Inline Function Expansion for Compiling C Programs

Wen-mei W. Hwu and Pohua P. Chang

Coordinated Science Laboratory
University of Illinois
1101 W. Springfield Ave.
Urbana, IL 61801
(217) 244-8270
hwu@bach.csg.uiuc.edu

Abstract

Inline function expansion replaces a function call with the function body. With automatic inline function expansion, programs can be constructed with many small functions to handle complexity and then rely on the compilation to eliminate most of the function calls. Therefore, inline expansion serves a tool for satisfying two conflicting goals: minizing the complexity of the program development and minimizing the function call overhead of program execution. A simple inline expansion procedure is presented which uses profile information to address three critical issues: code expansion, stack expansion, and unavailable function bodies. Experiments show that a large percentage of function calls/returns (about 59%) can be eliminated with a modest code expansion cost (about 17%) for twelve UNIX* programs.

* UNIX is a trademark of the AT&T Bell Laboratories.

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.
© 1989 ACM 0-89791-306-X/89/0006/0246 $1.50

1. Introduction

Large computing tasks are often divided into many smaller subtasks which can be more easily developed, understood, and reused. Function definition and invocation in high level languages provide a natural means to define and coordinate subtasks to perform to original task. Structured programming techniques therefore encourage the use of functions. Unfortunately, function invocation disrupts compile-time code optimization such as register allocation, code compaction, common subexpression elimination, and constant propagation. The decreased effectiveness of these optimization techniques increases memory accesses, decreases pipeline efficiency, and increases redundant computation.

Emer and Clark reported, for a composite VAX workload, 4.5% of all dynamic instructions are procedure calls and returns[1]. If we assume equal numbers of call and return instructions, the above number indicates that there is a function call instruction for every 44 instructions executed. Berkeley RISC researchers have reported that procedure call is the most costly source language statement[2].

1.1. Existing Remedies

Some recent processors provide hardware support for minimizing the extra memory accesses due to function calls. For example, the Berkeley RISC processors provide overlapping register windows to reduce the number of memory accesses required to save/restore registers and to pass parameters[2]. Another example is the CRISP processor that uses stack buffers to capture the memory accesses

to local variables so that register allocation crossing function calls can be simulated in hardware[3]. The problems with these hardware approaches are that they tend to consume a significant amount of hardware, stretch the processor cycle time, and provide little assistance for enlarging the scope of compiler code optimization.

In the software realm, inter-procedural register allocation schemes have been shown to reduce the register save/restore cost across function call boundaries[4]. Callers and callees can also communicate parameters and results through a small number of registers[5]. Wall has shown that combining profiling and inline expansion, link-time register allocation is comparable in performance to hardware register window schemes[6]. Inter-procedural analysis have been shown to be effective in reducing the negative effects of function calls on the code scheduling and other code optimization techniques. These software remedies assume that frequent function calls can not be avoided. If most of the function calls can be eliminated, these complicated remedies would be unnecessary.

1.2. Inline Function Expansion

Inline expansion replaces a function call with the function body. Inline expansion removes the function calls/returns costs and provides larger and specialized execution plans to the code optimizers. With automatic inline function expansion, the advantages of using functions remain and the costs are reduced. In a recent study, Allen and Johnson identified inline expansion as an essential part of an optimizing C compiler[7-11]. They gave a few critical reasons for implementing inline expansion. First, the variable aliasing problem becomes less onerous after inline expansion. Second, the code optimizer can work on the real effects of the callee after inlining. Third, inlining function calls contained in loops may increase the opportunities for vectorization.

Several code improving techniques may be applicable after inline expansion. These include register allocation, code scheduling, common subexpression elimination, constant propagation, and dead code elimination. Richardson and Ganapathi have discussed the effect of inline expansion and code optimization across procedures[12].

Many optimizing compilers can perform inline expansion[11,13-16]. IBM PL.8 compiler does inline expansion of all leaf-level procedures 13 . In the GNU C compiler, the programmers can use the keyword **inline** as a hint to the compiler for inline expanding function calls16 . In the MIPS C compiler, the compiler examines the code structure (e.g. loops) to choose the function calls for inline expansion[15]. It should be noticed that the careful use of the macro expansion and language preprocessing utilities has the same effect as inline expansion, when inline expansion decisions are made entirely by the programmers.

