Global Register Allocation at Link Time

David W. Wall

Digital Equipment Corporation
Western Research Lab

Abstract

In previous work in global register allocation, the compiler colors a conflict graph constructed from liveness dataflow information, in order to allocate the same register to many variables that are not simultaneously live. If two procedures are in separately compiled modules, however, the compiler must do this allocation separately for each procedure. As a result, the two procedures might use different registers for the same global, or the same register for different locals.

We can remove these problems if we delay the register allocation until link time. Our compiler produces object modules that can be linked and run without global register allocation, but includes with each object module a body of information describing how the module uses variables and procedures. A link-time register allocator then decides which variables are used most frequently, selects registers for them, and rewrites the code to reflect the decision that these variables reside in registers rather than in memory. Construction of the call graph allows us to use the same register for locals of procedures that are not simultaneously active, giving us most of the advantages of a full-scale coloring without the expense.

When we use our method for 52 registers, our benchmarks speed up by 10 to 25 percent. Even with only 8 registers, the speedup can be nearly that large if we use previously collected profile information to guide the allocation. We cannot do much better, because programs whose variables all fit in registers rarely speed up by more than 30%. Moreover, profiling shows us that we usually remove 60% to 90% of the loads and stores of scalar variables that the program performs during its execution, and often much more.

1. Introduction

Several recent machines [8,9,10] have been designed with simple instruction sets. They have only a few ways of accessing memory, sometimes no more than a single load instruction and a single store instruction. This allows most operations to be fast, since they operate only on registers. In addition, these machines often have a large number of registers, relying on smart compilers to exploit them.

How can we best take advantage of a machine with many registers? One promising idea is to pick the most frequently used variables and keep them in registers instead of in memory. Chaitin et al. [2,3] and Chow

An earlier version of this paper appears in the Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, published as *SIGPLAN Notices 21,* 7, pp. 264-275,

[4] have explored a technique in which many different variables can be assigned to the same register. In their technique, the compiler does a liveness dataflow analysis, and builds a conflict graph in which an edge appears between two variables if they are ever live at the same time. Two variables that are never live at the same time can be kept in the same register, so a coloring of this graph is equivalent to a register allocation for the variables. Good linear-time coloring heuristics exist, even though the problem of finding a minimal coloring is NP-complete.

The technique of Chaitin et al. and Chow is less satisfactory if we also want to use separate compilation. If procedures in two different modules must be analyzed separately, then there is no way to keep their register allocations from interfering with each other. If one procedure calls the other, one of them must save and restore its locals around the call, in case the other procedure uses the same registers for its own locals. If both procedures use a common global variable, they may allocate different registers for the global, so each procedure must spill the global to a home memory location before transferring to the other, and load the global from that location whenever it gets control again.

These problems are less important in an environment where the modules of a program are compiled all together instead of separately, or in machines with few registers. The papers by Chaitin et al. seem to assume such an environment, and the first of these papers describes a machine with only 16 general purpose registers. Both of these assumptions seem unrealistic, however: the machine we are working with has 64 registers, and separate compilation seems essential if we want to develop large programs.

Our solution is to assign variables to registers at link time.

2. Global register allocation at link time

If we assign variables to registers over the entire program at link time, these problems disappear. We can keep each global in a specific register where any procedure can access it. And when one procedure calls another, we can keep the locals of the two procedures in different registers, so that there is no need to save and restore them around the call.

Simply postponing the coloring technique until link time is impractical, however. If we wait until link time and then do an expensive dataflow analysis, conflict graph construction, graph coloring, and final code generation, we will be doing the bulk of the compiler's work at link time, and recompilation after a change will take almost as long as if we did not have separate compilation. Moreover, algorithms for dataflow analysis and conflict graph construction require slightly more than linear time; the cost may be acceptable if we apply the algorithms repeatedly to a lot of smaller modules or procedures, but we have found it too expensive to apply them to an entire program. It is important to do as much as we can at compile time.

We therefore treat the assignment of variables to registers as a form of relocation. The compiler generates code that can be linked and executed without further work, but it includes extra information similar in purpose to the information in the relocation dictionary. If the global register allocator is invoked, it decides which variables should be kept in registers, and then uses this extra information to rewrite each module based on

the allocation, just as a relocating linker rewrites a module in order to relocate it.

Generating code that is correct as it stands but which we can easily modify later is attractive for three reasons. First, it means that the assignment of variables to registers is entirely optional. If the expense of the assignment is unwarranted, or if bugs are suspected, we can omit it. Second, it means we are not forced somehow to allocate registers for all the variables in the program; if we manage to keep the most frequently used variables in registers, we can keep the rest of them in memory and the results are still correct. Finally, it keeps us honest as designers of the system; once we postpone anything until link time, the temptation is great to postpone everything, so that we can know what the other modules look like. The requirement that we generate working code at compile time makes it harder to succumb to that temptation when we should not.

