Reducing Procedure Call Overhead: Optimizing Register Usage at Procedure Calls

Feipei Lai and Chia-Jung Hsieh
Dept. of Computer Science and Information Eng. & Dept. of Electrical Eng.
National Taiwan University, Taipei, Taiwan, R.O.C.
E-mail: flai@cc.ee.ntu.edu.tw

Abstract

This paper proposes a common global variable reassignment and an integrated approach which takes advantage of the complementary relationship of inlining and interprocedural register allocation to reduce the procedure call overhead without causing additional negative effect. Our approach is based on the observation of analyzed program characteristic to identify the heavy called procedures regions and the register usage information to optimize the placement of register save/restore code. This method also takes full advantage of free-use *registers at each procedure call site. The average performance improvement is 1.233 compared with the previous schemes that performed either of them independently.*

1 Introduction

On the run-time of a program, the main procedure call overhead includes common global variable access and preserving registers for local variables across procedure calls. There has been much research focuses on either of the inlining or the interprocedural register allocation independently to reduce the procedure call overhead. But either of them has some constraints. The inlining would cause the code size expansion and control stack overflow. Moreover, the interaction with the register allocation will decrease the performance[4]. On the other hand, if one procedure is called from more than two sites, some situations of call sites may prevent the others from applying interprocedural register allocation. In such condition, we pick some call site to do inlining according to the critical region information to augment the opportunity of taking full advantage of the free registers. Moreover, if the placement of the register save/restore operations of the inlined procedure is guided by the interprocedural analyzer, the problem caused by the interactions with the register allocation would be removed. So an integrated approach is proposed.

Also, we maintain common global variables in registers to avoid the extra register save/restore code at procedure call. Instead of dedicating a register to a global variable throughout the program call graph, we uses profile information to allocate an important global variable into a register only in limited subsets of the call graph.

This paper is organized as follows. Section 2 exhibits some program characteristics by a series of experiments on six C programs. The approach we provided to make more efficient use of registers is outlined in Section 3. Some experimental results are shown in Section 4. Finally, Section 5 gives the conclusion.

2 Program Characteristics

Table 1 shows the set of benchmarks that we selected as the examples in our studies. For each benchmark, most of the called C library functions are merged into the testing program with their C source code and also participate in the interprocedural optimization. The 'Size' column shows the sizes of the programs in terms of the number of lines of C code. The 'Proc.' column gives the number of procedures within the program. The 'Description' column briefly describes each program.

First, we are curious about the distribution of procedure calls on the compiler-time. The following function is applied to each benchmark program and the result is depicted in Figure 1.

$$y(f) = \frac{1}{\mathbf{T}} \sum_{j=1}^{f \times \mathbf{N}} sort\_list[j]$$

Where $\mathbf{T} = \sum_p callIn_p$ means the number of all dynamic procedure calls, and $callIn_p$ be the number of

0-8186-6555-6/94 $04.00

Program	Size	Proc.	Description
compress	1606	20	compress program
espresso	13879	376	boolean function minimization
grep	5815	82	search a file for a given string
lex	10536	134	lexical analysis program generator
li	7517	362	lisp interpreter
yacc	6881	180	parsing program generator

Table 1: Benchmarks

times the procedure p is called. The total number of procedures, N, is sorted in increasing *callIn* sequences into the Array *sort_list*. Figure 1 plots the cumulative fraction of all dynamic procedure calls against the fraction of total procedures.

According to Figure 1, we find that most of the dynamic procedure calls are dedicated to only a small proportion of the whole set of procedures. As a consequence, very significant improvement will be gained after removing the call overhead out of the call sites that the control is transferred to these heavy called procedures.

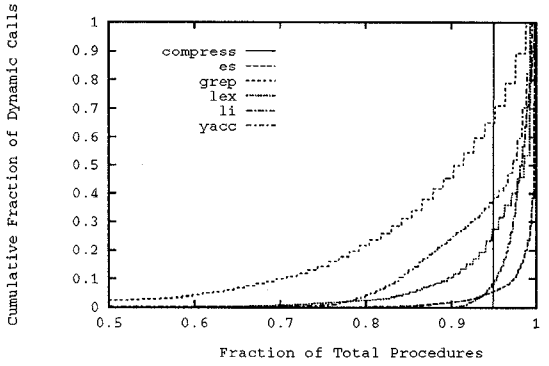


Figure 1: Distribution of procedure calls

In order to do the inlining and register upthrust optimization, the interprocedural analyzer must have the register usage information of each call site. The *free register* at call site s is a physical register that holds no live value across call site s, but it may be live elsewhere within the procedure. These free registers can be safely used by target callee procedure of the call site without any save/restore actions. In Figure 2, the number of *free registers* on each call site is depicted.