The IMPACT-I (Illinois Microarchitecture Project using Advanced Compiler Technology - stage I) C compiler expands function calls to increase the effectiveness of compiler code optimization[17-19]. As far as the hardware is concerned, the goal of inline expansion is to reduce the function calls so that mechanisms such as register windows and stack buffers become unnecessary. For compiler code optimization, the inline expansion serves to enlarge the scope of register allocation, code scheduling, and other optimizations. The IMPACT-I Profiler to C Compiler interface allows the profile information to be automatically used by the IMPACT-I C Compiler. The inline expansion is based on execution profile information to ensure that only the important function calls are expanded. It is critical that the inputs used for executing the equivalent C program are representative. Therefore, this approach is more suitable for characterizing realistic programs for which representative inputs can be easily collected.

1.3. Organization of Paper

Section two discusses major implementation issues and potential hazards. Section three describes a simple inline expansion procedure. Section four shows the experimental results. Finally, we give some concluding remarks in section five.

2. Implementation Issues

The core inline expansion algorithm is fairly simple. Most of the implementation difficulties are due to hazards, missing information, and minimizing the compilation time. The implementation issues are listed as follows.

- *Determine when inline expansion should be applied.*
- *Propose an appropriate program representation.*
- *Select call sites for inline expansion.*
- *Avoid hazards.*

- *Prevent code explosion.*
  - *Prevent control stack overflow.*
- *Evaluate cost.*
- *Inline a function.*
  - *Code duplication.*
  - *Variable renaming.*
  - *Changes to variable declarations.*
- *Resolve missing information.*
- *Eliminate unreachable functions.*
- *Reduce the number of expansions.*
  - *Order the expansion sequence.*

2.1. When to Perform Inline Function Expansion

Inline expansion should be performed before other code optimizations, such as constant folding and dead code elimination. Therefore, tt is natural to perform inline expansion at compile-time. Another advantage of performing inline expansion at compile-time is that the program structures, such as loop and if_then_else, are visible and can be used to make inline expansion decisions. Yet another advantage is that if inline expansion can be applied to a system-independent three-address code, inline expansion can become a part of a portable front-end. However, inline expansion requires the function call characteristics of all functions including those of library functions, and thus imposes restrictions to the separate compilation.

An alternative is to implement inline expansion at the link time. Because all functions are available at the link time, inline expansion can naturally be performed without sacrificing separate compilation. Also it is easier to interface to a profiler at this level. The disadvantage is that many code optimizations have to be applied after inline expansion, and therefore, have to be performed at the link time.

2.2. Program Representation

Weighted call graphs are suitable for inline expansion. A weighted call graph $G = (N, E, main)$ is characterized by three major components: N is a set of nodes, E is a set of arcs, and *main* is the first node of the call graph. Each node in N is a function in the program and has associated with it a weight, which is the expected execution count of the function. Each arc in E is a static call site in the program and has associated with it a weight, which is the expected invocation count of the call site. Finally, *main* is the first function executed in this program.

Each node contains three major pieces of information: 1) the body of the function, 2) the node weight, and 3) a set of outgoing arcs which points to the callees. Each arc contains four essential pieces of information: 1) a unique identifier, 2) the caller, 3) the callee, and 4) the arc weight. It is necessary to assign each arc a unique identifier because there may be several arcs between the same pair of caller and callee. Later, we will also associate each arc with a *status* attribute which tells us whether this arc should be considered for inline expansion, rejected for inline expansion, or inline expanded.

The node weights and arc weights may be determined either by program structure analysis or by profiling. Since a node may be entered from any one of its incoming arcs, it is necessary to know the weights of all outgoing arcs associated with a particular incoming arc. Therefore, after inline expansion the arc weights remain accurate. It is assumed in the remaining part of this section that we have complete and accurate weights for all nodes and arcs.

The most important reason for using a weighted call graph is because graph algorithms are well understood. For example, detecting recursion is equivalent to finding cycles in the call graph. For another example, function-level dead code removal is equivalent to finding unreachable nodes from the *main* node.

Inlining a static call is to absorb a frequently executed arc by expanding a copy of the callee to the caller at a particular call site designated by the instruction id attribute of the arc. The removal of an arc may cause the callee to become unreachable from the *main* node. In this case, the original copy of the callee function may be eliminated.