Making the assignment of variables to registers optional means that the compiler must set aside some registers as expression temporaries, and deal with them separately from the registers allocated at link time as the homes of variables. This is contrary to the philosophy of Chaitin et al., which included expression temporaries in the conflict graph, and introduced spills and reloads in order to reduce the complexity of the conflict graph to the point where it was colorable using the number of registers available. This meant that global register allocation was an integral part of code generation and could never be omitted. We contend that this is inappropriate in a realistic setting with the need for separate compilation and optional optimization.

Our scheme for assigning variables to registers consists of three phases: compilation, register allocation, and module rewriting.

2.1. Compilation

The compiler must do two things to support global register allocation. First, it must annotate the code it generates so that it can be rewritten after registers have been allocated. Second, it must collect usage information that lets the allocator build the program call graph and estimate the usage frequency of each variable.

2.1.1. Annotated code generation

Our target machine allows memory access only through a simple load or store instruction. The compiler generates good code that does not load a variable twice in the same basic block unless necessary, but it makes no attempt to allocate variables to registers. Moreover, it uses only eight registers for expression temporaries; if it needs more, then it spills some of these eight into internal memory variables which are themselves candidates for global register allocation. In short, the compiler generates the code that you would expect if the machine had only a few registers.

The compiler annotates some of the instructions with ‘‘register actions’’ that explain how the operands and results are related to the candidates for global register allocation. These candidates include scalar user variables, numeric constants, and the addresses of data structures and procedures. For example, the assignment ‘‘x = y + z’’ would lead to annotated code like this:

instruction	*actions*
r1 := load y	REMOVE.y
r2 := load z	REMOVE.z
r3 := r1 + r2	OP1.y OP2.z RESULT.x
store x := r3	REMOVE.x

Each action in this example is qualified by a variable name; the action specifies what needs to be done to the instruction if that variable is assigned to a register at link time. The action REMOVE.v means to delete the instruction if v is assigned to a register. The action OP1.v, OP2.v, or RESULT.v means to replace the first operand, second operand, or result of the instruction by the register allocated to v. The three examples below show the rewritten code that results at link time from various selections of register variables.

register y	*register x,y*	*register x,y,z*
r2 := load z	r2 := load z	x := y + z
r3 := y + r2	x := y + r2	
store x := r3		

This example was easy. Our job is more complicated if a variable is evaluated and then changed, and later the original value is used. For example, the C assignment ‘‘x = y++ + z’’, which increments y but adds the unincremented value to the value of z, might lead to annotated code like:

r1 := load y	LOAD.y
r2 := r1 + 1	RESULT.y
store y := r2	REMOVE.y
r2 := load z	REMOVE.z
r3 := r1 + r2	OP2.z RESULT.x
store x := r3	REMOVE.x

The LOAD.y action says that the instruction is loading the current value of y, but that y will be changed before this value is used for the last time. As a result we cannot delete the load; instead we change it to a register move that copies the current value of y from its home register into the temporary register. For the same reason, we do not flag later uses of this value with an action like OP1.y, as we did in the previous example. If x, y, and z are all assigned to registers, this code segment becomes:

```
r1 := y
y := r1 + 1
x := r1 + z
```

We will see that distinguishing between these two cases is not difficult with a little analysis of the basic block.

Some source languages allow an assignment to be used as an expression. In this case, the computed value is used as an operand to more than one command, not just as an operand to the assignment. The variable to which the value is assigned may even change again before we finally use the computed value. This makes it hard to use the RESULT action to combine the computation with the assignment. So we let the computed value remain in the temporary register. The C assignment ‘‘x = y = a + b’’, which assigns the sum of a and b to y and then to x, might lead to annotated code like:

```
r1 := load a      REMOVE.a
r2 := load b      REMOVE.b
r3 := r1 + r2     OP1.a  OP2.b
store y := r3     STORE.y
store x := r3     STORE.x
```

A STORE.v action is analogous to a LOAD action. It says that we should replace the store instruction by a register move into the register allocated to v.

There is one degenerate case that we must handle properly. If we compile the assignment ‘‘x = y’’ the above examples would lead us to expect the annotated code:

```
r1 := load y      REMOVE.y
store x := r1     OP1.y  REMOVE.x
```

If we assign both x and y to registers, both instructions will be removed! We must therefore handle simple assignments of one variable to another as a special case, which we can do by changing the annotation of the store:

```
r1 := load y      REMOVE.y
store x := r1     OP1.y  STORE.x
```

How do we decide how to annotate the code in practice? Let us represent a basic block as a sequence of commands, where each command specifies the evaluation of a variable, the performance of an operation, or the assignment of a result. Then the C assignment ‘‘x = y++ + z’’ would appear as:

```
#1:   leaf      y
#2:   operate   #1 + 1
#3:   assign    y := #2
#4:   leaf      z
#5:   operate   #1 + #4
#6:   assign    x := #5
```

There are three types of commands. A leaf command evaluates a single variable. An operation command performs an operation using the values of previous commands, producing a new value. An assignment command assigns the value of a previous command to some variable. This organization of commands is appropriate for a machine like ours where we must access memory only via loads and stores. Other kinds of commands, such as indirect loads and stores and procedure calls, do not concern us here.

