be able to perform simplifications (*e.g.* further evaluating expressions after constant propagation) or even partially reverse the abve simplification to exploit hardware idioms. We believe it important that an intermediate language can express structured and non-structured constructs identically wherever possible—thus loops constructed from `while` and `goto` should produce the same code, as should C forms like `a = *p++;` and `a = *p; p = p+1;`. This is particularly important for the increasing use of C as a target language for higher-level language compilers.

Optimisations can normally be classified into several kinds. One is tree-oriented optimisation, in which parse-tree manipulations take place. This includes things like constant reduction and algebraic rearrangement. It cannot easily express certain types of optimisations because the dynamic behaviour of the code does not always follow the tree-structure—indeed code parts distant in the tree can often become adjacent in the generated code bringing forth additional optimisation possibilities. A second form is referred to as 'local' optimisation in which knowledge for optimisation is valid approximately within basic blocks (see later, but here meaning up to the next conditional jump or label). For example, if an expression calculated in a register is stored in a variable allocated to memory, then the value in the register may be sometimes re-used in subsequent code instead of re-loading the variable. Peephole optimisations are generally of this form. These types of optimisations do not seem able to encompass facts such as that two variables in the same scope, possibly with different types, can be allocated to the same machine register because they are never simultaneously live. For this we turn to so-called 'global' optimisation. This misnomer usually refers to optimisation within a single procedure. It consists of calculating details of dynamic execution (*control* and *data flow*) throughout a procedure—for example whether some expression is *available* at a given point because all control flow paths to it necessarily involve its previous calculation. Calculating global dataflow analysis details and exploiting them is the subject of this paper. Finally, there are 'inter-procedural' optimisations which consist of optimising one procedure on the (possibly recursive) basis of optimisations in another. Examples are choosing register allocations in one procedure to be disjoint from another to reduce register save/restore code and exploiting the fact that a particular procedure does not affect some global variable.

Given a parse-tree the approach we describe involves a sequence of transformations. First we generate intermediate code from the parse-tree. Arithmetic operations are mapped into *3-address* intermediate instructions such as

```
ADD.int32       r1, r2, r3      % i.e. r1 = r2 + r3
LOAD.float64    r4, r5, 8       % i.e. r4 = *(r5+1)
```

in which each operand is used as source or destination, but not both. This form has the advantage that all intermediate values are explicitly visible; hence analysis and especially transformation are much simpler than a beginner's treatment of compiling *via* a stack-oriented (0-address) intermediate code. Comparisons, data movement and unary operations map to degenerate forms having only two operands; special instructions handle procedure call and other less regular special needs. In principle, we translate the parse tree into a *flowgraph*—a directed graph in which the nodes hold such instructions and the arcs joining nodes represent (conditional) jumps. Dataflow information (such as liveness information) can later be attached to each node. However, since this dataflow information can be quite large, it is convenient and standard to partition this graph to form a flowgraph in which the nodes represent *basic blocks* of sequential instructions which do not contain entries (except at the top) nor jumps (except at the bottom). A block has one successor block for each branch destination—two for conditional jumps, many for switch statements and none for blocks ended by return.

The flowgraph format of basic blocks of 3-address code is a convenient one to use for performing various forms of global analysis and transformation on the code. Common sub-expressions and loop-invariants can be detected and processed, code may easily be moved (in contrast to stack-oriented code), code that is unreachable or that has no effect can be deleted and control-flow oddities (*e.g.* jumps to jumps) vanish.

Part of the analysis performed on the basic block representation will find the ranges over which variables or temporaries used in the program are *live*. On the basis of this information an attempt can be made to decide which values can reside in machine registers (modern RISC machines typically have enough registers that this will turn out to include almost all heavily used variables) and which have to be stored on a stack.

Because the problem of generating intermediate code and simultaneously allocating registers is hard, it is productive to generate intermediate code in which the machine is assumed to have an infinite number of *virtual* registers (tagged as to their use as integer or floating point) and to use one for each temporary or local user-variable. A second phase is then used to map these virtual registers into *real* or *physical* registers of the target architecture. This breaks a complicated single-pass problem into a manageable two-pass problem.

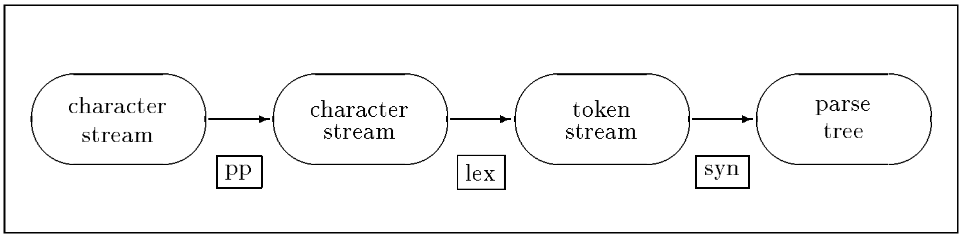
The directed graph of basic blocks is next flattened into a single linear stream of abstract machine operations. The order in which blocks are placed can influence the sense that must be given to conditional branches and the span of branch instructions generally. In the Norcroft compilers we provide for two levels of peephole optimisation to be performed at this stage. One is machine-independent but allows for the restrictive simplicity of the 3-address code by reconstructing some commonly occurring slightly higher level operations such as replacing procedure-call/return sequences with a tail-call branch. The second is target-specific and uses detailed knowledge about the target instruction set to rearrange code to take better advantage of it.

