Minimizing Register Usage Penalty at Procedure Calls

Fred C. Chow

MIPS Computer Systems, Inc.
930 Arques Ave
Sunnyvale, CA 94086

Abstract

Inter-procedural register allocation can minimize the register usage penalty at procedure calls by reducing the saving and restoring of registers at procedure boundaries. A one-pass inter-procedural register allocation scheme based on processing the procedures in a depth-first traversal of the call graph is presented. This scheme can be overlayed on top of intra-procedural register allocation via a simple extension to the priority-based coloring algorithm. Using two different usage conventions for the registers, the scheme can distribute register saves/restores throughout the call graph even in the presence of recursion, indirect calls or separate compilation. A natural and efficient way to pass parameters emerges from this scheme. A separate technique uses data flow analysis to optimize the placement of the save/restore code for registers within individual procedures. The techniques described have been implemented in a production compiler suite. Measurements of the effects of these techniques on a set of practical programs are presented and the results analysed.

1. Introduction

Recent trends in computer architecture favor large register sets [1]. Making good use of a large number of registers can speed up program execution by reducing memory accesses. Coincidental to the use of a large number of registers is additional overhead from the need to save and restore registers at procedure calls. When a procedure is called, it is necessary to save the value of all the registers it will use and restore their values on return from the call. This overhead is especially pronounced in programs that are call-intensive. Thus, using a large number of registers has the undesirable effect of making procedure calls expensive by increasing the memory traffic during the calls.

One solution is to provide a special run-time windowing mechanism on the register file in the underlying architecture. This is very effective in reducing memory traffic [2]. However, its implementation consumes hardware resources, imposes an extra burden on the hardware designer and may impact the cycle time of the processor [3, 4]. Floating-point references are not helped at all when floating-point computations are performed in a separate co-processor with its own floating-point registers. Though this hardware feature benefits programs that are call-intensive, irregular call patterns can result in sub-optimal performance [5].

Another solution is to provide dynamic tracking of register usage during execution, as described by Steele [6] and Lang [7]. These schemes use dynamic masks that need to be updated during procedure calls and at register accesses. Although shown to be effective at reducing register saves and restores, they introduce additional complexity at run-time, and also require special hardware support for efficient implementation. The additional hardware may impact the pipeline design and thus the cycle time of the processor.

This paper presents techniques to reduce register save/restore traffic purely in software. This same problem can be regarded as inter-procedural register allocation, but the latter also implies the ability to allow the same global variable to be accessed in the same register across procedure boundaries. Wall [8] presents an inter-procedural register allocator that allocates registers over the entire program at link-time. It assigns local variables that cannot be active concurrently to the same register, according to the call graph. It has the the advantage of being able to estimate the global usage counts of local and global variables and assign only the most frequently-used to registers. Attempts to save register contents so that they can be re-used are not made except in the case of recursive procedures and indirect calls. Wall reported excellent results even when each register is allowed to be assigned to only one variable within each procedure, using from 32 to 52 registers. The programs in his benchmark suite require from 165 to 693 pseudo-registers (termed "groups"), and with his machine actually providing 52 registers, his allocator still left many variables unassigned to registers.

Steenkiste [9] presents a simple and efficient interprocedural allocation scheme that assigns registers to local variables in procedures in a depth-first traversal of the program call graph. A bottom-up approach brings about more opportunity to share registers among procedures than a top-down approach. For example, any register used in the main program cannot be re-used by any other procedure without saving its contents and restoring it afterwards. In contrast, all leaf procedures can

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1988 ACM 0-89791-269-1/88/0006/0085 $1.50

Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation
Atlanta, Georgia, June 22-24, 1988

use the same register since they are never active at the same time.† At each call, the registers used by the callee are known since the callees have already been processed. By avoiding re-use of these registers in the current procedure, the need to save and restore the registers around the call is eliminated. When the allocator runs out of registers at the upper regions of the call graph, Steenkist's allocator switches back to ordinary per-procedure allocation, with their associated register saves/restores at procedure boundaries. Local variables of recursive procedures are also saved at recursive calls.

In this paper, we present a method for inter-procedural register allocation based on the bottom-up approach. We made no attempt to allocate global variables to the same registers throughout the entire program, because this would have made it impossible to allocate in one pass. But we do allocate them to registers within procedures in which they appear. For the same reason, we do not rely on usage counts over the entire program, but we maximize the utility of all the registers over the program's call tree. Our method is aimed at reducing the register save/restore traffic at procedure calls and at streamlining the parameter-passing operation. The techniques are applied at compile-time and are built on top of coloring-based intra-procedural register allocation already performed by the compiler.

2. One-pass Algorithm

One advantage of the depth-first ordered inter-procedural register allocation method is that it allows the entire program to be processed in one pass. At each instant, only one procedure is being looked at. As a result, it is possible to perform intra-procedural register allocation on each procedure, but extend the local algorithm to take register usages in called procedures into account at call sites. We use the priority-based coloring allocation technique [10-12] in our intra-procedural register allocation, and have been able to extend the algorithm to suit our purpose.