When a node has an outgoing arc to itself, the corresponding function is obviously recursive. There are standard ways of removing tail recursion and expanding simple recursive functions. In the rest of this paper, we call a cycle which contains only one node a simple recursion.

2.3. Select Expansion Sites

We do not deal with simple recursion in this paper. Although a simple recursive function can be inline expanded, recursive call(s) made from the function must

be to the original copy of the function and thus only the first iteration of the recursion can be absorbed by inline expansion. This may be useful only if the recursive calls are rarely made.

Considering the general inline expansion decision, it is desirable to expand as many heavily weighted arcs as possible. However, several things prevent us from doing so. First, to avoid excessive code expansion, it is necessary to set an upper limit to the program size. Second, expanding a call which uses a lot of control stack space into a recursion may cause control stack explosion. Below, we describe each of the two possible hazards in more detail.

2.3.1. Code Explosion

In order to expand a call site, the body of the callee must be duplicated and the new copy of the callee must be transformed and absorbed by the caller. Obviously, this code duplication process increases program space. It may be true that many C functions are called once, and thus the original copies of these call-once functions can be eliminated by finding unreachable nodes from the *main* node after inline expansion. In a later subsection, we will show it is not always possible to eliminate all unreachable nodes in an incomplete call graph. In fact, most call graphs of large programs are incomplete due to system calls.

It is necessary to set an upper bound on the instruction space usage. This limit may be specified as a fixed number or as a function of the original program size. For virtual space limited machines, such as PDP11, a fixed upper limit for the program space is mandatory. On the one hand, the problem with using a fixed limit is that the largest program ever going to be compiled in the system can not be known. On the other hand, setting the upper limit as a function of the original program size tend to favor large programs.

2.3.2. Control Stack Overflow

Parameter passing, register saving, local variable declarations, and returned value passing associated with a call can all contribute to the control stack usage. A summarized control stack usage can be computed for each call site. A control stack overflow may occur when inline expanding a call site with extremely high control stack usage into a recursion. For example, a recursive function m(x) and another function n(x) are defined as follows.

```
m(x) { return (x?m(x-1)+m(x-2)+n(x):1); }
n(x) { int y[100000]; ..... }
```

If m(x) tends to be called with a large x value, expanding n(x) will dramatically increase the control stack size.

To prevent control stack overflow, a fixed limit on the control stack usage can be imposed for inline expanding a call into a recursion.

2.3.3. Cost Function

Given a weighted call graph, a cost function is provided to determine the most plausible arc(s) for inline expansion. Then, the inline expansion problem can be formulated as an optimization problem which attempts to minimize the cost of inline expansion in an arbitrary number of inline expansion steps. Because the real costs of code expansion and the real benefits of inlining a static call site are unknown at the compile time, it is not possible to derive optimal solutions. Also, the search space of this optimization problem is too large for a practical implementation. Therefore, it is desirable to use heuristic and keep the search space small. In designing the cost function, we not only need to prevent hazards but also need to eliminate unimportant arcs.

```
cost(G, arc Ai) =
  if ((caller(Ai) is recursive) and
    (control_stack_usage(Ai)>BOUND)) then
    cost = INFINITY;
  else
  if (weight(Ai)<MIN) then
    cost = INFINITY;
  else
  if (instruction_space_after_expansion
   (G, Ai)>MAX) then
    cost = INFINITY;
  else
    cost = Code Expansion Cost of Ai -
     Benefits of Inlining Ai;
```

The code expansion costs are the increase in memory space to accommodate the program and the effect on the instruction cache memory performance. Precise costs can not be obtained at compile-time. A rough estimate of the cost may be derived by multiplying a constant number to the marginal increase in the code size.

Accurate benefits of inline expansion are equally difficult to obtain. If one assumes that the register

save/restore and the control transfer costs dominate the costs of function calls, and if these costs are approximately equal for all calls, the *benefit* term can be eliminated from the cost function, because this term would be the same for all call sites.

2.4. Inline a Function

After determining the expansion sites and the sequence of expansions, physical expansion step proceeds. The work involved in inlining a call site include three major tasks: 1) duplication of the callee, 2) variable renaming, and 3) modifications of the symbol table.