In a number of places the steps outlined above need to know something about the target machine—for instance register allocation is only possible when it is known how many registers will be available. However the information needed can almost all be expressed in terms of a modest number of characterising features that can be set up in a table before compilation starts, and it is reasonable to think of the entire process thus far as machine-independent but machine-parameterised. Its output is a linear stream of 3-address instructions which use registers in a way that will be compatible with the target machine.

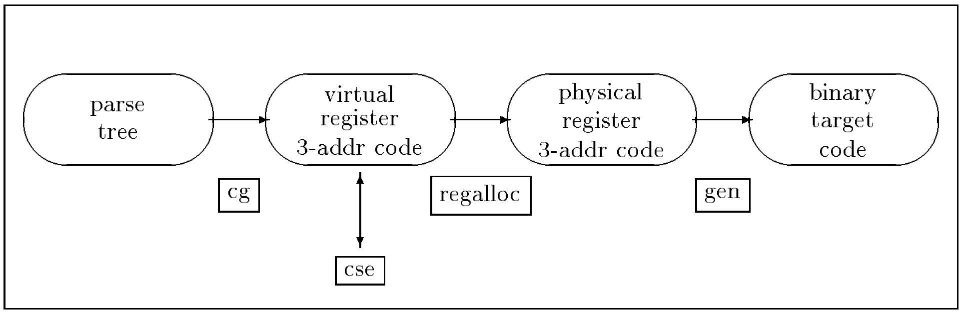
The final stages of compilation involve mapping these abstract instructions into the bit-patterns or mnemonics needed for the real machine. As well as instructions it is necessary to emit data definitions, relocation records and debugging tables. It is also often useful to have yet further optimisation of the code to exploit delayed branches on machines where these are relevant and to schedule instructions to avoid unnecessary pipeline clashes.

In the Norcroft compiler this 3-address to machine code conversion constructs binary object files directly; if the user wants to see textual assembly code we disassemble the binary, making use of the tables of internal label names *etc.* to help generate tidy output. Our decision to work in this way was motivated by the expectation that generating object code would be the common case and hence the one that should take the most direct path through the compiler.

A couple of pictures summarise the compiler structure using ANSI C for concreteness. These show the data representations used in ovals, and the names of parts of the compiler as names (in boxes) attached to arrows that indicate transformations performed. The first picture shows that parsing will often be split into a preprocessing phase (*i.e.* macro expansion and the treatment of conditional compilation) [pp], followed by lexical analysis [lex] to convert from a character stream representation into a token stream one. The final step is to parse and type-check, converting the token stream into a parse tree:



After parsing there are the parts of the compiler discussed here: 3-address code creation [cg], common sub-expression elimination [cse], register allocation [regalloc] and the creation of target machine code [gen]:



Note only 'gen' is essentially machine-dependent.

The remainder of this paper describes some of these components above in more detail and we conclude with a discussion of C constructs which facilitate or inhibit optimisation.

2 Intermediate code generation

Here it will be convenient to suppose that compilation and optimisation is performed one procedure at a time. Thus the fragment of parse tree presented for optimisation will represent the definition of a single function; hence we discuss so-called 'global' optimisation but not 'inter-procedural' optimisation.

The introduction explained the use of virtual registers which are used to generate basic blocks of 3-address code by a simple tree-walk; we use a fresh virtual register for every separate value that is computed. The set of virtual registers can be divided into three sub-classes. The first of these stand directly for real machine registers and will be used when code conventions (*e.g.* the location of the result handed back by a procedure) mean that register use is known very early on. The second class of virtual registers are used in contexts where later stages of the compiler must be able to allocate a real register—for instance all operands in arithmetic instructions are of this sort. Finally there are virtual registers that stand for user variables and for temporary values that have more than a very short life-time. These are referred to as *spillable virtual registers* and for each one a (separate) location on the stack is provisionally allocated. Final allocation of location is only done after register allocation to avoid wasting stack locations. The 3-address code is generated in such a way that spillable virtual registers only occur in very simple data movement instructions so that if they do in fact degrade to being kept on the stack it is still possible to generate code to access them. The 3-address code is constructed on the supposition that spillable virtual registers retain their values across procedure calls.

Naturally this leads to a large number of virtual registers, with many of them used over only very short sections of the complete program and with many apparently redundant movements of data between them.

As discussed in the introduction, 3-address instruction opcodes are provided for the usual complement of loads, stores, integer and floating point operations and housekeeping. The exact set of operations provided does not seem to be especially important, although support for additional source languages and target machines can sometimes make it useful to add extra opcodes to express some especially important idiom—for example, we find it convenient to have a `MOVC` opcode which represents C structure assignment (of known size and alignment) rather than generate a procedure call or synthesise a loop.