Before we apply our inter-procedural algorithm, it is necessary to define the usage convention of our registers. Traditionally, registers are classified according to software convention at procedure calls. The content of a caller-saved register is regarded as being un-preserved across a call. The caller is responsible for saving the contents of caller-saved registers before a call and restoring their contents after the call returns. The content of a callee-saved register must be preserved across a call. A procedure that uses a callee-saved register must save its original content and restore it before exit. Under normal situations, our registers are divided into a caller-saved set and a callee-saved set, with four additional registers designated for parameter-passing. When not being used for passing parameters, the parameter registers are treated as caller-saved.

Caller-saved and callee-saved registers have their respective merits under different circumstances. Let's first assume the absence of any inter-procedural register usage information, which corresponds to the ordinary intra-procedural register allocation case. For a variable whose range of appearance spans many procedure calls, the use of a callee-saved register is advantageous, because saving and restoring once at the procedure entry and exit respectively are cheaper than saving and restoring around each call. If the variable's live range does not span any procedure call, which applies to all variables when the current procedure is a leaf in the call graph, then a caller-saved register is advantageous because its usage incurs no saving and restoring. Our register allocator computes different priorities with respect to the register classes and assigns each program variable to the best register class. Thus, our register allocator strives to reduce register saves and restores even when not making use of inter-procedural information.

When inter-procedural register usage information is taken into account, the register allocator tries to avoid re-use of the registers that it knows are being used inside other procedures. At points of call, the register usage information of the callee applies only to caller-saved registers, because if a callee-saved register is used by the callee, the save and restore of it has already been generated at the entry and exit respectively of the callee and cannot be optimized away. Based on the register usage information in the callee, the allocator can select a caller-saved register across the call without incurring the cost of save/restore both at the point of call and at the entry and exit of the current procedure. Thus, to derive maximum benefits when the procedures are allocated in depth-first order according to the call graph, it is advantageous to use all the registers in the caller-saved mode.‡

Thus, when inter-procedural register allocation is invoked, our set of callee-saved registers are made to operate in the caller-saved mode. The register usage information of each procedure consists simply of a flag for each register marking it as used or unused. This register usage information includes the whole call tree rooted at that procedure, and at the end of processing each procedure, it is necessary to merge the register usage in the current procedure with those of all its callees. Thus, at each call, only one set of information needs to be looked at.

When we perform register allocation under the inter-procedural mode, we compute priorities not just for

† Wall's basic algorithm also works in a depth-first ordered traversal of the procedures.

‡ By similar reasoning, if the procedures were processed top-down in the call graph, it would have been advantageous to use all registers in the callee-saved mode.

each variable, but for each variable-register combination. This is because the cost incurred by the use of each register is now different depending on their usage by callees at points of calls that lie within the live range of the variable. Apart from computing and selecting from more priorities, the basic priority-based coloring algorithm is unchanged. This does not add noticeably to the running time of the coloring algorithm, since most of the time is already spent in creation and manipulation of the live ranges and interference graph. Our implementation of the register allocator can actually switch between intra- and inter-procedural allocation according to a compilation flag.

Because we perform coloring within each procedure, a register used by a child procedure can actually be re-used without save/restore if the range of usage does not span the call to the child. For example, in Fig. 1, even though functions *p* and *q* can be active at the same time, the same register can be assigned to variables *a*, *b* and *c* without save/restore. With equal priorities, the allocator will prefer a register that has already been used in the current call tree. The effect is to minimize the number of registers used in each call tree.

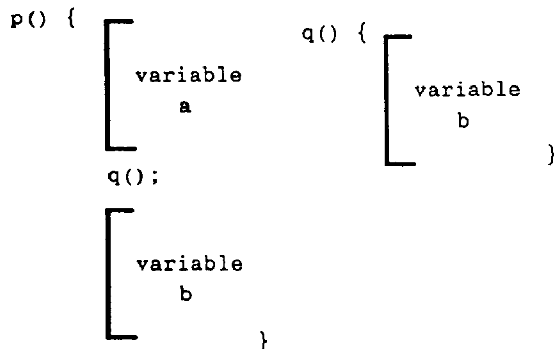

Fig. 1. Re-use of register in simultaneouly active procedures

At the upper regions of the call graph, the register allocator will be forced to use registers that require saving/restoring across calls, since the limited number of registers will eventually be exhausted. The resulting performance of the inter-procedural register allocator for the procedures high up in the call graph will then be little different from that obtained from intra-procedural allocation. However, there are ways to alleviate this problem, and we'll discuss them in the upcoming sections.

3. Incomplete Procedure Information

Our inter-procedural register allocation relies on knowledge about the register usage patterns in the called procedures at call sites. However, incomplete register usage information for called procedures can arise under three different situations. Under separate compilation, the bodies of the called procedures are not visible to the allocator. With indirect calls, the compiler cannot identify the possible call candidates without lengthy program flow analysis, and even if that is done, the allocator has to take the union of the registers used by all the candidates. Lastly, recursion creates cycles in the call graph so that, no matter what ordering is used, it is impossible to process a procedure in a recursive call chain after all its callees have been processed.

