The Complementary Relationship of Interprocedural Register Allocation and Inlining

Feipei Lai and Yung-kuang Chao

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

Inline expansion and interprocedural register allocation are two general approaches used for interprocedural optimization. However, there are certain situations which prevent either of them from being applied smoothly to procedure calls. This paper describes a method of integrating interprocedural register allocation and inlining in order to reduce the procedure call overhead. The method uses profile information to identify the heavy called procedures regions and optimizes the placement of the register save/restore code. This method also takes full advantage of free-use registers at each procedure call site. The average performance improvement is 1.21 compared with the previous schemes that performed either of them independently.

1 Introduction

Subroutine calls incur a serious overhead on the run-time of a program. These overheads include:

- **Function prologue/epilogue –**
The largest part of the function prologue/epilogue is the save/restore code for the registers used in the procedure. Because most intra-procedural register allocators color the conflict graph for each individual procedure independently, the two separate procedures might use the same register for different local variables concurrently. Then, save/restore code used for preserving the different contents are needed between these two procedures.
- **Common global variable access –**
Owing to the different registers assigned to the same global variable, the compiler must insert the

save/restore code around the call site to keep the content of variable consistent in the whole program.

- **Delay slots of calls –**
On machines with a pipeline architecture, the procedure calls usually force the pipeline stage to be stalled during the period in which the called instructions being loaded from memory.
- **Barrier of instruction promotion –**
On superscalar machines, there are no instructions of the caller to be boosted beyond the procedure call sites unless the procedure call has been expanded with its code body. This is because, under the general *shadow register file* mechanism[8], the result of speculative execution was kept on the shadow registers. However, it is impossible to save the entire shadow register file across the procedure calls. If one permits the speculative instruction to be executed before the procedure call, then the instructions execution within the callee might destroy the original contents of the shadow registers.

There are two kinds of methods to reduce such procedure call overhead. One is the inlining and the other is interprocedural register allocation. By using interprocedural analysis, the compiler can avoid generating the redundant save/restore code for preserving the registers which do not hold live values across the procedure call and also make the common global variables accessible via the same register.

In section 2, we briefly describe related work. The complementary relationship of inlining and interprocedural register allocation is discussed in section 3. Section 4 exhibits some program characteristics by a series of experiments on six real programs. Section 5 presents an integrated approach which applies both

interprocedural register allocation and inlining on reducing the procedure call overhead. Some experimental results are shown in section 6. Finally, section 7 gives the conclusions.

2 Related Work

Much research focuses either on the interprocedural register allocation or on inlining, independently. This paper briefly discusses these work.

Using the inlining approach, McFarling[4] introduces the loop-base miss rate model, which characterizes the locality attribute of program execution in the loop structure, into the inlining decision. The model could predict how instruction caches behave after the inlining. The decision is made by weighing the benefit of removing calls against the increase in the instruction cache miss rate.

Chang et al.[7] describe the practical implementation issues that must be considered on the development of an automatic inliner, such as program representation, hazard prevention, and expansion sequence control. Davidson et al.[5] implement a source-to-source inliner that automatically performs inlining of C functions in a single module. Cooper et al.[6] study the effectiveness of inlining as a method for improving the program execution time of five commercial FORTRAN compilers.

Using the interprocedural register allocation approach, Wall[3] proposes the link-time register allocation based on the program call graph. In this technique, the global register allocation is postponed until link time. The compiler annotates those code which access the frequently referenced global variables with the register relocation actions. The linker assigns the dedicated registers to these global variables over the entire program according to the register relocation actions. It also assigns the local variables that are not concurrently active in the program call graph with the same register.

Chow[2] presents a one-pass bottom-up compilation scheme to minimize the register save/restore code. The procedures are compiled in the bottom-up ordering of the program call graph. At each call site, the register usage information in the called procedures is already known so that the register allocator can avoid re-use of these registers. In this approach, the allocator always favors the procedures in the lower region of the program call graph. Restricting to the one-pass approach, the global variables are not allocated with the same registers throughout the entire program. However, the allocator independently promotes

```
for ( ... , ... , ... ) {
    .....
    sub1( ..... );
    .....
    sub1( ..... );
    .....
}
```

Figure 1: Locality within the loop structure.

them to the registers within procedures in which they are accessed.