Then generation of unannotated code is easy. To annotate the code with register actions, we must first determine where the value of each command is used; for instance, the value of command #1 above is used in commands #2 and #5, but not thereafter. Command #1 evaluates the variable y, and command #3 is an assignment to y, so the use of command #1 in command #5 means that we must mark #1 as ‘‘time critical,’’ meaning that the value used is not always the current value of the variable.

We can find these cases with a backward pass over the commands, as follows.

```
Let LIVELEAVES be a set of leaves in this
        basic block, initially empty.
for each command, in reverse order, do:
    if the command is an assignment, then
        let DEST be the assignment destination
        for each leaf in LIVELEAVES, do:
            let LEAF be the leaf variable
            if LEAF is the same as DEST, then
                Mark the leaf as ''time critical''
            end
        end
    end
    if the command is a leaf, then
        remove it from LIVELEAVES
    else
        for each operand of the command, do:
            if the operand is a leaf command, then
                add the leaf command to LIVELEAVES
            end
        end
    end
end
```

This done, we generate annotated code for each command as follows.

Case 1. If the command is a leaf for some variable v, then generate a load into a temporary register. (Remember that expression temporaries are different from the registers we allocate to variables.) If the leaf is marked ''time critical,'' then annotate the load with LOAD.v, and otherwise annotate it with REMOVE.v.

Case 2. If the command is an operation, generate the instructions to perform the operation. Usually this is just one instruction, but for operations that are not available in hardware it may be a sequence of instructions. If the value of this operation is used only once, by an assignment command ''v:='', then annotate the instruction that produces the result with RESULT.v.

Case 3. If the command is an assignment to some variable v, generate a store. If the operand is a leaf command, or if it is used in some other command as well as this one, then annotate the store with STORE.v. Otherwise annotate it with REMOVE.v.

In either case 2 or case 3, if an operand of the command is a leaf for some variable v and is not marked as ''time critical,'' annotate any instructions that use that leaf command with OP1.v or OP2.v, depending on which operand of the instruction it is.

There are really only two things going on here, each of which consists of two pieces that must fit together properly. First there is the loading of a variable. If the value is used only in places where the variable still has the same value, we can mark the load for removal and mark the uses to be replaced by the variable's register. If there are uses after the variable has changed, then we must retain the load (but replace it by a register move) and leave the uses unchanged. Second there is the assignment of a value to a variable. If the value is used only to assign it to the variable, we can mark it for direct computation into the variable's register, and mark the store for removal. If the value has other uses, we must retain the store (again replacing it by a register move). A smarter analysis would let us see whether the other uses could instead be replaced by the variable's register, but we don't do that.

2.1.2. Usage information

The compiler collects usage information about each module it compiles, and records this information in the object file. This information includes a list of the procedures in the module. For each procedure there is a list of the variables local to this procedure, a list of the procedures this procedure calls, and a list of the variables this procedure references. Each entry in the last two lists includes an estimate of the number of times the procedure is called or the variable is referenced in each execution of this procedure. When I obtained the results described later in this paper, these estimates were computed by counting the static number of times the reference is made, with references inside loops, no matter how deeply nested, counting as 10.

2.2. Register allocation

Before the program is linked, the register allocator collects the usage information for all the modules being linked, and then builds a call graph for the program. This call graph is a directed acyclic graph (DAG) and does not include edges for recursive or indirect calls, which require special handling.

The allocator then estimates the total number of times each procedure is called, by summing the counts associated with the procedure in all of the call lists. (At first we tried to traverse the call graph and multiply counts, so that if P called Q ten times and Q called R ten times we ended up with an estimate of 100 calls for R. Since the estimates in the individual call lists were only rough guesses, this tended to give wildly wrong counts, some even overflowing the 32-bit integer. The current more conservative scheme seems to work better.)

The allocator also estimates the total number of times each variable is referenced, by summing the reference counts for each procedure, weighted by the estimated number of times the procedure is called.