The work required to duplicate the callee is trivial. However, the actual implementation difficulty is in caching the definitions of the most frequently inlined functions in memory to reduce the number of file reads.

Variable renaming of the formal parameters and the local variables must be made for the new copy of the callee before inserting the code into the caller. For source code level inline expansion, a new scope may be introduced to simplify the local variable renaming and only formal parameters need to be renamed. New local temporary variables may be introduced to buffer the results of the actual parameters. And the formal parameters are replaced by these temporary variables. Finally, copy propagation and other optimizations can be applied to eliminate unnecessary overhead instructions.

2.5. Resolve Missing Information

When the compiler or the linker fails to access the internal function calling and variable usage characteristics of some functions, the call graph is incomplete. For example, if system calls may access user data and call user functions, a large piece of the call graph can not be detected at compile-time. Similarly, library functions and archives may not be accessible by the compiler. In later sections of this paper, an inaccessible function to the compiler is called an external function.

Call through pointer is a peculiar feature of C that introduces ambiguities to the call graph. If a call through pointer may results in one of several functions, the call graph must appropriately reflect this situation in order to identify all recursions.

It is appropriate to introduce special arc types to represent call to external functions and call through pointers. Interprocedural dataflow flow analysis may reduce the potential callee sets of call-through-pointer sites.

Given an incomplete call graph, one must make the worst case assumptions. It must be assumed that external functions may call any function in the call graph. Therefore, a call to an external function may go to any function in the call graph. This results in many more cycles in the call graph and the original copy of an inlined call-once function can no longer be deleted.

Because calls through pointers are rare, it is possible to assume all calls through pointers reach every function which can be called through pointer. This maximum set is simply the set of all functions whose addresses have been used in computation. However, in the presence of external functions, precise computation of this maximum set is no longer possible. Again, we have to assume that all functions can be reached from a call through pointer.

2.6. Eliminate Unreachable Functions

Because programs always start from the *main* function, any function which is not reachable from the *main* function will never be used and can be removed. A function is reachable from the *main* function if there is a (directed) path in the call graph from the *main* function to the function, or if the function may be called by the exception handler or be activated by some other asynchronous events. In C, this can be easily detected by identifying all functions whose addresses are used in computations. However, if there is external function, it must be assumed that all functions can be reached.

2.7. Reduce the Number of Expansions

An inline expansion decision not only selects the expansion sites but also determines an order in which these expansion sites should be expanded. Let (A→B) denote inline expanding A into B, [X Y Z] be an ordered sequence of events, starting from X, to Y and finally to Z, and {X Y Z} denotes three unordered events which all must be carried out. The exact ordering of the expansion sequence is important. For example, [(A→B) (B→C)] is different from [(B→C) (A→B)].

[(A→B) (B→C)] == [(B' = A+B) (C' = B'+C)]

[(B→C) (A→B)] == [(C' = B+C) (B' = A+B)]

Several different sequences may achieve the same final effect. For example, [(C→A) (C→B)] is equivalent to [(C→B) (C→A)].

[(C→A) (C→B)] == [(A' = A+C) (B' = B+C)]

[(C→B) (C→A)] == [(B' = B+C) (A' = A+C)]

In complex situations, the number of expansions required to accomplish all expansions depends on the ordering of the expansion. For example, {(D→C) (C→A) (C→B)} can be realized by at least two sequences.

[(C' = D+C) (A' = C'+A) (B' = C'+B)]

[(A' = A+C) (B' = B+C) (A'' = A'+D) (B'' = B'+D)]

It can be observed that larger the fan-in degree of the callee, more likely excessive number of expansions will occur. Since in structured programming a low-level function may be called from many different sites, this problem is severe. Therefore, among several sequences which offer comparable benefits, it is critical that the shortest sequence be used.

3. Our Approach

A simple inline function expander has been implemented in the IMPACT-I C compiler. In order to determine the execution counts of the functions and the numbers of invocations made from the call sites, profiling provides program dynamic information. The execution counts are converted into *weights* of the call graph. Then a linear sequence of the functions in the call graph is determined. Inline expansion is constrained to follow this linear sequence to minimize the number of expansions. Based on this linear order and the *weight* information, some call sites are selected for expansion. Finally, the actual expansion process is executed.