The above three situations can be generalized as follows: we cannot use the inter-procedural register allocation scheme for a procedure if any of its callers has been processed* or is unknown. The former occurs when the call graph contains cycles or the procedures are not processed in depth-first ordering. The latter occurs if the procedure is external to the current module or its text address is ever assigned to a procedure pointer so that it can be called indirectly. We call a procedure to which either of these conditions applies an *open* procedure. Procedures that are not open are called *closed* procedures. An open procedure cannot propagate its register usage information to all its callers, since when its callers are processed, such information is either not yet determined or is unobtainable. For open procedures, we switch back to the default intra-procedural allocation linkage rules where the callee-saved registers reclaim their original identities as callee-saved registers.

When we process an open procedure, some of its callees may be closed procedures. In such a case, the register usage information in the children is still useful to the parent. The difference is that when a callee-saved register is used by the parent or any of its children, the parent must save it on entry and restore it on exit. Our inter-procedural allocation scheme can alternatively be looked upon as providing a mechanism to propagate the saves/restores of callee-saved registers to the upper regions of the call graph. In the ideal case, when all procedures are closed except the main program at the root of the call graph,‡ and there are enough registers provided by the machine, the saving and restoring of the callee-saved registers can be propagated all the way to the entry and exit respectively of the main program. There, they are saved and restored only once in the entire duration of the program's execution. This propagation of the saves/restores of callee-saved registers is interrupted by the occurrences of open procedures in the call graph.

When a register is saved by an open procedure, that register can be marked un-used in the register usage information the procedure provides to its callers, because its content is undisturbed by the procedure. For open procedures, the register allocator can assume

* This includes the case of a self-recursive procedure, when one of the caller, itself, is being processed.

‡ The main program is always open since it is called externally by the operating system when the program is invoked.

at once that all callee-saved registers are unused but all caller-saved registers are used, which is the default linkage protocol. As a result, open procedures do not have to specify their register usage information.

Thus, the performance of our inter-procedural allocator is not compromised by incomplete procedure information. In any given call graph, inter-procedural and intra-procedural register allocation can co-exist. Though saving and restoring a register at entry and exit poses additional run-time costs, this allows the parents' free usage of the register. At the upper regions of the call graph, our inter-procedural register allocation can still perform well compared to intra-procedural register allocation, when some registers have been saved in the lower part of the graph to allow for their free usage inter-procedurally in the upper part of the graph. This effect can be further enhanced by optimizing the positioning of the save/restore code, which we'll discuss in Section 5.

4. Parameter Passing

A natural way to optimize parameter passing arises out of our inter-procedural register allocation. Incoming parameters to a procedure are local variables whose values are pre-set at the procedure's entry point. For a closed procedure, the parameter variable is allowed to be assigned to an arbitrary register, and the register usage information of the procedure additionally describes which parameter is being passed in which register. At the points of call, the callers will pass the outgoing parameters in the designated register. Under inter-procedural register allocation, any register can serve as a parameter register. For open procedures, the default linkage convention is re-instituted where the first four parameters are passed in the four parameter registers.

At points of call, the coloring algorithm assigns higher priority to a variable/register pair if the variable is an outgoing parameter that has to be passed in that register. This pre-assignment of the parameter register to the outgoing parameter variable applies not only to the point of call, but also to the entire live range of the variable. Thus, from caller to callee, the parameter can be left undisturbed in the parameter register.

5. Shrink-wrapping Callee-saved Registers

A typical procedure consists of many different execution paths, not all of which may be exercised for each invocation. The ordinary convention for callee-saved registers requires that the saving and restoring are done at the entry and exit of the procedure respectively. This introduces some redundancy, because the saved registers may not be used for the execution path of that invocation. This problem does not occur for caller-saved registers, because the caller saves them only when they are used in both the called procedures and the region spanning the call. As we saw in Section 3, one way of removing this redundancy is by propagating the saves and restores of callee-saved registers to the upper regions of the call graph. This propagation is hindered by open procedures, in which case the saves and restores have to be placed in the bodies of the open procedures.

It is possible to optimize the placement of the saves/restores for callee-saved registers so that they occur only over regions where the registers are used, rather than at the entry and exit of the procedure. The effect is to *shrink wrap* the saves/restores around their regions of activity. This serves to suppress the redundant execution of the code when the flow of control does not cover the regions where the registers are used. This optimization is based on flow analysis of the save and restore code of each callee-saved register with respect to the control flow graph of the procedure [13]. Since the number of callee-saved registers is fixed and usually not large, the data flow information for the registers can be compactly encoded in bit vector form using a word of storage, so that the data flow analysis can be efficiently performed.