Santhanam et al.[1] describe a two-pass compilation system. For each eligible global variable, the program analyzer finds out the associative subgraph regions, *webs*, on the program call graph such that the global variable would not be referenced by any procedure outside the webs. A single callee-saved register is dedicated to a promoted global variable within a web region. The disjoint webs can be assigned with the same callee-save register. A clustering technique is proposed to allow the callee-save register spill code to be able to migrate upwards in the call graph to infrequently executed procedures.

3 The Complementary Relationship

If we do the inlining alone, there exist some constraints that prohibit the inlining operation. Firstly, the locality characteristic of the program behavior, especially while executing the loop iteration, can affect the decision of inlining[4]. In Figure 1, if *sub1* is large and the instruction cache is already full, then *sub1* could not be merged. Otherwise, each additionally copied instruction of *sub1* will miss on each iteration. Secondly, the activation stack overflow might happen after inlining a callee procedure with heavy local variable declarations into some procedure in the recursion[7]. Thirdly, the program/procedure size explosion also limits the possible inlining. Finally, more register pressure introduced by inlining might result in a longer execution time[5, 6]. If the number of callee registers usage plus the number of live registers at the procedure call site exceed the total amount of available registers, then the inlining on this call site should be refused. However, except for this last situation, we can still reduce the procedure call overhead by applying the interprocedural register allocation to them. In effect, we could also eliminate a few overheads under the last situation if we assign the common global variables to the same dedicated registers across the procedure

Figure 2: Doing interprocedural register allocation on-ly.

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

boundaries.

If we do the interprocedural register allocation alone, there incurs a large amount of unnecessary saves/restores operations. For example, in Figure 2, the executing frequencies of call sites *a*, *b*, and *c* are 80, 70, and 15, respectively. Because procedure *Z* is called by procedure *X* and procedure *Y*, we should take the registers usage of both *X* and *Y* into account. If the register pressure on the call site *c* is very heavy and on the call sites *a* and *b* are light, then each call stemming from *a* or *b* would suffer from a large amount of unnecessary saves/restores at the entry/exit of procedure *Z*. However, because of *a* and *b* having the high executing frequencies, it is worth doing the inlining on both call sites, if the inlining does not violate any constraint stated above.

4 Program Characteristics

This section exhibits some procedure-call characteristics by a series of experiments on six-selected re-

al programs. Table 1 shows the set of benchmarks that we have selected as the examples in our study. For each benchmark, most of the called C library functions[21] are merged into the testing program with their C source code instead of including them with object code through the linker. The merged source code of those libraries will also participate in the interprocedural optimization so that we could get more improvement opportunities owing to having more completed program code.

The ‘Size’ column in the Table 1 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.

Firstly, we are curious how many *heavy-called procedures* are responsible for most of the dynamic procedure calls. Figure 3 plots the cumulative fraction of all dynamic procedure calls against the fraction of total procedures. Let $callIn_p$ be the number of times the procedure *p* is called, and $T = \sum_p callIn_p$ the total number of dynamic procedure calls. Array *sort_list* is a sorted list that consists of all procedures in increasing *callIn* sequences. *N* is the total number of procedures. The following function is applied to each of the benchmarks in Table 1, and the result is depicted in Figure 3.

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

For any given fraction *f*, the function sums up each *callIn* value of *sort_list* from the first one up to the $f \times N$ in the increasing order. From Figure 3, we find that the majority of the dynamic procedure calls are dedicated to only a small proportion of whole set of procedures. This suggests that if the optimization operations are to be restricted by reasonable time and effort, these heavily called procedures should be identified. And, removing the calling overhead from most of the call sites in which control is transferred to these heavy called procedures will gain very significant improvement.

Figure 4 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 by the fraction *f* of the total general registers *R*; $f \times R$. For any degree of register usage *ru*, the value of the *y*-axis indicates the ratio (**H1**) of total procedures for each benchmark program; where

$$\mathbf{H1} = \frac{\text{number of total elements in set } Q}{\text{total number of procedures}}$$

$$Q = \{p \mid \text{register usage of } p \leq ru \times R\}.$$

Figure 3: Distribution of procedure calls

Figure 4: Degree of register usage on each procedure

For each benchmark program, the register usage fraction of half of the procedures is less than 0.6 (approximately, 19 registers). This means that there is still room for improvement in the register utilization. By means of register allocation across procedure boundaries, the different variables in different procedures would be assigned to different physical registers. Then, a better register utilization can be achieved.

In Figure 5, the number of *free registers* on each call site is depicted. The free register at call site s is a physical register holding no live values across s but carrying something elsewhere within the same procedure. Each point (r, f) of benchmark k in the figure denotes that f percent of all call sites of program k have r free registers. These free registers can be safely used by target callee procedure of the call site without any save/restore actions.

