April 1991

UILU-ENG-91-2220
CRHC-91-13

Center for Reliable and High-Performance Computing

PROFILE-GUIDED AUTOMATIC INLINE EXPANSION FOR C PROGRAMS

Pohua P. Chang
Wen-mei W. Hwu

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY none	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE none	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-91-2220 CRHC-91-13	5. MONITORING ORGANIZATION REPORT NUMBER(S) none
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION NSF, NCR, NASA, AMD
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) NSF: 1800 G. Street, Washington, DC 20552 NCR: Personal Computer Div.-Clemson 1150 Anderson Dr., Liberty, SC 29657
8a. NAME OF FUNDING/SPONSORING ORGANIZATION same as 7a. 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NSF: MIP-8809478 NASA: NAG 1-613
8c. ADDRESS (City, State, and ZIP Code) same as 7b.	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Profile-guided Automatic Inline Expansion for C Programs

12. PERSONAL AUTHOR(S)
Chang, Pohua P., and Hwu, Wen-mei W.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1991 April	32

16. SUPPLEMENTARY NOTATION
none

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	inline expansion, C compiler, Code optimization

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

We describe critical implementation issues that must be addressed to develop a fully automatic inliner. These issues are: integration into a compiler, program representation, hazard prevention, expansion sequence control, and program modification. An automatic inter-file inliner that uses profile information has been implemented and integrated into an optimizing C compiler. The experimental results show that this inliner achieves significant speed-ups for realistic C programs.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

7b. NASA Langle Research center, Hampton, VA 23665
Advanced Micro Devices, 5900 East Ben White Blvd., Austin, TX 78741

Profile-guided Automatic Inline Expansion for C Programs

Pohua P. Chang and Wen-mei W. Hwu

Center for Reliable and High-performance Computing

Coordinated Science Laboratory

University of Illinois, Urbana-Champaign

Abstract

We describe critical implementation issues that must be addressed to develop a fully automatic inliner. These issues are: integration into a compiler, program representation, hazard prevention, expansion sequence control, and program modification. An automatic inter-file inliner that uses profile information has been implemented and integrated into an optimizing C compiler. The experimental results show that this inliner achieves significant speedups for realistic C programs.

Keywords: Inline expansion, C compiler, Code optimization

1 Introduction

Large computing tasks are often divided into many smaller subtasks which can be more easily developed and understood. Function definition and invocation in high level languages provide a natural means to define and coordinate subtasks to perform the original task. Structured pro-

gramming techniques therefore encourage the use of functions. Unfortunately, function invocation disrupts compile-time code optimizations such as register allocation, code compaction, common subexpression elimination, constant propagation, copy propagation, and dead code removal.

Emer and Clark reported, for a composite VAX workload, 4.5% of all dynamic instructions are function calls and returns [Emer 84]. 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. Eickemeyer and Patel reported a dynamic call frequency of one out of every 27 to 130 VAX instructions. Gross, et al., reported a dynamic call frequency of one out of every 25 to 50 MIPS instructions. Berkeley RISC researchers have reported that function call is the most costly source language statement [Patterson 82]. All these previous results argue for an effective approach to reducing function call costs.

Inline function expansion (or simply *inlining*) replaces a function call with the function body. Inline function expansion removes the function call/return costs and provides enlarged and specialized functions to the code optimizers. In a recent study, Allen and Johnson identified inline expansion as an essential part of a vectorizing C compiler [Allen 88]. Scheifler implemented an inliner that takes advantage of profile information in making inlining decisions for the CLU programming language. Experimental results, including function invocation reduction, execution time reduction, and code size expansion, were reported based on four programs written in CLU [Scheifler 77].

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 functions [Richardson 89].

Many optimizing compilers can perform inline expansion. For example, the IBM PL.8 compiler

does inline expansion of all leaf-level functions [Auslander 82]. In the GNU C compiler, the programmers can use the keyword *inline* as a hint to the compiler for inline expanding function calls [Stallman 88]. In the Stanford MIPS C compiler, the compiler examines the code structure (e.g. loops) to choose the function calls for inline expansion [Chow 84]. Parafrase has an inline expander based on program structure analysis to increase the exposed program parallelism [Huson 82]. It should be noted 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.

Davidson and Holler have developed an automatic source-to-source inliner for C [Davidson 88] [Davidson 89]. Because their inliner works on the C source program level, many existing C programs for various computer systems can be optimized by their inliner. The effectiveness of their inliner has been confirmed by strong experimental data collected for several machine architectures. The implementation of their inliner has been described in detail in [Davidson 88] and [Davidson 89].