The idea behind our register allocation scheme is that locals of procedures that are not simultaneously active can be grouped together and assigned to the same registers.* We do this by traversing the call DAG in reverse depth-first search order [1], assigning the locals of leaf procedures to a series of groups beginning with

* Michael L. Powell is responsible for this insight, for which I am most grateful.

group 0, and then assigning locals of other procedures to a series of groups beginning after the groups used by its children. The algorithm[†] is as follows:

```
for each proc p in reverse depth-first search order, do:
    (*  Each child was given groups in a previous
            iteration.  Leaves have no children.  *)
    childGroups := 0
    for each child q of p in the call DAG, do:
        if groups[q] > childGroups then
            childGroups := groups[q]
        end
    end
    groups[p] := childGroups
    for each local v of p, do:
        assign v to group number "groups[p]"
        groups[p] := groups[p] + 1
    end
end
```

The register allocator then assigns each global variable to a singleton group, gives each group a frequency that is the sum of the reference frequencies of its variables, and sorts the groups by frequency. Then it assigns the most frequently used groups to registers.

A variable whose address is taken anywhere in the program is ineligible for assignment to a register. This is an extremely conservative decision, which could be relaxed by using dataflow analysis to determine the places where the address might be used. We were interested in how well we could do without that sort of analysis, so we didn't do it.

2.3. Module rewriting

When we have decided which variables to keep in registers, we know which register actions to apply to the code of each module. Applying these actions is easy, since each one is independent of context; the compiler did the hard work when it generated the actions in the first place. The only tricky part is that we may add or delete instructions, so addresses that appear in instructions or data must be adjusted. Finding these instructions or data is easy as well, because the compiler flags them for relocation by the linker.

The module rewriter starts by building an adjustment table that it uses to map old addresses into new ones. This table specifies the adjustment that we must make to addresses in a given range; to find the range for a code address we do a binary search. Data addresses are easier: we always adjust them by the change in the

[†] My thanks to Richard Beigel for this elegant algorithm.

size of the code.

Then the rewriter steps through the code segment, modifying or discarding each instruction according to the register actions that apply to it. If the instruction has a relocation entry, the rewriter discards the relocation entry if the instruction is removed or if relocation is no longer needed, and modifies the relocation entry by adjusting the address it applies to if the instruction is retained but has a different address than before. We rewrite the data segments in much the same way; there are no actions to apply, but a data item itself may be flagged for relocation as some kind of address, in which case the value must be adjusted.

We rewrite the linker symbol table in exactly the same way. Each symbol has a value and a type; the type tells us how to adjust the value.

The rewritten module is then returned to the linker, to be combined with other rewritten modules.

2.4. Complications and Extensions

When I implemented the scheme described above, I encountered several complications and several opportunities for extensions. My implementation handles all of these.

2.4.1. Initialization

Some variables have initial values that we must consider in our global register scheme.

A global variable may be declared with an initial value. This value normally appears in the core image produced by the loader, so no instruction is executed to assign the value to the variable. If we decide that some variable will reside in a register rather than in memory, we must somehow get that initial value from the home memory location into the register.

Identifying all such globals is easy for the compiler, so it includes information about these globals with the usage information. The driver program, which is responsible for initializing the program environment and invoking the user main program, includes an action called INIT. The INIT action is unconditionally performed by the module rewriter. This action requests that instructions be inserted to copy the initial value into each initialized global that has been assigned to a register.

Local variables declared with initial values do not present this problem, since the compiler generates ordinary code to assign their initial values at the beginning of the procedure. This code is annotated just like code for explicit user statements.

Parameters, however, do remain a small problem. To avoid copying argument values if register

allocation is not requested, our calling conventions require that the calling procedure put arguments on the stack in locations that will correspond to the parameters of the called procedure. If the register allocator decides to assign one of these parameters to a register, the module rewriter must insert code at the beginning of the procedure to load the argument into that register from the stack. The compiler specifies another register action to mark the place where we must generate these loads.*

2.4.2. Recursive and indirect calls

If the program contains recursive calls, we must do something to maintain the correctness of our scheme. Our approach is potentially expensive, but not extremely so in typical cases. And recursive calls are a small fraction of the total, especially when tail recursion is removed.

What is the problem with recursive calls? Our aim was to allocate registers so that when we call procedure P, the registers allocated to P's locals are not currently in use, so we don't have to save and restore these registers. This obviously cannot be true in a recursive call, because the set of variables local to P are already in use, in a previous invocation of P. If we kept all of our variables in memory, we would handle this by creating a new frame on the stack. If we are going to use the registers for the new instances of these locals, we must archive the old values. This archiving is enough to let P operate correctly, since the new invocation of P cannot access the locals of the previous invocation.

One way to organize the archiving of recursively instantiated locals is to have each recursive procedure be responsible for archiving its own local registers. This unfortunately is not consistent with our algorithm for combining non-conflicting locals into groups, as shown by the following call graph:

Here B and C are leaves of the call DAG, because the recursive call from B to A does not appear in the DAG. Thus it looks like we can use the same group of registers for the locals of B and C; but this is wrong because B and C can be simultaneously active after all, if B calls A and A calls C. If we made each procedure responsible for archiving its own locals on entry, we would make the following error. The call to B would not archive the locals of C because C is not part of any recursive chain. The call from B to A would archive the locals of A, but not those of B and C. But this means that B and C would use the same registers without archiving them in between. So we need a better method.