The first step is to initialize a local attribute that tells whether each register is used in each basic block of the procedure. We call this attribute APP. The next step involves determining the anticipability and availability of the uses, which we call ANT and AV respectively. The use of a register is *anticipated* at a given point if a use of the register will be encountered in all possible execution paths leading from that point. The use of a register is *available* at a given point if a use of the register has been encountered in all possible execution paths that lead to that point. We use the names ANTIN and AVIN when referring to these attributes at the entries of basic blocks, and the names ANTOUT and AVOUT when referring to these attributes at the exits of basic blocks. These data flow attributes can be determined by solving the following boolean equations by iterative data flow analysis. The subscript i identifies the attribute as being for the ith basic block.

$$\mathrm{ANTOUT}_i = \begin{cases} \text{false} & \text{if } i \text{ is an exit} \\ \prod_{j \in succ(i)} \mathrm{ANTIN}_j & \text{otherwise} \end{cases} \tag{3.1}$$

$$\mathrm{ANTIN}_i = \mathrm{APP}_i + \mathrm{ANTOUT}_i \tag{3.2}$$

$$\mathrm{AVIN}_i = \begin{cases} \text{false} & \text{if } i \text{ is an exit} \\ \prod_{j \in pred(i)} \mathrm{AVOUT}_j & \text{otherwise} \end{cases} \tag{3.3}$$

$$\mathrm{AVOUT}_i = \mathrm{APP}_i + \mathrm{AVIN}_i \tag{3.4}$$

We want to insert a register save only at a basic block where the use is anticipated. If inserted at any other place, partial redundancy is introduced [14]. Similarly,

we want to insert the restore only at a basic block where the use is available to avoid partial redundancy in the restore. The next task is to determine the best places to insert the save and the best places to insert the restore. These two problems are symmetrical with respect to the flow graph, so that one can be solved using the same technique as the other.

We first consider the solution for the best place to insert the save. We adhere to the rule of always inserting register save code at basic block entries, which will not limit the effects of our optimization in any way. The insertions should be at the earliest points in the program leading to one or more contiguous regions where the register is used. This implies that, at the points of insertion, the ANTIN attributes at the preceding basic blocks (the immediate predecessor nodes in the control flow graph) must be false for the register concerned. In addition, there must not be any earlier save inserted, because if saved twice, the second save will destroy the original content. This means the AV attribute must also be false.† Using the attribute SAVE to describe the insertion of register save code at each basic block, we can compute SAVE as follows:

$$SAVE_i = ANTIN_i \times (\neg AVIN_i) \times \prod_{j \in pred(i)} (\neg ANTIN_j) \qquad (3.5)$$

The above computation of SAVE can produce incorrect results in some situations. In Fig. 2(a), variable x appears in both basic blocks 3 and 5, so that a register is used to contain it. The above solution will cause register saves to be inserted at the entries of both basic blocks 3 and 5, but this leads to the register being saved twice when the execution path goes through nodes 3 and 5. The only possible way to correct the insertion is to create a new node in the control flow graph (node 6 in Fig. 2(b)) and insert the save there instead of node 5. However, the creation of such new basic blocks requires the introduction of extra branches into the program code, which lengthens the execution time. This offsets the advantage gained by removing the partial redundancy in the save code. Thus, under such situations, we choose not to introduce new nodes in the control flow graph. Instead, we extend the range of usage of the register by propagating the APP attribute (and thus the ANTIN attribute also) to the basic blocks that cause the above incorrect insertion. The same applies to the corresponding situation when computing positions to insert restores. Each time we get a new set of values for APP, the global attributes ANTIN, ANTOUT, AVIN, and AVOUT are re-computed and we again check for range extension. This process is repeated until the range can no longer be extended. The range extension applies to each register, but by operating on the bit vectors we can process all the registers at once. The iteration terminates when the APP vector is not changed at any node in the flow graph. In practice, this extension of the usage ranges of the registers requires from one to two iterations depending on the program, and is thus inexpensive.

† If AV is true, the register has been saved but not yet restored; if false, the register has either not been saved, or has been both saved and restored. This is according to the definition of RESTORE. Thus, the definitions of SAVE and RESTORE are mutually dependent.

By virtue of the symmetry between register save and restore with respect to the flow graph, the attribute to describe the insertion of register restore code, RESTORE, is computed as given by Eq. (3.6). We always insert register restores at the exits of basic blocks.

$$RESTORE_i = AVOUT_i \times (\neg ANTOUT_i) \times \prod_{j \in succ(i)} (\neg AVOUT_j) \qquad (3.6)$$

There is one more consideration in performing the above shrink-wrap optimization. If the shrink-wrapped region lies completely inside a loop, there will be serious performance impact: instead of saving and restoring once per invocation of the procedure, this is now repeated for each iteration of the loop. To prevent this from occurring, whenever a register is used inside a loop, we propagate its APP attribute throughout the entire region of the loop. Thus, any shrink-wrap is not allowed to penetrate loop boundaries.