In the process of developing an optimizing C compiler, we have decided to allocate 6 man-months to construct an automatic inliner. We expect that an inliner can enlarge the scope of code optimization and code scheduling, and eliminate a large percentage of function calls. In this paper, we describe the major implementation issues regarding a fully automatic inliner for C, and our design decisions. We have implemented the inliner and integrated it into our prototype C compiler. The inliner consists of approximately 5200 lines of commented C code, not including the profiler that is used to collect profile data. The inliner is a part of a portable C compiler front-end that has been ported to Sun3, Sun4 and DEC-3100 workstations running UNIX operating systems.

Our implementation is different from other automatic inliners [Scheifler 77] [Davidson 88]. Unlike the CLU language [Scheifler 77], C is a complex programming language, which supports calls

through pointers, variable number of arguments, and a large library of basic functions (e.g., cos) whose source code are not always available. As we will discuss in the next section, there are other types of hazards that must be avoided by an automatic C inliner. Unlike the INLINER program [Davidson 88], our inliner operates on compiler intermediate codes, and the inline decisions are based on profile information to maximize the number of calls eliminated, while maintaining an allowable code expansion ratio.

2 Critical Implementation Issues

The basic idea of inline expansion is simple. Most of the difficulties are due to hazards, missing information, and reducing the compilation time. We have identified the following critical issues of inline expansion:

(1) Where should inline expansion be performed in the compilation process?

(2) What data structure should be employed to represent programs?

(3) How can hazards be avoided?

(4) How should the sequence of inlining be controlled to reduce compilation cost?

(5) What program modifications are made for inlining a function call?

A *static function call site* (or simply *call site*) refers to a function invocation specified by the static program. A *function call* is the activity of invoking a particular function from a particular call site. A *dynamic function call* is an executed function call. If a call site can potentially invoke more than one function, the call site has more than one function call associated with it. This is usually due to the use of the call-through-pointer feature provided in some programming languages. The *caller* of a function call is the function which contains the call site of that function call. The

callee of a function call is the function invoked by the function call.

2.1 Integration into the compilation process

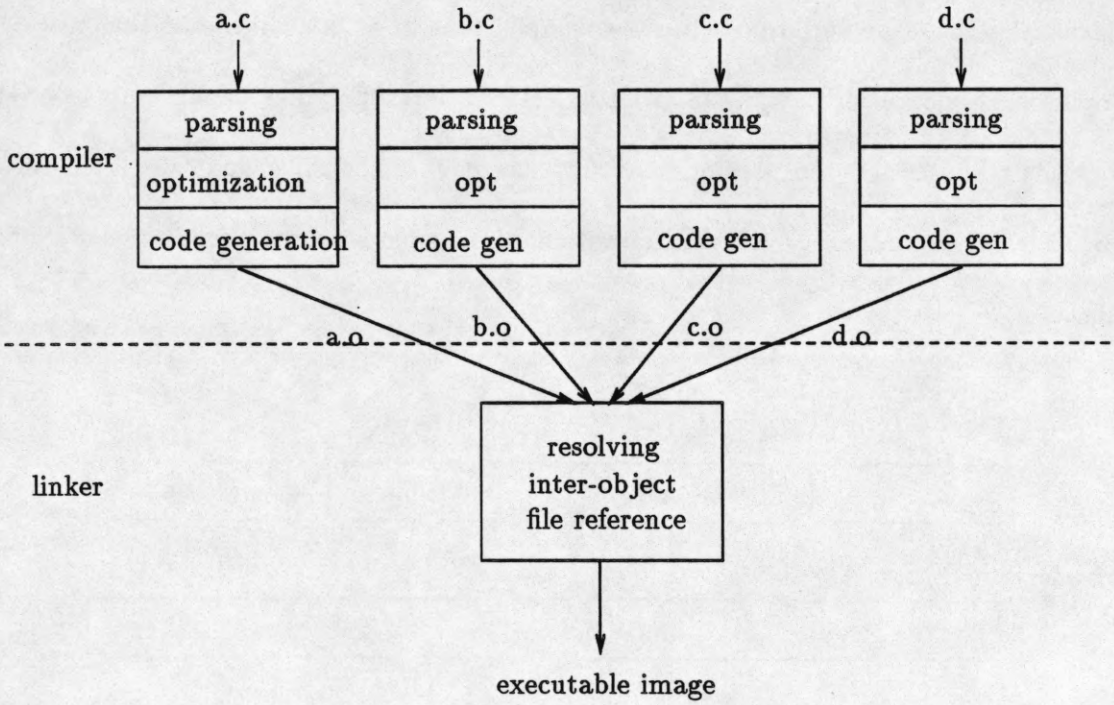

Figure 1: Separate compilation paradigm.