By having a fairly basic intermediate code this stage of the compiler becomes simple—but the code involved is still bulky since every syntactic construction potentially present in the parse tree has to be covered. Part of the simplicity of the intermediate code is that an unlimited supply of virtual registers are available. There are two ways in which this idealisation cannot be completely maintained throughout the construction of intermediate code. The first involves function calls on machines where the standard calling convention passes at least some arguments in registers. As indicated before this is coped with by allowing codes that stand for the real machine registers to occur in the place of the name of a virtual register, and thus arguments can be loaded directly into the registers that they are needed in and results can be retrieved from the physical registers they are returned in. Because the real registers used in such contexts are potentially a scarce resource it is useful to compile code that prepares all arguments in virtual registers and then copies values into physical registers at the last moment just before the `CALL` instruction. Similarly returned values are best transferred into a virtual register for later use.

The second complication relates to spillable and non-spillable virtual registers. It is essential that the number of non-spillable virtual registers live at any stage is kept smaller than the number of real registers in the target machine. Extremely complicated expressions can require extra copying of some non-spillable temporary values into spillable registers—thus an addition of two sub-expressions P and Q where Q may need to use many registers might turn into:

```
<code to load P into r1>
STORE     r1, -, r2      % r2 is spillable
<code to load Q into r3>
LOAD      r4, -, r2      % recover first operand
ADD       r5, r4, r3     % do the arithmetic.
```

However if Q is a very simple expression or if the above addition occurred in a small procedure where registers were not under strain it would be unnecessary to introduce `r2`.

The introduction of spillable virtual registers as sketched above is not the only way of allowing for limited register availability at the end. The main alternative is to start by generating 3-address code without any spillable virtual registers and then re-start the analysis after having introduced spill code if later register allocation fails. Our approach increases the bulk of 3-address code that has to be scanned, and raises the challenge that instructions such as

```
MOVE      r1, -, r2
```

should be annulled wherever possible by allocating both virtual registers `r1` and `r2` to the same real register. However, this enables the analysis to be completed in one pass and has the additional benefit that the register allocator can also exploit such copy information for user-variables (see section 4.3).

3 Common sub-expressions

At any point in a body of code the *available expressions* are just those values that are certain to have been evaluated on any path leading up to the specified point. Code that re-calculates an expression that is already available is redundant and can be replaced by a reference to the previously computed value. Let $avail(p)$ be the set of expressions available at (actually just before) a given program point, p, and p_0 be the entry point of the procedure. Then the $avail(p)$ satisfy the *dataflow equations*

$$\begin{aligned} avail(p_0) &= \{\} \\ avail(p) &= \bigcap_{q \in pred(p)} (avail(q) - kill(q) \cup gen(q)) \quad \text{if } p \neq p_0 \end{aligned}$$

where $gen(p)$ is the set of expressions computed by p (typically just its right-hand-side) and $kill(p)$ the set of expressions killed by p (for example an assignment to `x` kills all expressions containing `x`). Because the set of expressions available at a given instruction node p depends on the expressions which are available at its predecessor nodes $pred(p)$ this is said to be a *forward* dataflow analysis.

Computing available expressions from these implicit equations is done by establishing names for all results ever calculated, and initially flagging all of them as being available, except at p_0. A forward scan through the code can then reveal both when calculations make some value available and when assignments render the value out of date. For example in the sequence

```
c = a + b;
d = a + b;
a = a + 1;
e = a + b;
```

the expression `a+b` becomes available at the first line, and can be re-used on the second, but the assignment to the variable `a` must invalidate it so the the calculation of `e` must be started from scratch. Availability information must be propagated across the boundaries between basic blocks, and loops in the source code mean that in general the analysis proceeds by iteration. Scanning must be repeated until the information stabilises. This is guaranteed by a lattice-theoretic monotonicity argument.

For some languages with tightly constrained formats of loop structure it is possible to decide how many scans will be needed based on the depth of nesting of loops, but we prefer to have a single robust optimiser capable of interfacing to parsers for a variety of source languages and so just iterate until we have found a fixed-point.

Available expression tabulation can reveal constant calculations that did not correspond to obvious opportunities for constant folding in the front-end of the compiler—so even if the parser did not evaluate such expressions they can be reduced here.

In general we follow the above scheme and use new (spillable) virtual registers to hold the values of common sub-expressions. Similarly the available expression information makes it possible to identify loop invariants. A fresh basic block can be created at the head of the loop[3] and code to evaluate the invariant can be installed there. References within the loop then get replaced by the regular common sub-expression mechanism. Some common sub-expressions, however, are so simple to calculate that although it would be nice to re-use the previous value if it was available in a register it would not be worth saving it onto the stack but would be better to recompute. In such cases we combine a reference to the available virtual register with the instruction to recompute the value—and select which part of the code to use after register allocation.

Note that choosing appropriate common sub-expressions depends on the target machine parameterisation table. Consider code like

```
int f(int i, int *v, int *w) { return v[i]+w[i]; }
```

and suppose `int` to be 4-bytes wide on a typical byte-addressed target. Hence `i` has to be left-shifted by two as part of the indexing process. If the target has scaled addressing (the ability or requirement to shift an index during address calculation) then the two implicit `t = i<<2;` instructions should not be lifted out as a common sub-expression whereas if it lacks scaled addressing the common sub-expression should be exploited.