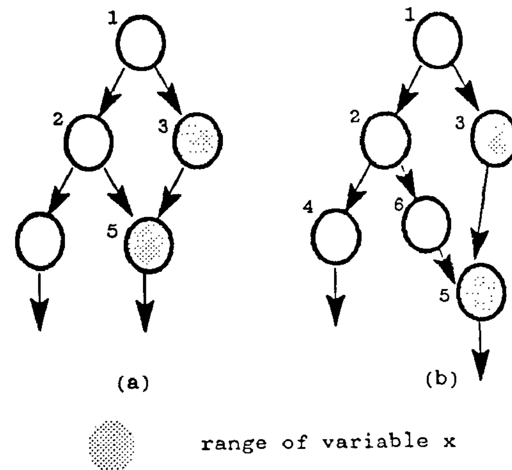

Fig. 2. Dependence on form of control flow

There are other situations where the shrink-wrap optimization produces results whose effects are uncertain without dynamic execution profiles. Fig. 3 depicts a typical situation. Assuming equal probability for each branch target, the four possible execution paths will each occur 25% of the time. The optimization produces positive impact on run-time in one case, negative impact in another case and no net effect in the remaining two cases.

6. Combining the Techniques

As we saw in Section 3, our inter-procedural register allocation algorithm propagates the saves/restores of callee-saved registers up the call graph. On the other hand, saving and restoring a register in the current procedure have the effect of shifting the cost of the register usage from the ancestors to the current procedure. If the allocator does not run out of registers in the upper portion of the call graph, then propagating the saves/restores up the call graph is beneficial. If it does run out of registers, however, either strategy can yield better result depending on the actual situations. In the example of Fig. 4, both procedures *p* and *r* use register 1. Register 1 may be saved and restored around the call to *q* in procedure *p*, or it may be saved at the entry of *r* and restored at the exit of *r*. If the call to *q* is executed less frequently than the call to *r*, then the former is advantageous. But if the call to *r* is less frequent than the call to *q*, the latter is preferred.

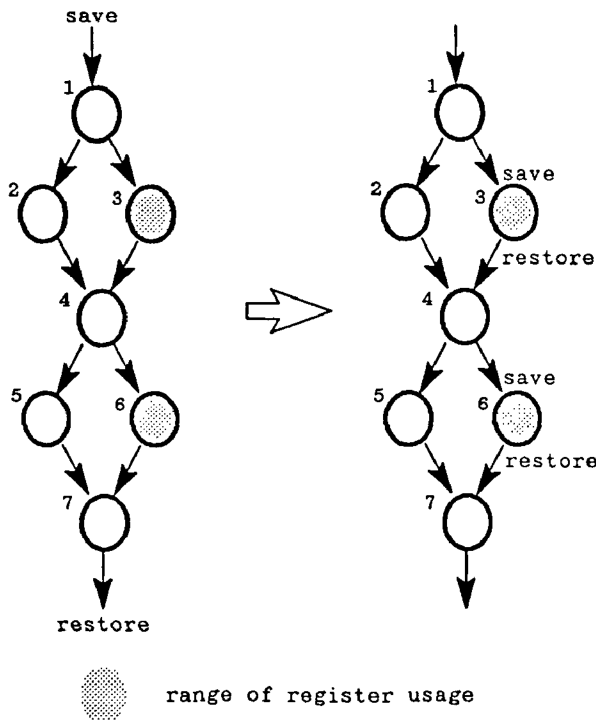

Fig. 3. Effects of shrink-wrap optimization

Since we process the program in one pass, we cannot anticipate whether we'll run out of registers later when we process the upper part of the call graph. However, a good strategy follows naturally from the preceding shrink-wrap optimization for callee-saved registers. When we process a closed procedure in inter-procedural allocation mode, we choose to propagate the save and restore of a callee-saved register up to the parent procedures only when the save has to be inserted at the procedure entry, which is when the usage range of the register spans the entire procedure.‡ If the range of usage does not include the entire procedure, then the saving and restoring of that register in the ancestors will be redundant when the actual execution does not take the path that passes through that range.

Adopting the above strategy in our inter-procedural register allocation can enhance its performance, especially in the upper part of the call graph. Because more callee-saved registers are saved in the lower regions of the call graph, more free registers are made available at the upper regions. Our whole inter-procedural register allocation strategy is built upon the interaction between the different techniques we have described.

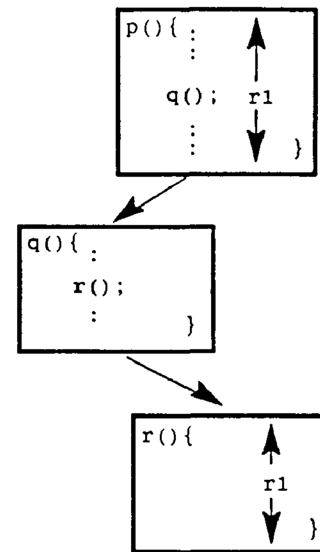

Fig. 4. Inserting saves and restores in call graph

7. Compilation Setting

A previous paper [15] has described a practical compilation environment that provides for the efficient and effective invocation of our inter-procedural register allocation for ordinary programs. The MIPS Compiler System uses the common back-end strategy to support compilation and optimizaton for a suite of programming languages [16]. The intermediate language is Ucode, and the register allocator is the last phase of the global optimizer on Ucode called Uopt.