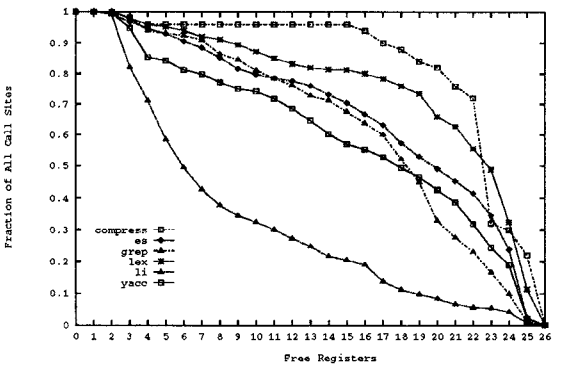


Figure 2: Number of free registers on each call site

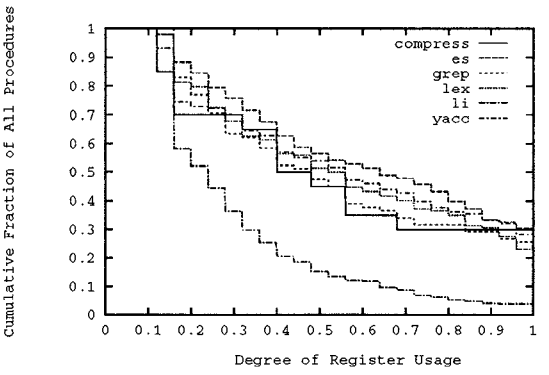


Figure 3: Degree of register usage on each procedure

Figure 3 shows the degree of register usage for each procedure under the configuration of 32 general registers. The x_axis denotes the degree of register usage measured in the fraction f of the total general registers, **R**. The following function is applied to each benchmark program.

$$y(u) = \frac{\text{number of procedures whose register usage} \geq u \times \mathbf{R}}{\text{total number of procedures}}$$

where u means the degree of register usage. Figure 3 exhibits that the register usage degree of half of the procedures is less than 0.6 (approximately, 19 registers) for each benchmark program. It means there is still opportunity for further improvement in register utilization.

All the analysis result exhibited in this section would guide the development of our interprocedural optimization to reduce most of the procedure call overhead with the least processing effort.

3 Implementation

If we restrict the processing scope of interprocedural optimization within each individual module, inlining would not be performed if the procedure bodies of caller and callee are not in the same module. So we construct a simple compilation system in which each separate module is stored in its intermediate representation form. Every time the compiler is triggered, it only re-compiles those modified source modules to the intermediate representations. Then, the modifier loads each module in its intermediate form and does the necessary modification according to the decision of the interprocedural analyzer. Therefore, our compiler can support separate compilation.

3.1 Overview of the Integrated Approach

Figure 4 illustrates the main phases in our compilation system. The GNU C compiler is used as the front end of our compiler. In the pre-compilation, we compile all modules of the source program and outputs the program structure information required to make interprocedural analysis. The program structure information includes the static program call graph, global variables of each procedure, candidates of implicitly called procedures and loop-structure. After all the information has been gathered, the frequently called procedures are identified, and then the common global variables reassignment and the interprocedural analyzer are run. The interprocedural analyzer makes the optimization operation decision based on the relationship described and generates the records of necessary modification action. Not modifying any code, instead, the relevant directions for each procedure is placed in a program database. Finally, the modifier does the actual modification according to the modification records in the program database. The data base is a pool used for keeping both the register usage information and the modification records.

3.2 Register allocator

In order to provide a more accurate estimation of register pressure on each call site and register usage information of each procedure to the interprocedural analyzer, we do the register allocation first.