* In fact, we can do better, by having the caller put the argument value directly into the parameter register. We discuss this in section 2.4.3.

We could change the way we group locals, or we could change the way we do archives. On the assumption that recursive calls are much less frequent that non-recursive calls, we chose the latter. Instead of each procedure in a recursive chain saving its own local registers, the procedure that makes the recursive call saves the locals of all the procedures in the chain. This occasionally saves some things unnecessarily, as in the example:

Here the recursive call from D to A saves the locals of both B and C, even though only one group of locals can possibly be active. This example is typical, however, in that the locals of B and C are assigned to the same group of registers. Moreover, many apparently recursive programs are recursive only in exceptional cases; this solution allows us to call such a routine in the normal case without archiving its locals unless we do an actual recursive call.

The same problem occurs with indirect calls, in which a procedure is called though a procedure variable, so that we cannot tell at compile time which procedure is being called. Since such a call cannot appear in the call graph, we cannot take it into account when we allocate registers to locals.

It turns out that we can apply the same solution here. An indirect call archives *all* of the apparently active locals, so that when we arrive at the called procedure we can behave as if we had performed a normal call visible in the call DAG.

You should note that this works only because the languages we are dealing with allow indirect calls only to top-level procedures. If P and Q are both nested in R and both refer to R's locals, and P calls Q indirectly, archiving R's locals around the call is wrong, because Q might legally change one of them. We could still apply the technique by postponing it to run time; the procedure variable could be implemented as a structure that includes information about what needs to be saved. This is not too different from current implementations, in which the procedure variable includes a display that describes the environment in which to call the procedure.

2.4.3. Fast argument passing

The scheme we have described so far deals with argument passing in a simple-minded way. We evaluate each argument and place it on the stack in the memory location associated with the parameter that we are about to create. If the called procedure has a register parameter, it loads the argument from the stack into that

register. It would be better if the calling procedure could put the argument directly into the parameter register to begin with.

This would not be possible if we were doing global register allocation at compile time, since the called procedure might be in a different module and we would not know which of its parameters would be assigned to registers. Since we are doing global register allocation at link time, however, it is straightforward to include this optimization.

We use the same idea here that we used in expression evaluation. The compiler generates code that stores the argument on the stack, but flags the store with an action PARM that tells which procedure is being called and the index of the parameter being passed. (It cannot refer to the parameter by name because it generates the action at compile time, when the called procedure may not be visible.) At link time, the module rewriter checks to see whether that parameter is assigned to a register, and if so converts the PARM action into an ordinary STORE action.

In the section ''Initialization,'' we described the insertion of loads into the procedure entry code, to load the arguments from the stack into the parameter registers. Now we must omit these loads, since the caller has already put the arguments directly into the parameter registers, and the value on the stack is garbage. However, there remains the possibility that someone will call the procedure indirectly, in which case this optimization cannot be applied. So we leave the inserted loads in place, and instead modify the call so that it enters the procedure after the point at which we have inserted the loads.

Proponents of the Berkeley RISC machine [9] are justly proud of the floating register windows that allow arguments to be passed in registers. It is satisfying to see that we can accomplish much the same thing without hardware support or runtime overhead. Moreover, letting the compiler and linker take care of it is more flexible, since we can decide for each program how many registers to use for globals and for locals of each procedure.

2.4.4. Profiling

We do not keep all of the variables in registers, but only those that seem to be referenced frequently. As a result, the quality of the estimates of the frequency with which variables are referenced and procedures are called is often important, especially in large programs. These estimates can be improved by using previously gathered execution-time profile information. We have two kinds of profilers that we can use with the register allocator.

The first is a variable-reference profiler that we built ourselves for precisely this purpose. The compiler prints statistics about removable loads and stores in each basic block, and generates code that counts the times each basic block is executed. These two sets of information are combined to give us the number of executions of loads and stores that could be removed by assigning each variable to a register. This will normally be a very good estimate of the amount by which assigning that variable to a register would make the program faster, although it does not consider changes in the behavior of the instruction cache that result from removed and

inserted instructions, and it does not consider the time spent archiving registers for recursive or indirect calls.

The variable-reference profile has the disadvantage that we may need to recompute it often for it to be most effective. If the programmer adds a new variable to a procedure that is executed very frequently, that variable will not appear in the previously computed profile and therefore is unlikely to get assigned to a register. An alternative is to use gprof [6] to get actual counts of run-time procedure calls, and then combine these with the compile-time estimates of variable usage by each procedure, which are computed every time the module is compiled. The result is a profile that is somewhat less precise but also somewhat less sensitive to small changes in the program.