‡ For our save/restore placement optimization algorithm to be correct, inserting the save at the entry must imply that a corresponding restore is inserted at all exits of the procedure.

				I. % reduction in cycles			II. % reduction in scalar loads/stores		
program	language	source lines	cycles/call	A	B	C	A	B	C
nim	Pascal	170	43	2.1%	12.0%	14.1%	7.0%	42.3%	49.6%
map	Pascal	410	71	-0.1%	3.9%	3.9%	0%	42.5%	42.5%
calcc	Pascal	500	31	0%	9.5%	9.5%	0%	57.7%	57.6%
diff	C	670	150	0%	0.9%	0.8%	0.1%	20.8%	19.7%
dhrystone	C	770	36	0%	4.1%	4.1%	0%	41.7%	41.7%
stanford	Pascal	940	70	0.8%	0.2%	1.3%	12.5%	-1.0%	20.8%
pf	Pascal	2400	111	0%	2.5%	2.3%	0.2%	50.3%	49.1%
awk	C	2500	91	-0.1%	2.2%	0.9%	0%	14.6%	4.5%
tex	Pascal	5700	45	0.2%	3.3%	3.7%	1.1%	11.8%	13.5%
ccom	C	12100	56	0%	-2.6%	-1.4%	0.6%	-26.1%	-15.9%
as1	Pascal/C	14100	51	-0.2%	2.7%	1.9%	0.1%	12.4%	10.8%
upas	Pascal	16600	46	0.1%	1.7%	1.3%	1.2%	9.3%	6.8%
uopt	Pascal	22300	49	0%	0.5%	1.0%	1.6%	-1.8%	8.1%

All comparisons are with respect to -O2 compilation with shrink-wrap disabled.

Key:
A – compiled -O2 with shrink-wrap enabled
B – compiled -O3 with shrink-wrap disabled
C – compiled -O3 with shrink-wrap enabled

Table 1. Effects of applying techniques on 13 different programs†

As discussed earlier, the inter-procedural allocation does not require complete program information to operate. But for more optimal results, our compiler system allows the Ucode from separate program units and from libraries to be linked together. A different phase re-arranges the linked procedures into the depth-first ordering before presenting them to Uopt. The approach provides maximum utilization of the allocator's capability with little impact to the user interface.

8. Measurements

We now study the effect of applying our techniques to programs compiled and run on the MIPS R2000 processor*. The R2000 is a RISC processor with a load/store architecture in which most instructions execute in a single clock cycle [17]. There are 20 general purpose registers available for use by the register allocator, of which 11 are caller-saved and 9 are callee-saved. The floating-point coprocessor has 4 caller-saved and 6 callee-saved floating-point registers. In addition, the function return registers and linkage registers are also used but they cannot be allocated inter-procedurally. All these registers are allocated within each procedure using priority-based coloring.

Global optimization and register allocation are ordinarily invoked with the -O2 compilation flag. If invoked under the -O3 flag, register allocation is performed using inter-procedural information according to the techniques we have described. However, the shrink-wrap optimization is independent of inter-procedural information, and thus is performed under both -O2 and -O3. We'll use -O2 with shrink-wrap disabled as the base-line for all our performance comparisons.

Executable programs typically comprise libraries provided by the programming languages that are linked with the user code. Thus, the lower regions of program call graphs can be library code. The increased size of the linked Ucode increases compilation and optimization time in the back-ends. Since our scheme is not seriously affected by incomplete call graphs, the current setup of -O3 compilation excludes linking with any external or library objects.

Our run-time data are generated using the MIPS instruction tracing facility *pixie*. This tool gives a full picture of the machine cycles and intruction mix needed to execute the programs, allowing us to measure program performance independent of other hardware parameters like cache and processor clock frequency. Any cycle savings realized are exclusive of cache misses and memory management unit overhead, and thus

† All measurements were generated using Release 1.30 of the MIPS Compiler Suite.

* R2000 is a registered trademark of MIPS Computer Systems, Inc.

usually understate the improvements in real time.

In addition to executed cycles, we provide data on the total scalar loads and stores executed. These are loads and stores attributed to scalar variables, common subexpressions and register saves and restores, which are removable by the register allocator given an unlimited number of registers. By relating the change in dynamic scalar loads/stores to the change in executed cycles, we can get a rough estimate of the performance improvement that can be brought about by perfect register allocation in individual programs. Due to the different natures of programs, the same percentage of reduction in scalar loads/stores can lead to widely different speed-ups in different programs.

When looking at data on inter-procedural optimization effects, it is important to take the call frequencies of the programs into account, since opportunities for improvement arise only at procedure calls. For the same reason, the benchmark programs we use are mostly call-intensive. Given that the number of registers provided by target machines are always fixed and limited, program size can seriously affect the performance of inter-procedural register allocation. As a result, the sizes of our benchmark programs range from small to very large. Details about our benchmarks are given in the Appendix. In Table 1, the benchmarks are arranged in the order of increasing source line counts, which do not include those from linked-in libraries. The code belonging to libraries were compiled without inter-procedural allocation.