Figure 6 measures the degree of *register pressure* on each call site. The register pressure on call site s is represented as the number of demanded registers for the target callee of s plus the number of live registers across s . The degree of register pressure is the ratio of

Figure 5: Number of free registers on each call site

register pressure to the number of total general registers. For each value rp of register pressure degree, the ratio (H2) of all call sites of each benchmark program is computed; where

$$H2 = \frac{\text{number of call sites with register pressure} \geq rp}{\text{total number of call sites}}$$

The figure shows that only less than thirty percent of call sites for all benchmarks of which the register pressure is greater than degree 1.

Figure 6: Degree of register pressure on each call site

In Figure 7, we re-measure the degree of register pressure against the dynamic calls for each benchmark. The y-axis denotes the accumulative fraction values except that static call site is replaced by real dynamic call. The picture is somewhat different from Figure 6. In this time, each plotted line of benchmarks is shifted right more or less in the direction of x-axis. It means that some call sites with high register pressure are heavily executed in real program execution. This suggests that the inlining optimization should carefully take the register pressure issue into consideration. Otherwise, the optimization might do

Figure 7: Degree of register pressure on each dynamic call

Figure 8: Number of free-use register on each procedure

harm around these call sites as described in Section 3.

In Figure 5, we examine the amount of free registers on each call site. For each called procedure v , we further gather together all free registers information from each call sites of procedure v and then compute the *free-use set* for procedure v . Free-use set of procedure v is a maximum set of common free registers among all call sites of v within all callers of v . Ideally, all registers in free-use set of v could be used safely within procedure v without incurring any extra save/restore code. Figure 8 shows the distribution of the number of free-use register on each procedure for the benchmark programs. For each procedure, the proportion of number of free-use register to total register usage is computed. Figure 9 plots the cumulative fraction (G) of procedures for each

$$G = \frac{\text{number of procedures with at least } x \text{ portion of free-use register}}{\text{total number of procedures}}$$

Figure 9: Percent of free-use register for each procedure

The result exhibited in this section have guided the development of our interprocedural optimization with the aim of reducing the much of the procedure call overheads with the least processing effort.

5 Integrated Approach

This section describes an integrated approach which applies both interprocedural register allocation and inlining on reducing the procedure call overhead. The primary principle of our approach is based on the observation of the program analysis in previous section.

5.1 Compilation Paradigm

Most modern compilers support the technique of separate compilation. Under this mechanism, the entire source program is divided into groups of separate modules. Each module can be modified and compiled independently. Instead of re-compiling the whole program each time, the compiler need only re-compile those modified parts of a program after the debugging or modification. Thus, a great deal of compiling effort could be saved during the program development. With the help of a linker, external references between all separate modules could be resolved.

But a separate compilation mechanism also makes it harder for the compiler to obtain the global information for the entire program. However, interprocedural optimization should take the whole program structure into account when it makes its decisions. And any changes to any procedure might cause another modification to other procedure. This kind of modification, like a “chain reaction”, would spread throughout most

part of the program. The situation is difficult to handle for most compilers.

One solution involves restricting the processing scope of interprocedural optimization within each individual module. The other is to build a smarter compiler environment. In the former approach, the limitation impedes improvement chance. For example, inlining can not be performed if the procedure bodies of caller and callee are not in the same module.

We adopt the latter approach and we have constructed 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 result of the interprocedural analyzer. Therefore, our compiler can support separate compilation. Figure 10 illustrates this compilation system.

Firstly, we compile all modules of the source program without interprocedural optimization. In addition to generating the executable code for the profiler, the pre-compilation also outputs the program call graph and the loop structures information of each procedure. Then, after the register allocation, the inter-procedural analyzer reads in the profile information, program call graph, loop structures information, and the register usage records. The interprocedural analyzer makes the operation decision and generates the records of necessary modification action. Finally, the modifier does the actual modification according to those action records generated by the analyzer.

5.2 Program Structure Information

During program pre-compilation phase for the profile, lots of program structure information is gathered. These information includes

- Static program call graph[17] –
The static program call graph is constructed after recognizing all pairs of relation between caller and callee. It includes all accessible routines, the routine whose source code body is touchable in compilation, and all static procedure call arcs.
- Candidates of implicitly called procedures –
The implicitly called procedures are those procedures that can be called through the function pointer. It is hard to identify accurately all callers of any specified procedure only from the plain program text, especially for languages in which

Figure 10: Compilation organization

system or library functions might implicitly call user defined function¹. Instead of doing the sophisticated inter-procedural dataflow analysis, we treat all implicitly called procedures as the candidate targets of every implicit call site. All procedures which addresses are computed in any expression or preloaded at any constant are the candidate of implicitly called procedures.