2.4.5. Local coloring

In grouping together local variables that we can assign to the same register, the only liveness information that we use is the fact that a local variable cannot be live except when its procedure is active. This is reasonable because more precise liveness information requires expensive intermodule dataflow analysis, and because the lifetimes of most locals are short anyway. This simplification does have one possibly serious limitation, however: it does not allow us to combine two non-conflicting locals of the same procedure.

Recognizing two such locals does not require interprocedure dataflow analysis, but only the local dataflow analysis used by Chaitin et al. and Chow. We made this technique an optional addition to our own. If requested, the compiler does a liveness analysis on each procedure it compiles, grouping together locals that do not conflict. Of course, it cannot allocate registers to these groups without interfering with the link-time register allocator, so it simply reports these groups to the link-time allocator.

The only tricky part is that the liveness analysis may allow a parameter to be combined with another local; if so, the actions associated with procedure call must make sure that the argument value gets properly loaded into the register that the two variables share. More subtly, we may combine one parameter with another parameter whose argument value is never used; then we should not load the unused argument into the shared register. The remaining case is two parameters whose arguments are both used; here the liveness information by itself prevents us from combining them.

We could also use this liveness dataflow information to keep track of which locals are live when their procedure calls another procedure. This would let us improve the allocation, since we might be able to keep two locals in the same register even if they are local to procedures that are simultaneously active. We didn't do this, however, because we expected that coloring would remove the need for this: there might be locals that are not live at the call, but there would usually not be registers that are not live. If we are going to incur the expense of the dataflow analysis, we might as well go ahead and do the coloring.

3. Results

I implemented this global register allocator in a code generator used for Fortran, C, and Modula-2, and my colleagues have used it to port the Unix* operating system to the new machine. To learn how well it works, I applied several kinds of allocation to six benchmarks: the Livermore Loops, the Whetstones benchmark, the LINPACK linear equation benchmark [5], the Stanford suite collected by Hennessy [7], the logic simulator RSIM [11], and a timing verifier.

	lines	*vars*	*groups*
Livermore	268	166	165
Whetstones	462	254	181
Linpack	814	214	119
Stanford	1019	402	211
Simulator	3003	811	262
Verifier	4287	1395	693

Table 1. The six benchmarks.

The six benchmarks ranged in length from 268 to 4287 lines of user source code, and the number of variables ranged from 166 to 1395. These variables were the candidates for assignment to registers, and included local user variables, global user variables, and global constants such as numbers and the addresses of arrays and procedures. The third column of Table 1 shows the number of non-conflicting groups the allocator combined these variables into; if we had this many registers then we could assign every variable to a register without causing any conflicts.

	register allocation	*with coloring*	*with profile*	*with both*
Livermore	18%	18%	19%	19%
Whetstones	10%	10%	10%	10%
Linpack	13%	13%	13%	13%
Stanford	25%	25%	27%	28%
Simulator	12%	14%	15%	16%
Verifier	10%	15%	16%	19%

Table 2. Percentage improvement in speed with 52 registers allocated to variables.

I applied the register allocator to these benchmarks in four different ways and compared the speeds of the resulting code to the speed of the original code with no global register allocation. Table 2 contains the

* Unix is a trademark of Bell Laboratories.

percentage improvements when we compare speed with allocation to speed without allocation. In this table I allowed the allocator to use 52 registers. These 52 registers count only those used for allocating variables, and not the expression temporaries or addressability registers managed by the compiler. Global register allocation speeded up the benchmarks by 10 to 25 percent over code without it. These improvements apply to a machine with a one-cycle memory cache; they would be better still for a machine on which the penalty for memory access was greater.

The first column of Table 2 contains the improvement when we did register allocation using the compile-time usage estimates. The compile time usage estimates include recognition of loops.

The second column of Table 2 contains the improvement when we compiled the program with local dataflow analysis and coloring, as described in section 2.4.5. Because this allowed us to combine locals of the same procedure with each other as well as with locals of non-conflicting procedures, the allocator was able to keep more of the variables in registers. The result was invariably faster, but it was noticeably faster only for the larger programs.

The third column of Table 2 contains the improvement when the usage estimates came from a previously collected profile instead of the compile-time estimates. This tended to make a larger difference than local coloring did, but again the difference is noticeable only for the larger programs.

The fourth column of Table 2 contains the improvement when we used both local coloring and the profile.

	register allocation	*with coloring*	*with profile*	*with both*
Livermore	80%	81%	93%	94%
Whetstones	73%	75%	84%	88%
Linpack	95%	96%	99%	99%
Stanford	90%	92%	97%	98%
Simulator	73%	83%	92%	95%
Verifier	52%	61%	78%	83%

Table 3. Percentage of removable dynamic memory references that were actually removed.