Since changes in the number of memory references can unpredictably affect the subsequent pipeline reorganization phase, a small reduction in the number of loads and stores may not positively affect the speed of the program. Such noise in the compilation process results in inconsistencies in the data for cycles and loads and stores in Table 1, as exemplified by the B columns for uopt.

The performance numbers shown in Table 1 are all normalized with respect to the performances under intra-procedural global register allocation. Columns IA and IIA give the isolated effect of shrink-wrap optimization before applying inter-procedural allocation. Column IIA shows that this optimization always reduces memory accesses. However, the effect on the execution time is barely noticeable, with the exception of nim.

Columns IB and IIB give the improvements brought about by our inter-procedural register allocation scheme but without shrink-wrap optimization. Since our base-line comparison is that of intra-procedural global register allocation, the high base does not make our data look good, because intra-procedural allocation has already removed a majority of the loads and stores attributed to scalar variables and common subexpressions. In general, the improvement is more pronounced for the smaller benchmarks. In the case of ccom, one of the larger programs, it actually runs slower due to inter-procedural allocation.

Columns IC and IIC show the effects of adding shrink-wrap optimization to our inter-procedural allocation scheme. There are good improvements over B in nim, stanford, tex and ccom, but slight decreases in performance in the case of awk, as1 and upas. On the whole, the extents of the improvements obtainable seem to justify the inclusion of shrink-wrap optimization under inter-procedural allocation.

We account for the less than promising results we obtained as follows. First, under intra-procedural allocation, our register allocator already has the choice to use either caller-saved or callee-saved registers. This provides one extra degree of freedom to the allocator, allowing it to minimize save/restore overhead even in the absence of inter-procedural information, as described in Section 2. Thus, we started out with a competent base from which to improve upon. Secondly, for large benchmarks, 20 registers (excluding floating-point) are clearly inadequate to cover the large call graphs, so that the improvement in execution time is smaller. The relevant parameter is the height of the call graph rather than the width. Thus, the nature of the program dictates how its size affects the performance of our allocator. Inter-procedural register allocation requires a large number of registers in order to have noticeable impact on practical programs. Finally, we lack information on the execution frequencies at different levels of the call graph. Knowledge of such profile data can enable the register allocator to distribute saves/restores more optimally across the different levels of the call graph. The negative improvement in ccom was caused by the propagation of saves/restores to the upper region of the call graph, which turns out to execute more frequently than the lower region. The feedback of profile data to the register allocator is a capability that we plan to add in the future.

Our next set of measurements, given in Table 2, compare the differences between caller-saved and callee-saved registers in our inter-procedural allocation scheme. Since different parts of the MIPS execution environment have been hard-coded according to the native convention for the two classes of registers, it is impossible to change the usage convention of the registers without drastic overhaul of the system code. However, it is possible to limit the number of registers used by the register allocator so that all the used registers belong to one of the two classes. In Table 2, columns D, the register allocator is allowed to use only 7 caller-saved registers in performing inter-procedural allocation, and in columns E, it is allowed to use only 7 callee-saved registers. There is no restriction in the floating-point register set, which should not affect our study, since our benchmarks use predominantly integer data. We use the same base-line as Table 1 for our comparisons. Because of the smaller number of registers, most of the programs run slower. But we are interested only in the relative performance between columns D and E.

According to the data in Table 2's column II, using caller-saved registers is better for four of the programs (nim, map, stanford and ccom), while using callee-saved registers is better for the rest of the programs, except dhrystone, which shows no clear preference for either class of registers. The fundamental difference between the two register classes under our scheme of inter-procedural allocation is that, under conditions of running out of registers, callee-saved registers permit the migration of saves/restores up the call graph, allowing more freedom in the allocation process. Thus, callee-saved registers are more advantageous in large programs, where register shortage is severe. In the small benchmarks nim, map and stanford, there are fewer cases of running out of registers, and since caller-saved registers can be used free of save and restore as long as there are free registers, they become more advantageous. Ccom is an exception among the larger benchmarks. The fact that it gets better performance with caller-saved registers than callee-saved registers bolsters our earlier inference that the upper region of its call graph executes more frequently than the lower region.

	I. % reduction in cycles		II. % reduction in scalar loads/stores	
program	D	E	D	E
nim	11.8%	6.9%	43.3%	28.2%
map	-7.2%	-10.5%	-120.2%	-159.6%
calcc	-7.7%	4.8%	-57.7%	24.2%
diff	-12.6%	-7.7%	-158.1%	-106.6%
dhrystone	0.7%	0.7%	10.0%	10.0%
stanford	-7.0%	-12.9%	-51.9%	-128.9%
pf	-0.5%	-0.6%	-0.5%	3.0%
awk	-2.8%	-1.5%	-26.6%	-20.1%
tex	-0.8%	3.3%	-9.7%	11.0%
ccom	-2.4%	-5.1%	-17.9%	-37.7%
as1	-2.2%	-2.4%	-17.2%	-12.8%
upas	-5.3%	0.6%	-26.7%	1.8%
uopt	-3.9%	-3.3%	-43.1%	-31.3%

All comparisons are with respect to -O2 compilation using full register set with shrink-wrap disabled.