By augmenting previous static program call graph with information of implicitly called candidates, we can get a conservative program call graph which is already suitable for correctly interprocedural optimization. However, these implicitly called candidates should be carefully processed carefully during the interprocedural optimization.

- Loop-structure –
A loop structure in programming language is a typical *locality* region in which the same program code is likely to execute more than once. As far as the cache miss issue is concerned, the whole of the loop code should be in cache while the loop is executed. Therefore, identifying the loop region in each procedure is necessary for the later phase of inlining.

In order to measure the size of loop precisely, the loop structure is extracted at the intermediate

¹For example, the *qsort* and *bsearch* functions in C language request user to provide the user-defined comparison function.

code level by identifying the strongly connected components using the Tarjan algorithm[20].

All above information is gathered together and output to the interprocedural analyzer.

5.3 Register allocator

In order to provide a more accurate estimate of register pressure on each call site and register usage information of each procedure to the interprocedural analyzer, we do the interprocedural register allocation and the inlining after the intra-procedural register allocation phase.

Our compiler divides the register file into the *general registers set* and the *dedicated registers set*. The general registers set is allocated to the local variables, the temporaries, or the global variables within each procedure. But the dedicated registers set is left to the common global variables across the procedure boundaries in the later interprocedural optimization phase. Within each individual procedure, our register allocator will allocate the general registers to as many of those frequently referenced global variables as possible. To make the contents of these promoted global variables consistent in the whole program, the allocator should insert the necessary load/store code for these global variables both at the entry/exit point of the procedure and around all call sites of the procedure.

Also, the interference graph used in our register allocator is composed of the live-range-conflict graph and the scheduling-conflict graph[11]. The graph can help our register allocator to get a balance between the tradeoff of the register allocation and instruction scheduling[12].

5.4 Data Base

The data base shown in Figure 10 is a pool used for keeping both the register usage information of each procedure which is provided for the interprocedural analyzer and the modification records generated by the interprocedural analyzer after the decision making.

5.5 Inter-procedural optimization

Because the inlining operation might change the original structure of the program call graph, we do the inlining before the interprocedural register allocation.

Inlining and common global variables reassignment

In the previous register allocation phase, the frequently referenced global variables in each referencing procedure have already been promoted to the general registers in the scope of each individual procedure. However, a further improvement can be achieved if the common promoted global variables between any two connected procedures on the program call graph are assigned to the same registers. The dedicated registers, left in the previous phase, now are used for carrying the common promoted global variables across the procedure boundaries. Then, one could get rid of those unnecessary store/reload code for the common promoted global variables around the procedure calls.

In order to 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 the dedicated registers to those common frequently referenced global variables within these regions. Applying the inlining operation to such region can also result in a dramatic performance improvement. Fortunately, Figure 3 in previous section shows us that most part of the dynamic procedure calls are dedicated to only small portion of the whole procedures. Therefore, the regions which we are looking for would not be too large. Streamlining those procedure calls within these regions can remove most procedure call overheads but only with little compiling effort.

Our analyzer identifies the *critical regions*, the regions covering most dynamic procedure calls, on the program call graph and does the inlining within these critical regions, instead of searching and examining each procedure call throughout the entire program call graph. In effect, in our compiler, the inlining and common global variables promotion are applied on the same data structure in one pass.

Here, the approach we adopted in this phase is briefly described:

1. Firstly, the analyzer identifies the critical region according to the calling execution times of each procedure. 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 11, *F*, *G* and *C* are those heavily called procedures. And, two critical region sets are picked out as shown in the Figure.
2. Secondly, it selects the common promoted global variables within the critical region. In Figure

Figure 11: Identify critical region.

Procedure	F	G	D	E
The locally promoted global variables	(x, r3) (z, r6)	(x, r5) (y, r3) (z, r2)	(x, r1)	(x, r4) (y, r2)
remove_entry	{ x }	{ x, z }	empty	empty
remove_around	(x, G) (z, G)	empty	(x, F)	(x, F)

x → rd1
z → rd2

Figure 12: Select the common global variables across procedure boundaries.

Figure 13: Intermediate code before the common global variables reassignment.