The profile provides the number of executed loads and stores that would be removed if each variable were assigned to a register. If we compare this profile to the list of variables actually assigned to registers, we can see how many of the removable loads and stores we actually removed. The fraction is surprisingly large, even for large programs. Table 3 shows the percentage of the removable executed loads and stores that we removed when 52 registers were used. (This is not a fraction of *all* loads and stores executed, but only of those that we can remove by keeping some scalar variable in a register instead of memory.)

There are two reasons that so many of the removable loads and stores got removed. First, these are *dynamic* loads and stores, actual counts from the execution of the program, and we give priority to those variables that seem to have a large number of run-time references. Second, we look at the call graph to discover non-conflicting procedures and we keep their locals in the same group of registers. I can illustrate the importance of both these effects by considering the timing verifier, the largest of the benchmarks. The usage frequencies in the profile range from 0 to 3257639. In the allocation without coloring or profiling, we kept 505 distinct variables in the 52 registers. If we had assigned only the 52 variables with the highest estimated frequency, we would have removed only 25% of the loads and stores instead of 52%.

The number of loads and stores removed will usually be a good estimate of the speed improvement (measured in number of instructions executed), though it neglects the effects of cache faults and the loads introduced to initialize globals, to pass arguments in indirect calls, and to archive locals for recursive or indirect calls. Benchmarks that are small enough that all the variables fit in 52 registers rarely improved by more than 30%, and often by much less. In this light it is unsurprising that there are so few removable loads and stores left in these programs.

These results also suggest that we are getting a benefit from global register allocation that is comparable to the benefit we would get from any global register allocation scheme. Even if a better scheme could somehow manage to keep *all* of the variables in 52 registers without introducing spills and reloads, it would not be able to double any of the improvement percentages in Table 2, and usually its advantage over our scheme would be even smaller.

	Data cache speed in cycles				
	1	*2*	*3*	*4*	*5*
Simulator	12%	19%	24%	27%	29%
Verifier	10%	15%	19%	21%	23%

Table 4. Percentage improvement in speed with 52 registers allocated to variables, for different data cache speeds.

If a machine has a slower data cache, the penalty for memory references is larger and therefore so is the improvement given by global register allocation. I combined the profile information with information produced by our machine simulator to determine how large the improvements would be if the data cache were slower. Table 4 shows these results. For this table, registers were allocated for the two largest benchmarks without using compile-time coloring or profiling, and data cache speeds between one and five cycles were considered. Our machine has a one-cycle data cache, so the first column is a direct result rather than a computed one, and was seen earlier in the first column of Table 2.

	register allocation	*with coloring*	*with profile*	*with both*
Livermore	16%	17%	18%	18%
Whetstones	7%	8%	9%	10%
Linpack	10%	13%	13%	13%
Stanford	24%	24%	26%	27%
Simulator	9%	13%	14%	15%
Verifier	9%	11%	12%	16%

Table 5. Percentage improvement in speed with 32 registers allocated to variables.

Tables 5 and 6 show how well the link-time allocation does with fewer registers. Naturally it does not do as well, but substantial increases in speed can occur with as few as 8 registers. It is interesting how well the compile-time estimates do when few registers are available. Using local coloring or profile information often improves the result, however, and should be considered a common or even normal part of global register allocation when few registers are available on the target machine.

	register allocation	*with coloring*	*with profile*	*with both*
Livermore	10%	10%	12%	12%
Whetstones	3%	3%	5%	5%
Linpack	2%	7%	8%	10%
Stanford	15%	16%	18%	20%
Simulator	3%	6%	8%	8%
Verifier	2%	3%	5%	7%

Table 6. Percentage improvement in speed with 8 registers allocated to variables.

I also allowed the register allocator to simulate compile-time global register allocation. In this compile-time method we look at only one procedure at a time. We keep each local of that procedure in a register if it seems to be used more than twice, and save these registers when the procedure begins and restore them when it returns. We do not keep globals in registers, and we pass parameters on the stack.

	link-time allocation	*compile-time allocation*	*compile-time with coloring*
Livermore	18%	12%	12%
Whetstones	10%	1%	2%
Linpack	13%	10%	10%
Stanford	25%	19%	20%
Simulator	12%	10%	11%
Verifier	10%	4%	7%

Table 7. Percentage improvement in speed for link-time allocation compared to the improvement for compile-time allocation.

Table 7 compares link-time allocation using 52 registers without coloring or profiling (the first column of Table 2) to this compile-time method. Link-time allocation does substantially better.

The compile-time method can also include local coloring, and the combination should be close to what you would get if you were to implement the method reported by Chaitin et al. and Chow so that it supported separate compilation. The third column of Table 7 shows the improvement that this method gives us. With 52 registers, our method of link-time allocation without dataflow analysis and coloring is a clear winner over the compile-time local coloring method.