The first issue regarding inline function expansion is where inlining should be performed in the translation process. In most traditional program development environments, the source files of a program are separately compiled into their corresponding object files before being linked into an executable file (see Figure 1). The *compile time* is defined as the period of time when the source files are independently translated into object files. The *link time* is defined as the duration when the object files are combined into an executable file. Most of the optimizations are performed at compile time, whereas only a minimal amount of work to link the object files together is performed at link time. This simple two-stage translation paradigm is frequently referred to as the *separate*

compilation paradigm.

Because the caller and callee functions may reside in different source files, inline function expansion and global optimization in general increase the coupling of the source files involved. Inline function expansion could be performed either at compile time or at link time. In either case, separate compilation is no longer possible in order to perform inter-file inline expansion. The GNU C Compiler has a limited inline expansion feature which requires the caller and callee to be in the same source file for expansion. With this limitation, the simple separate compilation paradigm remains intact.

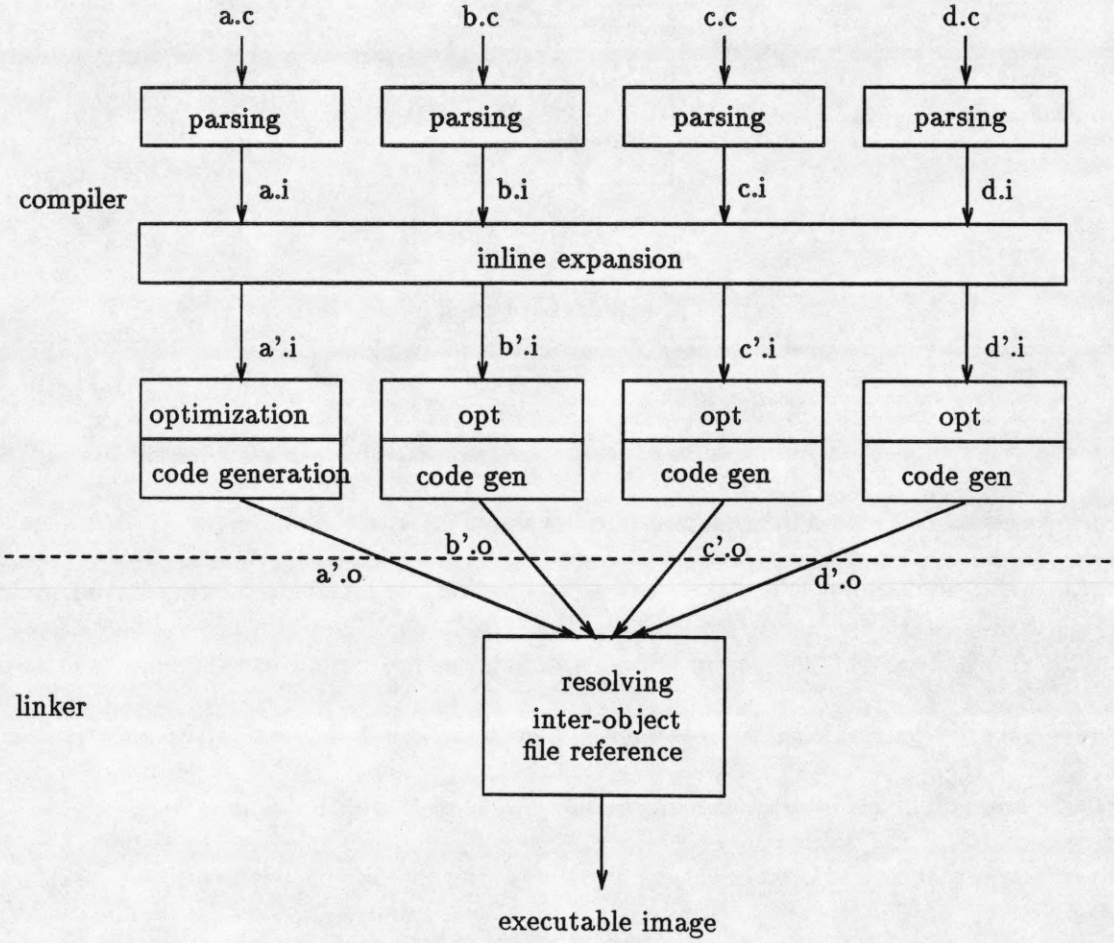

Figure 2: Inlining at compile time.