3.1. Profiling

A system independent profiler has been integrated into the IMPACT-I C compiler. The profiler accumulates the average run-time statistics over many runs of a program. From the profile information, the IMPACT-I C compiler can determine the execution counts of all instructions and the frequencies of each of the possible directions of branch instructions. From the execution and branch frequencies, the node weights and arc weights of the call graph can be inferred. The node weight is simply the number of times a function is called in a typical run of the program. The arc weight is the execution count of a call instruction.

3.2. Weighted Call Graph Construction

The call graph construction procedure is straight forward, except when dealing with external functions and call through pointers, for which we always assume the worst case behavior.

When a function calls an external function, it is assumed that all functions may be reached. In order not to create a large number of outgoing arcs for the function, a special node, $$$, is created to represent the summarized effect of external functions. A function which calls external functions requires only one outgoing arc to the $$$ node. In turn, the $$$ node has many outgoing arcs, one to each user function. One arc to the $$$ node sufficiently represent the effect of calling external functions, because calls to external functions can not be inlined expanded, cycle detection algorithm works, and conservative function-level dead code elimination is used.

For similar reasons, we use another special node, ###, to represent the effect of calling through pointers. We simply assume all functions which may be called through pointers can be reached by all calls made through pointers. When there is at least one call to an external function, a call through pointer is assumed to be able to reach any user function. This allows us not to implement a specialized inter-procedural dataflow analysis for this purpose. An inter-procedural analysis for detecting minimal callee sets for all call sites provides little help because of calls to external functions.

1. Allocate a new node for each function;

2. Connect nodes corresponding to the static calls;

3. Handle calls to external functions and calls through pointers as described above;

3.3. Linearization

In order to speed up the expansion site selection procedure and also reduce the number of file writes in the physical inline expansion step, inline expansion is constrained to follow a linear order. A function X can be inlined into another function Y if and only if function X appears before function Y in the linear sequence. Therefore, all inline expansions pertaining to function X must already have been done before function Y is processed.

This also allows us to cache the most recent definitions of functions. A write-back replacement policy can be implemented.

When determining this linear sequence, functions which tend to be absorbed by other functions should be placed in front of the list. For example, if the call graph is a tree, it is desirable to have all leaf-level functions appear in front of the linear list.

We have implemented a simple heuristic, which places functions randomly into the list, and then sort the functions by their execution counts. The most frequently executed function leads the linear list. We have chosen this heuristic because functions which are executed frequently are usually called by functions which are executed less frequently. Also, it is difficult to define the levels of the call graph with the presence of cycles.

1. Place all nodes in a list randomly;
2. Sort the list by the node weights;

3.4. Inline Expansion Decisions

Because we force the inline expansion to follow a linear order, all arcs which violate this linear order can be marked as *not_expandable*. Also all arcs connecting the ### node and the $$$ node are also *not_expandable*. All other arcs are marked as *expandable*. We then consider from the most frequently executed *expandable* arc to the least frequently executed *expandable* arc, excluding arcs whose weights are below a threshold value to cut down the compilation time.

1. Place all expandable arcs randomly in a list;
2. Sort the list according to the arc weights;
3. From the most important to the least important arc,
if (cost(Ai) < INFINITY) then
mark Ai as to_be_expanded;

It should be noticed that inline expansion changes code sizes of functions. Therefore, the code size of each function body must be re-evaluated as new function calls are considered for expansion.

3.5. Physical Inline Expansion

According to the linear ordering which we have determined in the expansion site selection step, expand all *to_be_expanded* arcs. As we have described in the previous section, code duplication, variable renaming and symbol table update are applied to each expansion site.

4. Experimentation

The purpose of this experiment is to answer the following questions.

1. *How many call sites are free of hazards and have significant benefits when inlined?*
2. *For all call sites which are considered for inline expansion, how many dynamic calls can be eliminated?*
3. *How much code expansion is incurred by inline expansion?*
4. *Do most programs have similar static and dynamic function call characteristics?*
5. *How frequently are the function calls executed before and after inline function expansion.*

This experiment consists of four major steps. First, we select a benchmark suite of twelve frequently used UNIX programs. Second, representative inputs for each benchmark are applied to establish reliable profile information. For example, we select from many sources 20 files of C programs, ranging from 100 to 3000 lines, as inputs for *cccp*, the GNU C language preprocessor. We also make special effort to exercise as many program options as possible. Third, the benchmarks are recompiled using profile information. Finally, we measure the effect of inline expansion.