4 Register allocation

Following CSE and loop optimisation the 3-address code needs to be analysed to see how virtual registers can be mapped onto real ones.

In principle this proceeds in the following manner. First the live ranges for all virtual registers are determined. This process is analogous to that which identified available expressions. Instead it has *backwards* dataflow equations—the set of live variables at (just before) a node p depends on the liveness information of its successor nodes $succ(p)$. The $live(p)$ satisfy

$$live(p) \quad = \quad \left(\bigcup_{q \in succ(p)} live(q) \right) - kill(q) \cup gen(q)$$

where $gen(p)$ is now the the set of variables referenced by p (for example by occuring in the right-hand-side of an assignment) and $kill(p)$ the set of expressions defined by p (for example the left-hand-side of an assignment). Computation of live sets for each instruction

[3] In Norcroft compilers this is done by placing the code at a 'dominator' block which allows loops created by `goto` to be treated identically to those created by more structured techniques.

starts with the supposition that no registers are live until the contrary is proved, and works backwards from the exit points of blocks. A live register is taken to be one whose value is going to be used, so at blocks terminated by a return instruction the only live registers are those holding values to be returned to the caller. On scanning backwards past an instruction such as

```
        ADD     r1, r2, r3
```

register `r1` can be removed from the list of those that are live (before the `ADD` instruction `r1` is not needed), while `r2` and `r3` are added to the list. Again the analysis must iterate until it stabilises. Live variable analysis can reveal code that computes values that are not then used and variables that are used but not initialised ('dataflow anomalies'). Sometimes this may be a consequence of transformations made by earlier stages of the optimiser and the redundant code can be removed without comment, while sometimes it will indicate oddities that should be reported back to the programmer—for example a user-variable which is live at a point where it is undefined (by scope entry) represents a warning that "variable 'x' may be referenced before definition".

Now live ranges are inspected, and any pair of virtual registers whose live ranges overlap are said to *clash*. Two clashing virtual registers may not be assigned to the same real register, but any number of non-clashing virtual registers may. If in the real machine registers a_1 to a_n (say) are corrupted by a procedure call, while v_1 through v_m are preserved across calls we can arrange that a_1 to a_n are marked as clashing with each virtual register that is live across a call. Similar treatment can allow for other cases where there are constraints on which real registers are available in some context.

Clash information can be interpreted as defining an undirected graph where virtual registers are the nodes and clashes are edges. If the target machine has k real registers and this graph can be vertex-coloured in k or fewer colours (with no directly connected vertices painted the same colour) then the colouring immediately shows how to map virtual onto real registers.

The result, in principle, is that the 3-address code is now transformed to use physical registers instead of virtual registers and can be turned into target machine instructions, *e.g.* by simple macro-expansion.

Of course, in practice things are not so simple.

- First we need to assume that the target's register and instruction sets are sufficiently orthogonal that any register suffices for almost[4] any instruction (or at least with a negligible performance penalty). This is true for many modern machines, especially RISC with on-chip register files containing 16 or 32 registers, such as the MIPS processors, Sun SPARC, Motorola 88000, Intel 860, IBM rs/6000 and a good approximation on the Intel 80386/80486 in 32-bit mode. The Motorola 680x0 range is a fair approximation (we found that we needed additional code to prefer an 'A' or 'D' register based on the uses of a variable) whereas the 8086/80286 and INMOS transputer mismatch fairly comprehensively.

- Secondly, we have not explained either how to k-colour graphs, or how to cope when the graph *chromaticity*, *i.e.* minimum number of colours which suffice to colour it

[4] A limited number of exceptions can be coped with gracefully, such as dedicated registers for multiply/divide are in principle no worse than the (procedure-calling-standard-dedicated) register holding the result of a function call.

exceeds the number of physical registers. In this case we must arrange to *spill* variables from registers to main memory for all or part of their range. Choosing effective spills is really the hardest part of register allocation. Norcroft compilers keep the number of live non-spillable virtual registers smaller than the number of physical registers so this colouring is always possible by sufficient spilling.

- Thirdly, there are places where the suggestion that registers clash if they have overlapping live ranges is over-pessimistic and can lead to inefficiency. The main case we encounter of this is after a simple move instruction where two registers in fact contain the same value. Consider the fragment of C code

```
r1 = r2;      /* r1 and r2 contain the same value */
<some big calculation>
r3 = r1 + r2; /* r1, r2 not used after this */
```

  where `r1` and `r2` are both live across the big calculation, and so would need to clash by our definition. However since they always contain the same value forcing this value to be kept in two separate real registers is unnecessary. Code of the above style can often arise as a consequence of the work of the common sub-expression removal code. The Norcroft compilers use the idea of one register being a 'read-only copy' of another to remove the worst of these.