12, the secondary row on the table gives those locally promoted global variables and their originally associated register which was allocated by the intra-register allocator in the last operation phase. The global variables x and z are selected and assigned with the dedicated registers $rd1$ and $rd2$, respectively. The *remove_entry*[p] is the set of locally promoted global variables whose associated initialization code can be eliminated from the entry/exit point of the procedure p . The *remove_around*[p] is the set of pairs (x, q) such that the associated store/reload code for the promoted global variable x can be eliminated from all the call sites calling to the callee procedure q in the procedure p .

- Then, the analyzer modifies the intermediate code guided by the result of the common global variables selecting. Figure 13 and Figure 14 illustrate this operation.
- Finally, the analyzer does the inlining within each critical region under those constraints described in section 3.

Reduce save/restore at procedure calls

After inlining, the analyzer re-examines all remaining non-inline procedure calls throughout the whole program call graph. The primary principle of this phase for further reducing the number of save/restore operations at procedure calls is trying to take advantage of the free registers set at each call site. For example, in Figure 15, there

Figure 14: Intermediate code after the common global variables reassignment.

Figure 15: Take advantage of the free registers.

are four free registers $\{r1, r2, r4, r7\}$ at the call site a in procedure X . If procedure Y has only one caller X and the unique call site a in procedure X , we can eliminate the save/restore code for $\{r1, r2\}$ at the entry/exit point of procedure Y . This can be done because both registers $r1$ and $r2$ do not hold live values across the call site a .

From Figure 9 in previous section, we find out that the largest proportion of local register usage in almost half the procedures for each benchmark can be freely used without extra save/restore operations. In real implementations, however, we do not propagate the register usage of the implicitly called procedure to any of its callers. This is because missing any one caller's register usage information for a procedure might cause the compiler difficult to take care of all call-in paths of that procedure. As a result, eliminating any

Figure 16: Example of hierarchical interval partition.

save/restore code for a register might be wrong.

The algorithm of this phase is briefly described as follows:

- (a) 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[10]. Based on this property, the register upstrut can be applied safely in each interval. Figure 16 gives an example of interval partition.
- (b) Compute the *localUse*, *localSpill* and *upthrust* register set for each procedure within each interval.
 - The *localUse* set is the original local register usage set of each procedure generated by the register allocator in the previous phase.
 - The *localSpill* set is the set of registers which is used in the callee and these registers are live at some call site of some caller so that they must be saved in the callee before use.
 - The *upthrust* set is the set of registers which do not hold any live value at every call site in all callers, so that they need not be saved at the callee side. A register in the upthrust set is either in the *localUse* set of the caller or not. In the latter case, it becomes the *extraSpill* register of the caller.
- (c) In bottom-up ordering, propagate the upthrust register set from the leaf procedures in the interval toward the interval header.

Figure 17: Example of register upthrust.

- (d) Optimize the placement of the save/restore code, guided by the profile information.
- Figure 17 gives an example of register upthrust.

5.6 Modifier

The modifier firstly reassigns the common promoted global variables within the critical region with the dedicated register and examines all procedures within each critical region in a bottom-up order, then 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. Finally, according to the result of register upthrust, the modifier inserts the suitable store/reload code to preserve the contents of registers across the procedure calls.

6 Results

The integrating approach described in this paper has been implemented in a prototype C compiler. The compiler uses the modified GNU C compiler[22] as its front end part. Our experimental results are measured on a RISC simulator which is modified from the *dlasim*[19]. There are 32 general purpose registers in our base architecture of which 4 are dedicated for common global variables usage and 28 are free usage. The comparison is built with interprocedural optimization and without this optimization.

Table 2 shows the amount of dynamic procedure calls contributing to the program execution time at real run. The values denote the frequencies of procedure calls for each benchmark program at run time. They do not display the burden of procedure call overhead. However, the real amount of procedure call overhead in program execution will be greater than the values shown at Table 2, approximately 2 to 3 times.

Program	Ratio (%)
compress	6.19
espresso	0.91
grep	6.08
lex	1.48
li	5.89
yacc	8.34

Table 2: Ratio of dynamic procedure calls to total execution

The performance improvement from the phase of inlining and common global variables reassignment is shown at Table 3. 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 'Ration' column in the Table 5 indicates the amount of the code increase from the inline expansion. The values are measured by the ratio of code increase to the original code size.