4.1. Benchmarks

Table 1 summarizes several important characteristics of our benchmarks. The *C lines* column shows the static code sizes of the C benchmark programs measured in the number of program lines. The *runs* column gives the number of different inputs used in the experiment. The *IL's* column gives the average dynamic code sizes of the benchmark programs, measured in number of thousands of intermediate instructions executed in a typical run of the programs. The *control* column gives the average dynamic count of thousands of control transfers, other than function call/return, executed in a typical run of the programs. The *input description* describes the nature of the inputs used in the experiment.

Note that we use the dynamic counts of intermediate instructions rather than those of machine instructions to keep the data general. The experiment involves measurements based on more than three billion intermediate instructions worth of program execution. The benchmark

programs exhibit very different code sizes, control structures, and applications. There does not seem to be any direct relation between the static and dynamic code sizes of these benchmark programs.

4.2. Static Characteristics

Table 2 shows the static function call characteristics. The *total* column gives the number of different function calls in the static program. Note that different static function calls could be calling the same function. We categorize the static function calls into four types. The *external* column gives the percentage of static function calls to functions whose body are unavailable to inline expansion, and to system functions (syscall). The *pointer* column gives the percentage of static function calls through pointers. Function calls through pointers defeat inline expansion. The *unsafe* column gives the number of static function calls which either introduce function bodies into recursive paths and may cause control stack explosion, or have an estimated execution count less than 10. The *safe* column gives the percentage of the static functions which can be safely inline expanded. Only the safe function calls are considered for inline expansion.

All benchmarks show large percentages of *unsafe* functions (average about 65%). Only very small percentages of static calls are considered *safe* (average about 11%). This observation suggests future research in determining whether or not inline expansion decisions based on program structure analysis without profile information are sufficient. Failure to identify the smallest possible set of *safe* static calls may result in excessive code expansion.

The numbers of static call sites are approximately 1/10 of the program sizes measured in lines of C code.

4.3. Dynamic Characteristics

Table 3 presents the dynamic behaviors of function calls. A static function call can correspond to many dynamic function calls. Only those static call sites corresponding to a large number of dynamic function calls should be considered for inline expansion. Note that for most benchmarks, the static calls corresponding to a large number of dynamic calls are safe for expansion. The only exception is wc, where function calls are unimportant because they are invoked very infrequently.

Although the percentages of static *safe* calls are small, *safe* call sites correspond to large percentages of dynamic calls (average about 69%). This means that by expanding few static call sites, a large number of dynamic calls can be eliminated.

The percentages of *unsafe* dynamic calls are amazingly small. This supports our previous observation that a small number of static calls contribute to most of the dynamic calls.

4.4. Effectiveness of Inline Expansion

Table 4 offers the most important results of inline expansion. The *code inc* column gives the percentages of increase in static code sizes due to inline expansion. The *call dec* column gives the percentage of dynamic function calls eliminated by inline expansion. The *IL's per call* column gives the average number of dynamic intermediate instructions executed between dynamic function calls after inline expansion. The *CT's per call* column gives the average number of dynamic control transfers executed between dynamic function calls after inline expansion.

Note that inline expansion mechanism eliminates large percentages of dynamic function calls for function call intensive programs. For programs with less frequent function calls to begin with, the inline expansion mechanism does not eliminate large percentages of dynamic function calls. This is a desirable behavior because the overall goal is to ensure infrequent function calls rather than to achieve high elimination percentages.

After inline expansion, function calls only account for about 1% of the control transfers (see the *CT's per call* column). Therefore, function calls become less important in the hardware design considerations. Also, large scopes for compiler optimizations can be expected for the critical parts of the programs. The code expansion, on the average, is about 17% increase in static code size.

At the time we took these measurements, constant folding and jump optimization were applied before the inline expansion procedure, but not after it. Because inlined call/return instructions were replaced with unconditional jump instructions into/out of the inlined function bodies, we have measured substantially more unconditional branch instructions. The *IL's per call* and *CT's per call* should be somewhat smaller if comprehensive code optimizations have been applied after inline expansion. However, we expect the magnitudes of these reductions to

be small.