- Finally, we have two forms of efficiency considerations. The graph colouring problem is NP-complete (just the process of colouring with the minimal number of colours, even assuming we knew it in advance) and hence we must use heuristic solutions. Moreover, certain routines generate large numbers of virtual registers, for example compiling the Edinburgh ML interpreter generates around 2000 and compiling Sun's NeWS more than 6000 (both by fairly aggressive use of macros). In the latter case the clash-graph risks being 36Mbits big which is bad enough for space reasons but searching for clashes in the colouring process can take a great deal of time too. (It can be fun to try your favourite optimising compiler on such examples with such optimisations enabled.) The problem is that the clash graph is globally sparse and locally dense. A small fraction of registers represent global (to a routine) variables which clash with most temporaries, but most temporaries only clash with a few other registers, so the design of a suitable data-structure is critical for performance.

The rest of this section considers various aspects of this problem in more detail.

4.1 Graph colouring

Given that k-colouring a graph is an NP-complete problem it is unreasonable to seek an algorithm that guarantees efficient solutions to it. A good heuristic method which often allocates all registers in small procedures works by gradual reduction of the size of the graph to be considered. Observe that if any vertex in the graph has $l < k$ neighbours then if that vertex is left to last there will always be at least $k - l$ colours available to assign to it. Thus any such vertex can be removed from the graph and placed in a (LIFO) queue for subsequent colouring. Of course removing a vertex from a graph will reduce the valencies of all the remaining vertices that it had been connected to—this may make some of them candidates for removal. If iterating this process leads eventually to an empty graph then the queue built up will give an order in which colouring can proceed with a guarantee of success. If this pruning stops then at least it will have reduced the size of the graph to which any more elaborate methods have to be applied.

If the residual graph is sufficiently small some more or less exhaustive search may be feasible—however failure of the simple method is more usually an indication that some virtual registers will need to be mapped onto stack locations rather than onto real registers (*e.g.* a $(k+1)$-clique). In the Norcroft compiler we first push the remaining registers onto the allocation queue with the ones with fewest clashes to be processed last and then start an attempt to colour registers in the order so produced. For each virtual register we assign the smallest numbered[5] real register that is consistent with clashes with registers already allocated. If the pruning phase of register allocation was successful this is guaranteed to succeed, and in many other cases it will too. If a register is found that cannot be coloured in this way we map some virtual register to the stack and restart the colouring process.

4.2 Spilling

If colouring fails it is possible to extract from the failure the set of simultaneously live virtual registers that caused the trouble. Spilling involves the selection of one of these, and good optimisation demands that this choice be made rationally. Thus earlier, while scanning the basic blocks, we count the number of uses made of each spillable virtual register, and give extra weight to uses within loops. If the user has given C `register` declarations we add that information into the use-statistics associated with each virtual register. Then if spilling is needed we can discard the register that seems to have lightest use. An effect of this strategy is that user-provided register declarations are not needed and have no effect in procedures small enough that all values live in registers anyway, and in larger procedures they can be used to guide, but not totally override, the optimiser's analysis.

At the end of the complete colouring process as described above we re-visit those virtual registers that were marked as spilt, and see if they can after all now be assigned to real registers. In large procedures it is quite common to find that several of them can.

Note that we have described a form of register allocation in which a variable is either allocated to one register throughout its scope (even though other non-clashing variables may also be allocated to it) or spilt into memory. Some compilers allow each live range to be allocated separately to a register or memory. This can give better performance but can significantly increase space (and hence time) pressure on the clash-map.

4.3 Copy avoidance

The allocation strategy described above still has some flexibility left in it—if some register clashes with significantly less than k others then there will be choices available when it is eventually coloured. This can be exploited by keeping a *preference* or *copy graph* which records when code would be improved by mapping a given register onto the same physical register as another. An arc (`r1`, `r2`) is placed in the preference graph when a `MOVE r1,-,r2` instruction occurs or when the target machine has 2-address operations and a 3-address instruction such as `SUB r1,r2,r3` occurs. The copy graph can be implemented in the same manner as the clash graph. Note that copy avoidance means that the spillability code introduced in section 2 can evaporate gracefully when it is not required.

[5] Actually we have a preference order for the use of real registers which does not necessarily correspond to their natural numbers.

4.4 Data structures

The main data structure needed in register allocation is the clash graph. Two obvious possible representations for this would be a square array of bits and a linked structure capable of exploiting sparsity. For the compilation of small procedures it hardly matters which is used, but for the optimisation of large blocks neither of these formats is entirely satisfactory. This is because to a large extent virtual registers fall into two classes. One class will be used for short spans, and will thus tend to clash with only a modest number of others. The other class consists of registers live across significant stretches of code, and these can clash with almost everything else. If the virtual registers were re-numbered so that the ones with most clashes were given small numbers, then a picture of the matrix representation of the clash graph would show symmetry about the leading diagonal, dense blocks of entries in the top rows and left columns and sparseness towards the bottom right. With this in mind it is easy to invent a mixed representation that uses linked chains of small bitmaps, so that the bitmaps can cover dense parts of the relation efficiently while the linked lists allow for sparseness where that is important. It is clearly sensible to keep just the part of the matrix that lies above the diagonal. Although in early experimental versions of our compilers the register allocation phase took more time and space than the rest of the compilation process (at least for medium to large procedures) with the mixed sparse/bitmap data structures we now find the cost of optimisation reasonable—often comparable with that of preprocessing and lexical analysis!