4. Conclusions

Link-time register allocation based on the call graph is a reasonable alternative to compile-time allocation based on dataflow analysis and graph coloring. It compresses variables into registers about as well, it allows procedure calls to happen without saving and restoring local register variables, and it allows us to include global variables in the allocation. If good usage estimates are used, obtained from a profiler or a good static estimator, speed improvements of 10 to 25 percent are possible even if only a few registers are used.

Another important contribution of this work is the idea of generalizing relocation information to let us cheaply postpone important decisions until link time. It is also an effective way to make studies of the sort described here possible; for example, we added the simulated compile-time allocation described in Table 7 quite late in this study, but we did it merely by applying different link-time interpretations to some of the register actions, with no change to the compiler required. We expect this idea to be useful in other areas as well.

5. Future work

There are several things we didn't try to do in this work, some of which we are investigating now.

Chaitin et al. report good results from the optimization of subsumption. If two variables are assigned to the same register, then any assignment of one into the other can be removed altogether. This is especially common when the assignment is really the passing of a value parameter. Recognizing these cases requires dataflow analysis, but it is a good thing to do if you are doing the analysis anyway, for instance if you are including the local conflict coloring.

A better solution to the problem of recursive or indirect calls is needed if we want to do functional or object-oriented programming. Recursive calls can probably be handled better by breaking the call graph into strongly connected components and treating each strongly connected component as a single node in the algorithm of section 2.2. This would allow the archiving to be done incrementally rather than all at once at the end of the cycle, which might be more robust. Indirect calls are a thornier problem. We may be able to record them in the call graph as if they were separate calls to every procedure that is assigned to a procedure variable, but we may need to do some dataflow and aliasing analysis to limit the number of different procedures that an indirect call might be calling.

We explicitly decided not to do more than the most primitive aliasing analysis in this system. The problem of global register allocation seems mostly independent of the problem of alias detection, and we were interested in how well we could do without it. It is clear, though, that the information provided by such an analysis would be helpful some of the time.

Finally, we are interested in designing a hybrid scheme, midway between the traditional compile-time approach and our link-time approach. In the former, all local variables are kept in registers, but these registers must be saved and restored on entry and exit, to prevent interference with other procedures. In the latter, there are no saves and restores because registers are allocated based on the call graph, but not all local variables get assigned to registers. Whenever a local variable is used more than twice, it is worth incurring the expense of a save and restore if it allows us to keep the variable in a register, and if there is a register available that we are not using for something more important during the lifetime of the procedure to which the variable is local. An allocation scheme that allowed us to do saves and restores only when warranted would allow us to keep more variables in registers. However, we are already keeping the most important variables in registers, so it would be interesting to see whether it would be substantially better than our simpler scheme.

6. History and acknowledgements

Many people contributed to the ideas described in this paper. Forest Baskett and Michael L. Powell wanted to try to do link-time allocation before I ever joined them. Richard Beigel built a prototype allocator to show the feasibility of the approach, and invented the callgraph traversal algorithm in section 2.2. Michael L. Powell, Loretta Guarino Reid, and Gene McDaniel were constant sources of help, ideas, and encouragement throughout the development of the system. To all of them, my thanks.

References

[1] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design,* pages 449-451. Addison-Wesley, 1979.

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

[3] G. J. Chaitin. Register allocation & spilling via graph coloring. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction,* pages 98-105. Published as *SIGPLAN Notices 17* (6), June 1982.

[4] Frederick C. Chow. *A Portable Machine-Independent Global Optimizer — Design and Measurements.* (PhD dissertation) Computer Systems Laboratory Technical Note No. 83-254. Stanford University, December 1983.

[5] Jack J. Dongarra. Performance of various computers using standard linear equations software in a Fortran environment. *Computer Architecture News 11* (5): 22-27, December 1983.

[6] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction,* pages 120-126. Published as *SIGPLAN Notices 17* (6), June 1982.

[7] John Hennessy. Stanford benchmark suite. Personal communication.

[8] John L. Hennessy, Norman P. Jouppi, Steven Przybylski, Christopher Rowen, and Thomas Gross. Design of a high performance VLSI processor. In Randal Bryant, editor, *Third Caltech Conference on Very Large Scale Integration,* pages 33-54. Computer Science Press, 11 Taft Court, Rockville, Maryland.

[9] David A. Patterson. Reduced instruction set computers. *Communications of the ACM 28* (1): 8-21, January 1985.

[10] George Radin. The 801 minicomputer. *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, pages 39-47 (March 1982).* Published as *SIGARCH Computer Architecture News 10* (2), March 1982, and as *SIGPLAN Notices 17* (4), April 1982.

[11] Christopher J. Terman. *User's Guide to NET, PRESIM, and RNL/NL.* M.I.T. Laboratory for Computer Science, 545 Technology Square, Room 418, Cambridge, Massachusetts.