An extension to the separate compilation paradigm to allow inlining at compile time is illustrated in Figure 2. Performing inline function expansion at compile time provides several advantages. First, inline function expansion enlarges the scope and thus increases the opportunities for global code optimization techniques. Performing inline function expansion at the early stage of the compile time (before the code optimization steps) ensures that these code optimization steps benefit from inlining. Second, functions are often created as generic modules to be invoked for a variety of purposes. Inlining a function call places the body of the corresponding function into a specific invocation, which eliminates the need to cover the service required by the other callers. Therefore, constant propagation, constant folding, and dead code removal can be expected to reduce the code size expansion due to inlining. Third, being applied before system-dependent code generation, inline expansion can be included in a portable front-end.

Performing inline function expansion at compile time requires the callee function source (or intermediate) code to be available when the caller is compiled. Note that the callee functions can reside in different source files than the callers. As a result, the caller and callee source files can no longer be compiled independently. Also, whenever a callee function is modified, both the callee and caller source files must be recompiled. This coupling between the caller and callee source files reduces the advantage of the two-step translation process.

In practice, some library functions are written in assembly languages; they are available only in the form of object files to be integrated with the user object files at link time. These library functions are not available for inline function expansion at compile time. Dynamically linked libraries represent a step further in the direction of separating the library functions from the user programs invoking them. The dynamically linked library functions are not available for inline function expansion at all.

Inline function expansion is performed at compile time in our C Compiler. Performing inline function expansion at compile time is compatible with most of the existing compiler structures.

2.2 Program representation

The second issue regarding inline function expansion is what data structure should be employed to represent the program. In order to support efficient inlining, the data structure should have two characteristics. First, the data structure should conveniently capture the dynamic and static function calling behavior of the represented programs. Second, efficient algorithms should be available to construct and manipulate the data structure during the whole process of inline function expansion. Weighted call graphs, as described below, exhibit both desirable characteristics.

A weighted call graph captures the static and dynamic function call behavior of a program. A weighted call graph (a directed multigraph), $G = (N, E, main)$, 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 number of invocations of the function by all callers. Each arc in E is a static function call in the program and has associated with it a weight, which is the execution count of the call. Finally, *main* is the first function executed in this program. The node weights and arc weights may be determined either by program structure analysis or by profiling.

An example of a weighted call graph is shown in Figure 3. There are eight functions in this example: main, A, B, C, D, E, F, and G. The weights of these functions are indicated beside the names of the functions. For example the weights of functions A and E are 5 and 7 respectively. Each arc in the call graph represents a static function call whose weight gives its expected dynamic execution count in a run. For example, the main function calls G from two different static locations;