Common global variables reassignment

Our compiler reserved 4 registers as dedicated registers for carrying the common promoted global variables across the procedure boundaries. In programs with

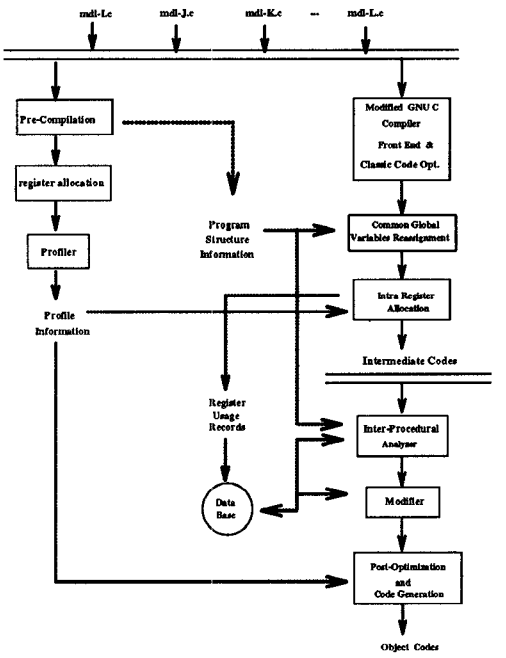


Figure 4: Compilation organization

high register usage pressure, the reservation of the registers will affect the amount of spill code. To avoid the additional spill code affects the benefit of the common global variable reassignment, the decision of promoting the global variables to the dedicated registers must take the integrated effect into account. And while not holding the common global variables, the dedicated registers are released as general registers. In order to

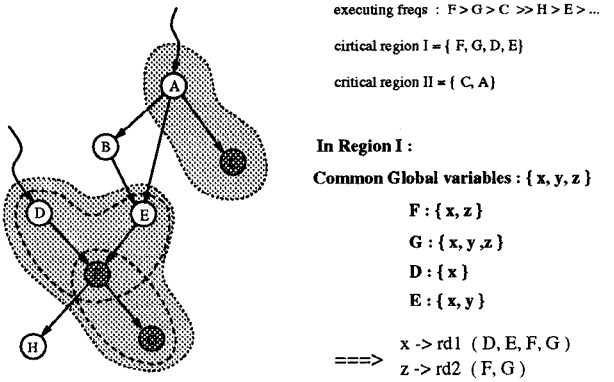


Figure 5: Critical regions.

get the most benefit with the least number of dedicated registers, one should be able to locate the regions with heavy dynamic procedure calls and assign those common frequently referenced global variables within these regions into the dedicated registers. Fortunately,

Figure 1 in previous section shows us that most part of the dynamic procedure calls are dedicated to only a small portion of the whole procedures. Our analyzer identifies the *critical regions*, the regions covering most dynamic procedure calls, on the program call graph to apply the common global variables promotion and inlining. The approach we adopted in this phase is

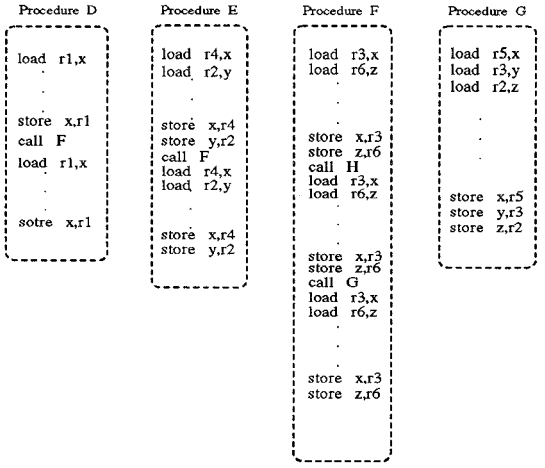


Figure 6: Intermediate code before the common global variables reassignment.

described below. Firstly, the analyzer identifies the critical regions. Each critical region is constructed by selecting the frequently called procedure, and then including *all* callers of the procedure together with itself into the same region set. Those overlapping region sets are merged as a single set. In Figure 5, F, G and C are those heavily called procedures, and then, two critical region sets are picked out as shown.

Secondly, the analyzer employs the algorithm proposed by Santhanam et al.[1] within critical regions to select the common promoted global variables. Figure 5 shows that, in a critical region, the global variables x and z are selected and assigned to the dedicated registers *rd1* and *rd2* respectively.

Finally, the analyzer modifies the intermediate code guided by the result of the common global variables selecting. Figure 6 and Figure 7 illustrate this operation.