5 Conversion to linear code

At the end of register allocation the code is still represented as a directed graph of basic blocks, but it is known which virtual registers in 3-address instructions have to map onto stack references and which can use real registers. This will mean that many stack locations provisionally reserved in case some virtual register needed spilling will be found to be unwanted. By keeping stack references as logical names rather than absolute offsets until after register allocation we can ensure that this does not result in waste.

The order in which basic blocks are placed to create linear code can be guided first by the numbers of arcs joining them. For instance if some block only has one entry then it should be placed immediately after its predecessor, in effect merging the two. A natural side-effect of the process of linearising the code is that jumps to jumps get removed and blocks of code that are never referenced get discarded. For target machines where jump instructions have short forms with limited span it can be useful to produce estimates for the bulk of real code that each block will expand into and to try to arrange blocks so as to maximise the number of jumps that will be short.

While converting the 3-address code to linear form it is useful to apply a number of local conversions based on characteristics of the target machine but still conducted at a fairly machine-independent level.

- Some constructs, held as abstract higher level operations throughout CSE optimisation and register allocation, may be expanded at this stage. For instance for the benefit of some targets which do not have direct support for halfword memory access the relevant combination of shifts and masks may be generated. Since it is now known how many machine registers are free for use as temporary workspace suitably unrolled versions of memory-to-memory copy loops can be synthesised. Of course there is often some tension about how early these conversions should be performed—a reasonable choice is to defer to this stage those things where different expansions might be suitable according to the number of real registers available.

- In the opposite direction sometimes it is useful to have a richer collection of 3-address code to pass on to the final code generator than are used throughout the machine-independent optimisation phases. This can simplify CSE optimisation and register allocation. While linearising it is thus appropriate to detect either common idioms or ones especially directly supported by the target machine and re-instate them. Again it is a matter of judgement which higher-level constructs should be carried intact through the complete compiler and which should be expanded into elementary components when the 3-address code is created and reconstructed when it is converted to linear form. Other things being equal we prefer expansion followed by reconstruction, since the reconstruction process will work whether the operation was manifestly visible in the original source code or not. Two examples from our compiler spring to mind—while flattening the flow-graph we can recreate scaled indexed address modes, and can turn C calculations such as

```
unsigned int a, b; /* assume 32-bit machine */
a = (b << n) | (b >> (32-n));
```

  into a rotate operation. Another example is that we typically maintain accesses to global variable as 'load address' followed by 'indirect access' since this is more efficient if a given variable (or set of variables in a known storage relationship) must be repeatedly accessed. On the other hand, if we find such a pair of instructions in which the register holding the address becomes dead at the indirect access then we can contract this to a 'load from absolute memory' instruction on machines possessing same.

- Fully effective detection of idioms for reconstruction means that it must be possible to rearrange the order of instructions within basic blocks. A fair amount of flexibility in this can be maintained provided full register use information is passed on from the register allocator. Sometimes it is useful to try to rearrange code in a way that will allow for the generation of specialised machine instructions or addressing modes. For instance migrating increment instructions to a memory reference provides the opportunity to exploit 'auto-increment' addressing. On several machines we have come across it is useful if the two load instructions implicit in

```
f(struct foo *x) { ... x->a ... x->b ... }
```

  can be brought into adjacent positions, for then it can be possible to combine them into a load-double instruction that picks up both words at once. C, of course, prescribes the offsets of fields `a` and `b`.

6 Machine dependent optimisation

To the greatest extent possible it is desirable to keep optimisation as generic as possible. The earlier stages of a compiler can very properly be made conditional on features present or lacking in the eventual target machine (register windows, memory-to-memory move instruction, number of real registers, presence of scaled addressing *etc.*). However eventually a stage is reached where real machine instructions have to be generated and when optimisations depend on the precise form and layout of these.

Two classes of code improvement seem to be relevant. The first is concerned with the selection of suitable machine instructions for the 3-address code, especially taking advantage of operations that the abstract 3-address code makes explicit but that real machine instructions can achieve as a partial consequence of some other operation. An example of this could arise where byte-wide machine operations could provide a simple way of implementing an operation such as `a = b ^ (c & 0xff)` avoiding need for an explicit mask with 0xff. Typically each individual such optimisation makes only a small difference to the performance of real optimised code, but regardless of performance implications it is socially vital to include a broad range of such peep-hole optimisations, since otherwise people who inspect the code generated by the supposedly optimising compiler will comment on their lack. We have found it important to provide at least one way in which each given machine idiom may be generated from C, in that certain uses (*e.g.* graphics applications) require the ability to code in C (for maintainability) but with quite specific assembler output expectations.