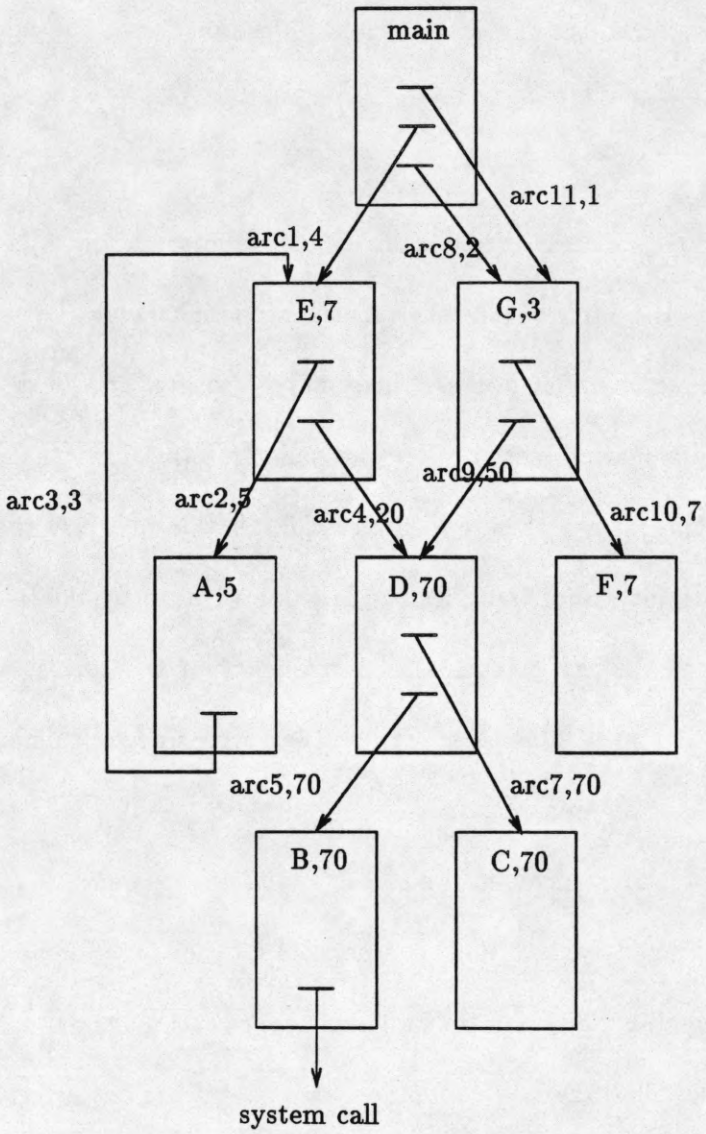

Figure 3: A weighted call graph.

one is expected to execute one time and the other is expected to execute two times in a typical run.

Each node in the weighted call graph contains three pieces of information: 1) the body of the function, 2) the node weight, and 3) a set of outgoing arcs to the callees. The body of a function gives all the program declarations and statements of the function. The node weight gives the expected invocation count of the function. The outgoing arcs identify all static function calls in the present function.

Each arc in the weighted call graph contains five pieces of information: 1) a unique identifier, 2) the name of the caller, 3) the name of the callee, 4) the arc weight, and 5) a status. It is necessary to assign each arc a unique identifier because there may be several arcs between the same pair of caller and callee; the combination of the caller and callee information can not uniquely identify a static function call. The caller attribute identifies the function in which the corresponding call site is located. The callee attribute identifies the function invoked by the function call. The arc weight attribute indicates the expected execution frequency of the corresponding function call. The *status* attribute indicates whether this arc is to be considered for inline expansion, rejected for inline expansion, or already inline expanded.

A weighted call graph is constructed in two steps. The first step generates all the nodes and arcs according to static program analysis. A node is generated for each function and an arc is generated for each call site. The function body and the outgoing arcs of each node are generated at this step. The unique identifier, the caller, the callee, and the status of each arc are also generated at this step. The second step is to fill in the weights for the nodes and the arcs. A system-independent profiler has been integrated into our compiler. The profiler accumulates the average run-time statistics over many runs of a program. From the profile information, our C compiler can determine the execution counts of all functions and the invocation counts of all call sites.

Inlining a function call is equivalent to duplicating the callee node, absorbing the duplicated node into the caller node, eliminating the arc from the caller to the callee, and possibly creating some new arcs in the weighted call graph. For example, inlining B into D in Figure 3 involves duplicating B, absorbing the duplicated B into D, eliminating the arc going from D to B, and creating a new system call arc. The resulting call graph is shown in Figure 4.

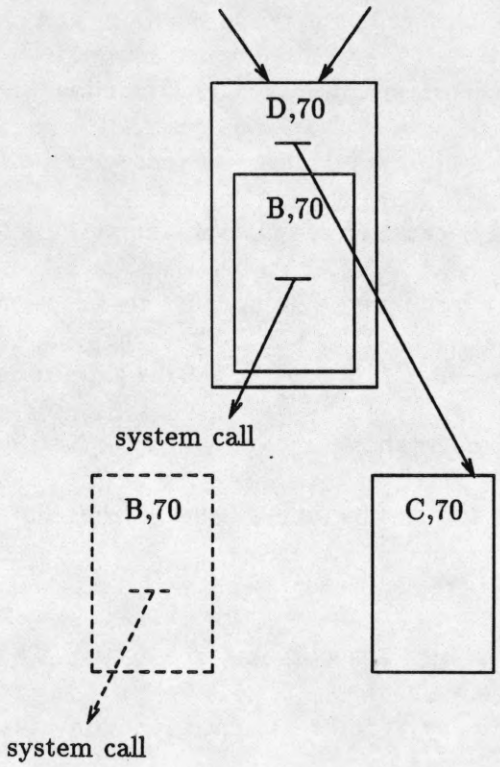

Figure 4: An inlining example.