Intra-procedure register allocation

To avoid the problem that the reuse of registers may severely hinder the later code scheduling, the register allocator takes both the live-range-conflict graph and the scheduling-conflict graph[6] into account. This method will naturally adapt to any pipeline architecture, and also be feasible to multiple-instruction-issue

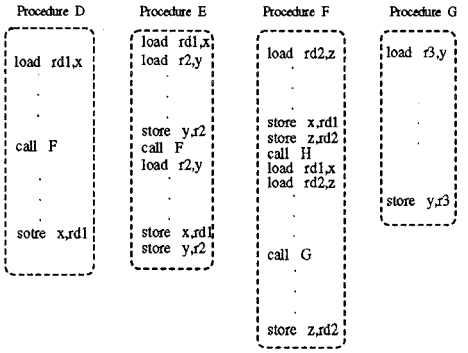


Figure 7: Intermediate code after the common global variables reassignment.

processors.

3.3 Inter-procedural optimization

Inlining

The code size expansion is a critical problem in inline expansion. For simplicity, the analyzer does the dead-code elimination firstly and then merges calls in decreasing order of the number of times they are executed until the code size increase reaches a threshold within the critical regions under those constraints depicted in section 1. Streamlining those procedure calls within these regions can remove most procedure call overheads and then result in a dramatic performance improvement but only with little compiling effort.

Reduce save/restore at procedure calls

The primary principle of this phase is to take advantage of the *free registers set* at each call site. Moreover, the registers save and restore operations can be placed at the least frequently executed sites by the *register upthrust* operation. Consider the following example. Assume procedure X is the caller of both procedures Y and Z. The spill code can be reduced if procedure Y or Z uses a free register on the call site of Procedure X. Furthermore, if sibling nodes Y and Z use the same registers that are not used in node X, and the execution frequency of node X is less than nodes Y and Z, the spill code can be moved to node X. This results in a performance gain.

The algorithm of this phase is described as follows. Firstly, do the hierarchical interval partition on the program call graph. An important property of intervals is that they have *header* nodes that dominate all the nodes in the interval; that is, every interval is a region. Based on this property, the register upthrust

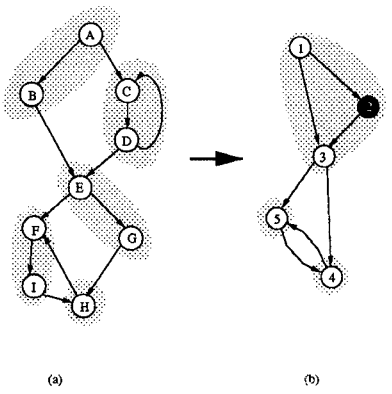


Figure 8: Example of hierarchical interval partition.

operation can be applied safely in each interval, even dealing with recursion calls. Figure 8 gives an example of interval partition. In this Figure, the nodes *C* and *D*, the recursive procedure group, are grouped into interval 2 according to the property described above. Applying register upthrust operation in interval 2 can achieve the efficiency suggested by Wall for recursive calls[3].

Secondly, for each procedure within each interval, compute the following sets:

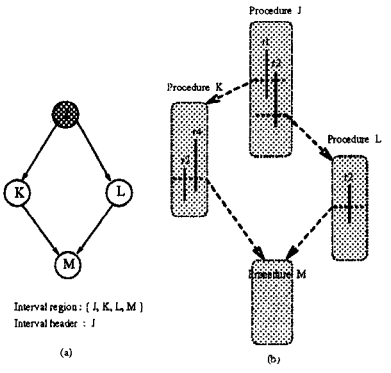
- $local\_spill_{(p)}$ = *local_use* $\cap$ {r | r *is live at some call sites that call to procedure p*}.
- $upthrust_{leaf\ nodes}$ = *local_use - local_spill.*
- $upthrust_{internal\ node}$ = (*local_use* − *local_spill*) ∪ ($upthrust_{callee}$ − *local_spill*).
- *extra_use from callee_upthrust* = ∪ $upthrust_{callee}$ − *local_use*.
- *extra_spill for callee_upthrust* = *extra_use* ∩ {x | x *is live at the call site* }.
- *final save/restore* = { *local_spill* } ∪ {*extra_spill for callee_upthrust*}.

Finally, in bottom-up ordering, propagate the upthrust register set from the leaf procedures in the interval toward the interval header. And then optimize the placement of the save/restore code, guided by the profile information.

After applying the register upthrust operation, the register save/restore code is moved to the least frequently executed sites. Figure 9 gives an example of the register upthrust.