Key: D – same as in Table 1's C but using only 7 caller-saved registers

E – same as in Table 1's C but using only 7 callee-saved registers

Table 2. Effects of the 2 different register classes

9. Conclusion

A simple and elegant extension to the priority-based coloring algorithm has made it possible to obtain the benefits of inter-procedural register allocation. The approach depends on processing the procedures in depth-first ordering according to the program call graph, and uses two different save/restore conventions for the registers. The techniques do not rely on any special hardware feature, and can be applied to any processor with a fixed register set. Our results show that pure software can effectively reduce register usage penalty at procedure calls. With the techniques, programs stand more ready to make good use of a larger number of registers.

Acknowledgement

The author has benefited greatly from ideas from John Hennessy, Earl Killian and Larry Weber. The implementation of these ideas in MIPS' compilers have been made possible by the related work of Kevin Enderby and Mark Himelstein. Other members of the MIPS compiler team, especially Steve Correll, Steve Hanson, Bettina Le Veille, have provided support throughout the development and testing phases.

Appendix

Benchmark	Description
nim	a program to play the game of Nim†
map	a program to find a 4-coloring for a map†
calcc	a program that manipulates dynamic and variable-length strings†
diff	the UNIX‡ file comparison utility
dhrystone	a synthetic benchmark by Reinhold Weicker
stanford	a benchmark suite collected by John Hennessy
pf	a Pascal pretty-printer written by Larry Weber
awk	the Awk pattern processing and scanning utility from UNIX
tex	virtex from the TeX typesetting package
ccom	first pass of the MIPS C compiler
as1	the MIPS assembler/reorganizer
upas	first pass of the MIPS Pascal compiler
uopt	the MIPS Ucode global optimizer, including the register allocator

References

1. William Stallings, *Reduced Instruction Set Computers,* IEEE Computer Society Press (1987).
2. David A. Patterson, "Reduced Instruction Set Computers," *Communications of the ACM*

† These programs originate from an undergraduate programming course given at Stanford.

‡ UNIX is a registered trademark of AT&T.

28(1) pp. 8-21 (January 1985).

3. R. Sherburne, "Processor Design Tradeoffs in VLSI," Technical Report UCB/CSD84/173, Ph.D. Thesis, University of California at Berkeley (April 1984).

4. John L. Hennessy, "VLSI Processor Architecture," *IEEE Trans. on Computers* **C-33**(12) pp. 1221-1246 (Dec 1984).

5. Yuval Tamir and Carlo H. Sequin, "Strategies for Managing the Register File in RISC," *IEEE Trans. on Computers* **C-32**(11) pp. 977-989 (November 1983).

6. G. L. Steele Jr. and G. J. Sussman, "The Dream of a Lifetime: A Lazy Variable Extent Mechanism," *Conference Record of the 1980 LISP Conference*, pp. 163-172 (August 1980).

7. Tomas Lang and Miquel Huguet, "Reduced Register Saving/Restoring in Single-Window Register Files," *Computer Architecture News*, pp. 17-26 (June 1986).

8. David Wall, "Global Register Allocation at Link-time," *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pp. 264-275 (June 23-27, 1986).

9. Peter Steenkiste, "LISP on a Reduced-Instruction-Set Processor: Characterization and Optimization," Technical Report 87-324, Ph.D. Thesis, Computer Systems Laboratory, Stanford University, Stanford, CA (March 1987).

10. Fred Chow, "A Portable Machine-Independent Global Optimizer - Design and Measurements," Technical Report 83-254, Ph.D. Thesis, Computer Systems Laboratory, Stanford University, Stanford, CA (Dec 1983).

11. Fred Chow and John Hennessy, "Register Allocation by Priority-based Coloring," *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pp. 222-232 (June 17-22, 1984).

12. James Larus and Paul Hilfinger, "Register Allocation in the SPUR Lisp Compiler," *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pp. 255-263 (June 23-27, 1986).

13. Alfred Aho, Ravi Sethi, and Jeffrey Ullman, *Compilers - Principles, Techniques, and Tools*, Addison-Wesley (1986).

14. E. Morel and C. Renvoise, "Global Optimization by Suppression of Partial Redundancies," *Communications of the ACM* 22(2) pp. 96-103 (February 1979).

15. Mark Himelstein, Fred Chow, and Kevin Enderby, "Cross-module Optimizations: Its Implementation and Benefits," *Proceedings of the Summer 1987 USENIX Conference*, pp. 347-356 USENIX Association, (June 8-12, 1987).

16. Fred Chow, Mark Himelstein, Earl Killian, and Larry Weber, "Engineering a RISC Compiler System," *Proceedings COMPCON*, pp. 132-137 IEEE, (March 4-6, 1986).

17. Gerry Kane, *MIPS R2000 RISC Processor Architecture*, Prentice-Hall, Inc. (1987).