After inline expansion, the dynamic *external*, *pointer*, *unsafe*, and *safe* calls correspond to 56.1%, 2.8%, 18.0%, and 23.1% of all dynamic calls respectively. Therefore, better ways to handle *external* functions are desirable. Since most *external* function calls in this experiment are system calls, ways to reduce the number of system calls should be studied. However, providing solutions to this problem is beyond the power of the compiler people, because the system call interface is dictated by the operating system.

5. Conclusion

We have finished the first version of the IMPACT-I C Compiler inline expansion mechanism. This mechanism has the following features to address critical issues for compiling C programs. First, the importance of each function call is estimated via profiling the target program with a spectrum of representative inputs. Second, the function code sizes are estimated in terms of intermediate code size and are updated after each expansion. Third, the function stack frame sizes are estimated in terms of local declarations and are updated after each expansion. Lastly, the identifiers are qualified with proper path names to simplify symbol table management after expansion.

We have shown, for twelve realistic programs, that inline expansion can substantially reduce the function call frequencies. This also results in decent optimization scopes for critical sections of the programs.

We have also pointed out problems with system calls. System calls not only hinder function-level dead code removal, but also become the major cost of function calls, after inline expansion. Further study of ways to reducing system calls is necessary.

In a recent paper, we have obtained good instruction cache performance after inline expansion[18]. Although inline expansion increases the static code size, it greatly reduces the mapping conflict in instruction caches with small set-associativities. We are conducting a detail study of the exact effect of inline expansion on the instruction cache performance.

References

1. J. Emer and D. Clark, "A Characterization of Processor Performance in the VAX-11/780," *Proceedings of the 11th Annual Symposium on Computer Architecture*, June 1984.
2. D. A. Patterson and C. H. Sequin, "A VLSI RISC," *IEEE Computer*, pp. 8 - 21, September, 1982.
3. D. R. Ditzel, H. R. Mclellan, and A. D. Berenbaum,, "The Hardware Architecture of the CRISP Microprocessor," *Proceedings of the 14th Annual International Symposium on Computer Architecture*, Pittsburgh, Pennsylvania, June 2-5, 1987.
4. Fred C. Chow, "Minimizing Register Usage Penalty at Procedure Calls," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 22-24, 1988.
5. R. Sherburne, M. Katevenis, D. Patterson, and C. Sequin, "Local Memory in RISCs," *Proceedings of the International Conference on Computer Design*, IEEE, October 1983.
6. David W. Wall, "Register Windows vs. Register Allocation," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 22-24, 1988.
7. F.E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," *Journal of the Association for Computing Machinary*, vol. 19-3, pp. 137-147, March 1976.
8. Matthew S. Hecht and Jeffrey D. Ullman, "A Simple Algorithm for Global Data Flow Analysis Problems," *SIAM J. Comput.*, vol. 4-4, pp. 519-531, December 1975.
9. Jeffrey M. Barth, "A Practical Interprocedural Data Flow Analysis Algorithm," *Journal of the Association for Computing Machinary*, vol. 21-9, pp. 724-736, September 1978.
10. Z. Li and P-C Yew, "Efficient Interprocedural Analysis for Program Parallelization and Restructuring," *Proceedings of the ACM/SIGPLAN PPEALS 1988 (Parallel Programming: Experience with Applications, Languages, and Systems*, pp. 85 - 99, ACM , New Haven, Connecticut, July 19-21.

11. R. Allen and S. Johnson, "Compiling C for Vectorization, Parallelism, and Inline Expansion," *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 241-249, June 22-24, 1988.

12. Stephen Richardson and Mahadevan Ganapathi, "Code Optimization Across Procedures," *IEEE Computer*, February 1989.

13. M. Auslander and M. Hopkins, "An Overview of the PL.8 Compiler," *Proceedings of the SIGPLAN Symposium on Compiler Construction*, ACM, June 1982.

14. C. A. Huson, *An In-line Subroutine Expander for Parafrase*, M.S. Thesis, University of Illinois at Urbana-Champaign, 1982.

15. F. Chow and J. Hennessy, "Register Allocation by Priority-bases Coloring," *Proceedings of the ACM SIGPLAN Symposium on Compiler Constructions*, pp. 222-232, June 17-22, 1984.