3.4 Modifier

The modifier examines all procedures within each critical region in a bottom-up order, then, according to the result of register upthrust, the modifier inserts the



Procedure	J	K	L	M
local_use	{ r1, r2, r3 }	{ r1, r2, r3, r4 }	{ r1, r2 }	{ r1, r2, r3 }
local_spill	{ r1, r2, r3 }	{ r1, r2 }	{ r2 }	{ r2 }
upthrust	empty	{ r3, r4 }	{ r1, r3 }	{ r1, r3 }
extra-use from callee_upthrust	{ r4 }	empty	{ r3 }	empty
extra-spill for callee_upthrust	{ r4 }	empty	empty	empty
final save/restore	{ r1, r2, r3, r4	{ r1, r2 }	{ r2 }	{ r2 }

(c)

Figure 9: Example of register upthrust.

suitable store/reload code to preserve the contents of registers across the procedure calls. Finally, the modifier merges all *to_be_inlined* call sites with the body of the callee procedure. At the same time, the modifier would do the register renaming on the inlined code, expand the activation record of the caller to absorb the local variables of inlined procedure, and eliminate the unnecessary store/reload code within the inlined code. To consider the interactions between inlining and register allocation, only live registers are saved and restored across calls[4].

4 Results

The experimental results are measured on a RISC simulator which is modified from the *dlxsim*. There are 32 general purpose registers in our base architecture with 4 dedicated registers for common global variables reassignment. The comparison is built between with interprocedural optimization and without this optimization.

The performance improvement from inlining is shown at Table 2. The improvement is calculated based on the execution cycles on our simulator. The 'Removing' column denotes the percent of static/dynamic procedure calls are eliminated after the operation of inline expansion, respectively. The 'Ratio' column indicates the amount of the code increase from the inline expansion. The values are measured

Program	Removing(%)		SpeedUp	Ratio(%)
	Static	Dynamic		
compress	2.00	99.99	1.159	12.80
espresso	0.06	52.14	1.063	0.15
grep	0.38	9.49	1.002	0.07
lex	0.43	46.23	1.020	0.03
li	8.99	82.47	1.560	44.20
yacc	4.49	46.36	1.096	16.49

Table 2: Result of inlining

by the ratio of code increase to the original code size.

Table 3 shows the speed-up from the phase of register upthrust. The 'Reducing' column means the amount of register save/restore operations which are reduced after the operation of register upthrust.

Program	Reducing(%)		SpeedUp
	Static	Dynamic	
compress	68	96	1.136
espresso	41	2	1.002
grep	46	59	1.025
lex	63	63	1.051
li	16	25	1.102
yacc	46	64	1.078

Table 3: Result of register upthrust

The column 'C.G. Speedup' in Table 4 shows the performance improvement from the phase of common global variables reassignment. The next column shows that the performance improvement of average 1.233 can be achieved when the compiler applies integrating approach.

Program	C.G. Speedup	Integrating Speedup
compress	1.0569	1.352
espresso	1.0028	1.068
grep	1	1.027
lex	1.0419	1.113
li	1	1.662
yacc	1.0031	1.177

Table 4: Result of integrating approach

5 Conclusions

Based on the program analysis in section 2, we employed the common global variable reassignment and the integrated approach to optimize the register usage and then to reduce the procedure call overhead. We do the inlining under the constraints depicted and also considering the interactions with the register allocation to avoid the additional penalty. On the other hand, with the help of inlining, the interprocedural register allocation could avoid having the worst-case treatment at all call sites of the callee.

The experimental results show that this integrated approach we applied to reduce the procedure call overhead can achieve a significant speed-up which is better than previous work that performed either of them independently.

References

[1] V. Santhanam and D. Odnert, "Register Allocation Across Procedure and Module Boundaries," *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pp. 28–39, June 1990.

[2] F. C. Chow, "Minimizing Register Usage Penalty at Procedure Calls," *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 85–94, June 1988.

[3] D. W. Wall, "Global Register Allocation at Link Time," *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pp. 264–275, June 1986.

[4] Jack W. Davidsion and Anne M. Holler, "Subprogram Inlining : A Study of its Effects on Program Execution Time," *IEEE Transactions on Software Engineering*, Vol. 18, No. 2, pp. 89–102, February 1992.

[5] K. D. Cooper, M. W. Hall, and L. Torczon, "An Experiment with Inline Substitution," *Software-Practice and Experience*, Vol. 21(6), pp. 581–601, June 1991.

[6] M.-C. Chang, F. Lai, and R.-J. Shang, "Exploiting Instruction-Level Parallelism with the Conjugate Register File Scheme," *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 29–32, December 1992.