The second class of machine-specific optimisation is important when the target is one of the current generation of processors where consequences of the pipelined implementation of the computer show through at the machine code level. They include the exploitation of delayed branches and the rearrangement of code so that instructions that reference a register are separated from preceding ones that load it. As before it is useful to keep information about register use and basic blocks at least partially available. Register liveness spans constrain how instructions may be permuted. The optimiser can also leave indications of when spare registers are available and code may be altered by renumbering[6] registers so as to make more aggressive code movement valid. The remainder of the basic block information (now distilled into tables that will turn into linker directives to relocate symbol-references) can help show when instructions at the head of one block may be migrated to a delay slot at the end of a predecessor. Even though the detailed optimisations to be performed depend on very fine details of the target machine to a large extent the code that implements them can be factored into a generic part together with small routines and tables that assess a cost function that is to be minimised.

7 The optimisation phase problem

One of the most frustrating problems in designing and constructing a practical optimising compiler is that of deciding on the order in which various transformations should be performed. For instance if common sub-expressions are commoned up and loop invariants lifted from their loops to a maximal extent then this can sometimes place a very great extra burden on the register allocator, and small CSE's and invariants may end up less well treated than if they had been left in their original places. Thus perhaps register allocation and CSE analysis should be performed simultaneously. Similarly it can be unclear when to map out some composite operations into sequences of smaller ones—block moves of memory (as in C structure assignments) would sometimes benefit if expanded early (when

[6] Of course, it would be nice to get this right at register allocation time, but at that time we do not know which virtual registers will map onto storage references—this is the optimisation phase problem discussed below.

the expanded code would be subject to the full strength of the rest of the optimiser), but again the most appropriate code may depend on the results of register allocation. One solution would be to iterate the entire optimisation process, so that second and subsequent passes have the benefit of all information gathered before. In constructing the Norcroft compilers we believed that the potential gains from this were not worth the costs, and we use heuristics to guide us as to (for instance) the size and number of common expressions it will be worth trying to eliminate.

8 Language specification, the optimiser and the user

The fine print of language specifications can often be of great importance to the designers of optimising compilers. For both Fortran and C, for instance, there are sections of the relevant standard that can look most curious to the naive user (the 1966 Fortran concept of *definition at the second level* is an example) but which provide compiler builders with license to apply particular code generation techniques and optimisations. Other parts of language definitions may at first seem to be nothing more than simple description of how code should behave, but in fact imply strong constraints on the compiler. Together with the formal description of what results must be produced when programs are run there are two more issues that the compiler builder needs to consider—the user's expectations about what style of source code will lead to fast and compact object code, and the way in which programs that stray beyond the strict language standard are treated. These last two points may well weigh heavily when potential users judge the quality of a compiler.

One problem is that optimisation is not uniform—certain pieces of code optimise well as we have seen, even reducing to zero the cost of a copy or cast where two variable can share a register. However, certain other 'assumptions' based on elementary compiler technology can inhibit optimisation over a whole procedure. We will illustrate these issues with reference to ANSI C where confusions like "address-taking is free" or "`setjmp` is only a procedure call" are dispelled. Another problem is that users are poor readers of language specifications—many an optimiser has been claimed to be 'broken' because it exhibited a bug in a voluble user's code. Therefore the 'best' optimising compiler may not be the one which exploits each dark corner of the standard, but one which treats dubious constructs gently.

8.1 Aliases

A general bane for optimisers is the possibility that there may be two or more access paths referring to the same quantity, but that the compiler's analysis may not be able to detect this. Wherever there is a possibility of conflict of this sort (and where the language specification demands that behaviour be defined) a compiler needs to generate conservative code. In C potential aliases arise through the use of pointers. The most cautious view a C compiler could take is that at any time where an indirect assignment might be made (including any time a procedure is called) any variable whose address has been taken might have its value changed, and arbitrary fields of data-structures in heap (`malloc`) memory might get altered. This clearly affects available expression analysis and thus common sub-expression optimisation, but it also affects live variable analysis.

A case where this can have performance consequences that a user may not expect is when (at least in intent) the address of some variable is only used for a short while—*e.g.*

```
r = scanf("%d", &n);
```

where the address of `n` is not used elsewhere. A pessimistic (*i.e.* correct) C compiler might see that the address of `n` is passed to an external procedure, which could potentially save it or make it generally available. The compiler would then conclude that safe code would keep `n` in memory at all times. A user aware of this problem might write

```
{ int tempn; r = scanf("%d", &tempn); n = tempn; }
```

so that the more widely used variable `n` was not blighted. A cleverer compiler could only really improve things if the standard header containing the declaration of `scanf` had also provided (in some system-specific way) a promise that `scanf` would not retain the pointer it was passed, and that it does not do any indirect assignments other than those through the explicit addresses provided to it. This itself carries a further message to users—not only should all functions be properly declared with the correct types before they are called, but the declarations in standard headers may provide the compiler with yet further information about side effects that library functions may have, and that information may be important to the optimiser.

8.2 `setjmp` and `longjmp`

On computers such as the VAX and with simple compilers such as the original pcc (Portable C Compiler) `setjmp` could work by just saving a stack pointer and a return address in the `jmp_buf` and `longjmp` could achieve all it needed by loading a couple of machine registers and unwinding the run-time stack using information stored there not strictly necessary for the non-`longjmp` execution.