16. R. M. Stallman, *Internals of GNU CC*, Free Software Foundation, Inc., 1988.

17. P. P. Chang and W. W. Hwu, "Trace Selection for Compiling Large C Application Programs to Microcode," *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitectures*, San Diego, California, November 29 - December 2.

18. Wen-mei W. Hwu and Pohua P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," *Proceedings of the 16th Annual Symposium on Computer Architecture*, May 1989.

19. Wen-mei W. Hwu, Thomas M. Conte, and Pohua P. Chang, "Comparing Software and Hardware Schemes For Reducing the Cost of Branches," *Proceedings of the 16th Annual Symposium on Computer Architecture*, May 1989.

benchmark	C lines	runs	IL's	control	input description
cccp	4660	20	585K	111K	C programs (100-3000 lines)
cmp	371	16	135K	30K	similar/disimilar text files
compress	1941	20	981K	155K	same as cccp
eqn	4167	20	1809K	537K	papers with .EQ options
espresso	11545	20	54496K	8522K	original espresso benchmarks*
grep	1302	20	2357K	857K	exercised .+"^$ options
lex	3251	4	152630K	56295K	lexers for C, Lisp, awk, and pic
make	7043	20	7629K	1620K	makefiles for cccp, compress, etc.
tar	1063	14	809K	104K	save/extract files
tee	3186	20	24K	9.5K	same as cccp
wc	345	20	392K	112K	same as cccp
yacc	3333	8	15668K	3935K	grammar for a C compiler, etc.

Table 1. Benchmark characteristics.

benchmark	total	external	pointer	unsafe	safe
cccp	393	15.8%	0.2%	74.3%	9.7%
cmp	40	50.0%	0.0%	47.5%	2.5%
compress	183	37.7%	0.0%	61.7%	0.5%
eqn	463	4.1%	0.0%	79.3%	16.6%
espresso	1466	4.7%	0.8%	64.0%	30.4%
grep	90	20.0%	0.0%	73.3%	6.6%
lex	560	8.9%	0.0%	73.2%	17.9%
make	686	15.2%	0.0%	63.5%	21.4%
tar	445	31.2%	0.0%	63.8%	4.9%
tee	82	40.2%	0.0%	59.8%	0.0%
wc	27	48.1%	0.0%	51.9%	0.0%
yacc	464	19.2%	0.0%	64.7%	16.2%
AVG	408	24.6%	0.1%	64.75%	10.56%
SD	401	16.3%	0.2%	9.28%	9.87%

Table 2. Static function call characteristics.

* See R. Rudell, "Espresso-MV: Algorithms for Multiple-Valued Logic Minimization, Proc. Cust. Int. Circ. Conf., May 1985.

benchmark	total	external	pointer	unsafe	safe
cccp	2569	5.3%	5.4%	6.5%	82.7%
cmp	1001	50.2%	0.0%	0.5%	49.3%
compress	4684	8.1%	0.0%	0.6%	91.3%
eqn	48428	8.2%	0.0%	0.9%	90.9%
espresso	295778	0.1%	9.4%	0.2%	90.3%
grep	17489	1.2%	0.0%	0.1%	98.8%
lex	84648	13.5%	0.0%	0.4%	86.1%
make	48056	9.2%	0.0%	0.2%	90.6%
tar	1442	35.3%	0.0%	12.0%	52.7%
tee	1583	99.1%	0.0%	0.9%	0.0%
wc	21	53.1%	0.0%	46.9%	0.0%
yacc	3935	7.7%	0.0%	0.3%	92.0%
AVG	42470	24.3%	1.2%	7.8%	68.7%
SD	84216	29.9%	3.0%	13.4%	35.6%

Table 3. Dynamic function call behavior.

benchmark	code inc	call dec	IL's per call	CT's per call
cccp	17%	55%	506	95
cmp	3%	49%	265	58
compress	4%	91%	2324	368
eqn	22%	81%	197	58
espresso	24%	70%	616	96
grep	31%	99%	11214	4071
lex	23%	77%	7807	2880
make	34%	59%	388	82
tar	16%	43%	983	127
tee	0%	0%	15	6
wc	0%	0%	18310	5146
yacc	24%	80%	1205	303
AVG	16.5%	58.7%	3653	1108
SD	12.0%	32.1%	5804	1832

Table 4. Inline expansion results.