Table 4 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

Program	Removing(%)		SpeedUp
	Static	Dynamic	
compress	2.00	99.99	1.11
espresso	0.06	52.14	1.06
grep	0.38	9.49	1.00
lex	0.43	46.23	1.02
li	8.99	82.47	1.56
yacc	4.49	46.36	1.09

Table 3: Result of inlining and common globals reassignment

upthrust. Again, the speed-up is calculated based on the execution cycles on our simulator.

Program	Reducing(%)		SpeedUp
	Static	Dynamic	
compress	68	96	1.136
espresso	40	2	1.002
grep	39	45	1.024
lex	62	63	1.051
li	16	25	1.102
yacc	46	64	1.078

Table 4: Result of register upthrust

Table 5 presents the speed-up comparison of code produced by our compiler w/o the integrating approach. It shows that the performance improvement of average 1.21 can be achieved when the compiler applies both inlining and interprocedural register allocation during the interprocedural optimization phase.

Program	Speedup	Ration
compress	1.244	12.80
espresso	1.066	0.15
grep	1.025	0.07
lex	1.076	0.03
li	1.655	40.55
yacc	1.177	16.49

Table 5: Result of integrating approach

7 Conclusions

The complementary relationship of the interprocedural register allocation and inlining on reducing the procedure call overhead has been described. Inlining, together with interprocedural register allocation, could still reduce the procedure call overhead without any extra penalty even if the inlining might incur additional miss, stack overflow, or program size explosion. 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 inlining and common promoted global variable reassignment has been applied to the critical regions. Most of the dynamic procedure calls overhead could be eliminated with small processing effort.

The register upthrust technique takes advantage of the free registers set at each call site. This method allows the same registers to be used both within the caller and callee without introducing any additional save/restore code either at the caller or at the callee and achieves a higher register utilization. Furthermore, the profile information guides the optimal placement of register save/restore code.

The experimental results show that applying both inlining and interprocedural register allocation on reducing 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] S. McFarling, "Procedure Merging with Instruction Caches," *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 71–79, June 1991.
 - [5] J. W. Davidson and A. M. Holler, "A Study of a C Function Inliner," *Software-Practice and Experience*, Vol. 18(8), pp. 775–790, August 1988.
 - [6] 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.
 - [7] P. P. Chang, S. A. Mahlke, and W. Y. Chen, "Profile-guided Automatic Inline Expansion for C Programs," *Software-Practice and Experience*, Vol. 22(5), pp. 349–369, May 1992.
 - [8] M. D. Smith, M. S. Lam, and M. A. Horowitz, "Boosting Beyond Static Scheduling in a Superscalar Processor," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 344–354, May 1990.
 - [9] J. W. Davidson and D. B. Whalley, "Methods for Saving and Restoring Register Values across Function Calls," *Software-Practice and Experience*, Vol. 21(2), pp. 149–165, February 1991.
 - [10] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.
 - [11] 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.
 - [12] W.-C. Hsu, "Register Allocation and Code Scheduling for Load/Store Architectures," Computer Sciences Technical Report 722, Ph.D. Thesis, Computer Sciences Department, University of Wisconsin-Madison, October 1987.
 - [13] F. C. Chow and J. L. Hennessy, "The Priority-Based Coloring Approach to Register Allocation," *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 4, pp. 501–536, October 1990.
 - [14] P. A. Steenkiste and J. L. Hennessy, "A Simple Interprocedural Register Allocation Algorithm and Its Effectiveness for LISP," *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 1, pp. 1–32, January 1989.
 - [15] Keith D. Cooper, Ken Kennedy, and Linda Torczon, "The Impact of Interprocedural Analysis and Optimization in the Rⁿ Programming Environment," *ACM Transactions on Programming Language and Systems*, pp. 491–523, October 1986.
 - [16] Jack W. Davidson, Member, IEEE, and Anne M. Holler, "Subprogram Inlining: A Study of its Effects on Program Execution Time," *IEEE Transactions on Software Engineering*, pp. 89–102, February 1992.
 - [17] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a Call Graph Execution Profiler," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp. 120–126, 1982.
 - [18] Stephen Richardson and Nahadevan Ganapathi, "Interprocedural Analysis vs. Procedure Integration," *Information Processing Letters* 32, pp. 137–142, August 1989.
 - [19] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., Reading, San Mateo, 1990.
 - [20] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press, Addison-Wesley, Reading, 1990.
 - [21] P. J. Plauger, *The Standard C Library*. Prentice-Hall, Reading, 1992.
 - [22] R. M. Stallman, "Using and Porting GNU C," Free Software Foundation, Inc., January 1991.