However, `setjmp` requires complicated analysis in an optimiser: when building a flow-graph of a body of code that contains an instance of `setjmp`, every procedure call which might (indirectly) invoke `longjmp` must be treated as a conditional branch to the logical label immediately following any preceding (dominating) call of `setjmp`. This immediatly inhibits tail-recursion optimisation. The C standard permits some variables to be left in an indeterminate state after `longjmp`, but demands that others are preserved. The effect is that the presence of a `setjmp` has a global effect on the graph used to analyse the code, and will in general lead to many more values having to be kept in memory rather than in registers. Given this it is just as well that ANSI C demands that the standard header declaring `setjmp` be included if this facility is to be used, and that it does not permit the function `setjmp` to be stored or passed around as a procedural argument—these arrangements make it possible for a conforming compiler unambiguously to identify uses of `setjmp`.

Hence `setjmp` can be expensive for two reasons on modern architectures. Firstly, stack frame optimisations can leave insufficient information for `longjmp` to restore the old context from the stack and hence `setjmp` must save many registers (*e.g.* 32 integer and 32 floating). Moveover, its mere presence in a routine can require seriously reduced optimisation (*e.g.* a factor of two or more in the speed of an interpreter). The moral is to use `setjmp` in a veneer procedure which calls another procedure to do any significant work and to make sure it is not executed unnecessarily often.

8.3 Register declarations

From early days C compilers have made it possible to keep especially heavily used values in registers, via an explicit declaration. The proper status of this declaration on compilers for machines with large numbers of registers and compilers that allocate these using advanced techniques has become obscure. It seems that if optimising compilers treat user register declarations as firm instructions about how to treat certain variables and the user has provided too many such declarations the effects can be fairly bad, with most other quantities spilt to the stack. In the majority of cases the variables that need special attention will be looked after well by automatic register allocation anyway. We consider that the realistic choice available to compiler builders is between ignoring register declarations totally and using them to give (slight) weight in choices when spilling is needed. We favour the latter.

8.4 Sequence points

There are a number of ways in which (especially) optimising compilers can follow a language specification accurately but end up surprising programmers who have not read the standards documents carefully enough. In these cases the compiler designer needs to decide whether it is possible to detect the relevant cases and issue a compile-time warning; and whether performance or support for plausible but faulty code is more important. Consider, for instance, the expression `f(*x++, *x++)` supposing that the function `f` calculates some symmetric function of its two arguments. Then C standard says that the behaviour of this code is undefined, which certainly *permits* a compiler to turn the above example into a call to a library function that deletes all the user's files. The possible ways in which the individual parts of the two `*x++` operations (loading `x`, incrementing the value, storing it back and doing an indirect fetch based on the original value of `x`) can interleave during optimisation mean that the code may compile so as to give several different effects—and the behaviour may be depend on the surrounding code context (even without making full use of the standard's license to treat it as undefined). Our view about such problems is that it behoves aggressively optimising compilers to include as comprehensive 'lint'-like checking as they can and, in the case of code that may behave other than as the user expected, either to generate an explicit warning or to back off and degrade optimisation. Current Norcroft compilers do not exploit the full power offered by the standard and behave *as if* only whole sub-expressions are re-ordered but never interleaved—this gives good optimisation with greatly reduced chance of upsetting users.

8.5 `char` and `short` variables

In C it is possible to declare variables of type `char` and `short`. When arithmetic is performed, C requires that such short types are widened to (possibly `unsigned`) `int`.[7] The effect of this is that variables of type `char` and `short` (and to a lesser extent `float`) have a rather anomalous character which can complicate the job of optimising compilers. To cope with this problem our compiler keeps such (non-address taken) variables in memory or registers widened to the natural size of an integer. We initially generate pessimistic 3-address code that widens values to `int` before storing them,[8] but the optimiser is then very often able to remove these extra instructions. An example showing why the masking may sometimes be needed is

```
int f(char *x, char y) { *x = y+1; }
```

[7] Prior to the ANSI standard it also was understood that if any arithmetic was to be performed on a `float` it would first be widened to `double`.

[8] We find it produces better code to widen such `char` and `short` variables on storing rather than on loading.

where the value returned must have been reduced to the range of character values. The extra zero- or sign-extension code that `char` and `short` variables sometimes call for can make them lead to slower code than if just `int` were used—this does not always match user expectations.

References

[1] Aho, A.V., Sethi, R. and Ullman, J.D. Compilers: principles, techniques and tools, Addison-Wesley, 1985.

[2] Auslander, M. and Hopkins, M. An overview of the PL.8 compiler, Proc. ACM symp. on compiler construction, 1982.

[3] Fischer, C.N. and LeBlanc, J.R. Crafting a compiler, Benjamin/Cummings, 1988.

Space considerations preclude a fuller bibliography, but the books [1, 3] cover much of compiler design and have good chapters on optimisation together with comprehensive references. The two annual ACM symposia on "Compiler Construction" and "Principles of Programming Languages" often have important articles including the very readable PL.8 compiler work by Auslander and Hopkins [2]. Of course, all C programmers should have easy access to the ANSI standard document X3J11 which defines the language instead of relying on habits learned from their usual implementations.