Detecting recursion is equivalent to detecting cycles in the weighted call graph. For example, a recursion involving functions A and E in Figure 3 can be identified by detecting the cycle involving nodes A and E in the weighted call graph. Identifying functions which can never be reached during execution is equivalent to finding unreachable nodes from the *main* node. For example, Function B is no longer reachable from the main function after it is inline expanded into Function D (see

Figure 4). This can be determined by identifying all the unreachable nodes from the main node in the weighted call graph. Efficient graph algorithms for these operations are widely available [Tarjan 83].

When the inline expander fails to positively determine the internal function calling characteristics of some functions, there is missing information in the call graph construction. The two major causes of the missing information are calling external functions and calling through pointers. Calling external functions occurs when a program invokes a function whose source file is unavailable to the inline expander. Examples include privileged system service functions and library functions distributed without source files. Because these functions can perform function calls themselves, the call graphs thus constructed are incomplete. Practically, because some privileged system services and library functions can invoke user functions, a call to an external function may have to be assumed to indirectly reach all nodes whose function addresses have been used in the computation in order to detect all recursions and all functions reachable from *main*.

A special node $EXTERN$ is created to represent all the external functions. A function which calls external functions requires only one outgoing arc to the $EXTERN$ node. In turn, the $EXTERN$ node has many outgoing arcs, one to each function whose address has been used in the computation to reflect the fact that these external functions can potentially invoke every such function in the call graph.

Calling through pointers is a language feature which allows the callee of a function call to be determined at the run time. Theoretically, the set of potential callees for a call through pointer can be identified using program analysis. A special node PTR is used to represent all the functions which may be called through pointers. Calls through pointers are not considered for inlining in our implementation. Rather than assigning a node to represent the potential callee of each call

through pointer, PTR is shared among all calls through pointers. In fact, PTR is assumed to reach all functions whose addresses have been used in the computation. This again ensures that all the potential recursions and all the functions reachable from the *main* can be safely detected.

2.3 Hazard detection and prevention

The third issue regarding inline function expansion is how the hazardous function calls should be excluded from inlining. Four hazards have been identified in inline expansion: unavailable callee function bodies, multiple potential callees for a call site, activation stack explosion, and variable number of arguments. A practical inline expander has to address all these hazards. All the hazardous function calls are excluded from the weighted call graph and are not considered for inlining by the sequence controller.

The bodies of external functions are unavailable to the compiler. External functions include privileged system calls and library functions that are written in an assembly language. In the case of privileged system calls, the function body is usually not available regardless of whether the inline expansion is performed at compile time or link time. In fact, inlining privileged system calls is usually not desirable due to security reasons. Therefore, privileged system calls should be considered as not inline expandable.

Multiple potential callees for a call site occur due to calling through pointers. Because the callees of calls through pointers depend on the run-time data, there is, in general, more than one potential callee for each call site. Note that each inline expansion is equivalent to replacing a call site with a callee function body. If there is more than one potential callee, replacing the call site with only one of the potential callee function bodies eliminates all the calls to the other callees by mistake. Therefore, function calls originating from a call site with multiple potential callees should

not be considered for inline expansion. If a call through pointer is executed with extremely high frequency, one can insert *I* statements to selectively inline the most frequent callees. This may be useful for programs with a lot of dispatching during run time, such as logic simulators.

Parameter passing, register saving, local variable declarations, and returned value passing associated with a function can all contribute to the activation stack usage. A summarized activation stack usage can be computed for each function. A recursion may cause activation stack overflow if a call site with large activation record is inlined into one of the functions in the recursion. For example, a recursive function m(x) and another function n(x) are defined as follows.

```
m(x) { if (x > 0) return(m(x-1)); else return(n(x)); }
n(x) { int y[100000]; ..... }
```

For the above example, two activation stacks are shown in Figure 5, one with inline expansion and one without. Note that inlining n(x) into the recursion significantly increases the activation stack usage. If m(x) tends to be called with a large x value, expanding n(x) will cause an explosion of activation stack usage. Programs which run correctly without inline expansion may not run after inline expansion. To prevent activation stack explosion, a limit on the control stack usage can be imposed for inline expanding a call into a recursion.

In C, a function can expect a variable number of parameters. Moreover, the parameter data types may vary from call to call (e.g., printf). Because calls to this type of functions are rare in practice, these calls are prevented from being inlined. In our compiler, this is done by writing the names of this type of functions in a file, and specifying this file as a compilation option.

The calls to external functions and the calls through pointers are excluded from inline expansion. Because our compiler performs inline expansion at compile time, any function calls whose callee source code (or intermediate code) is unavailable are excluded from inlining. A parameter to the

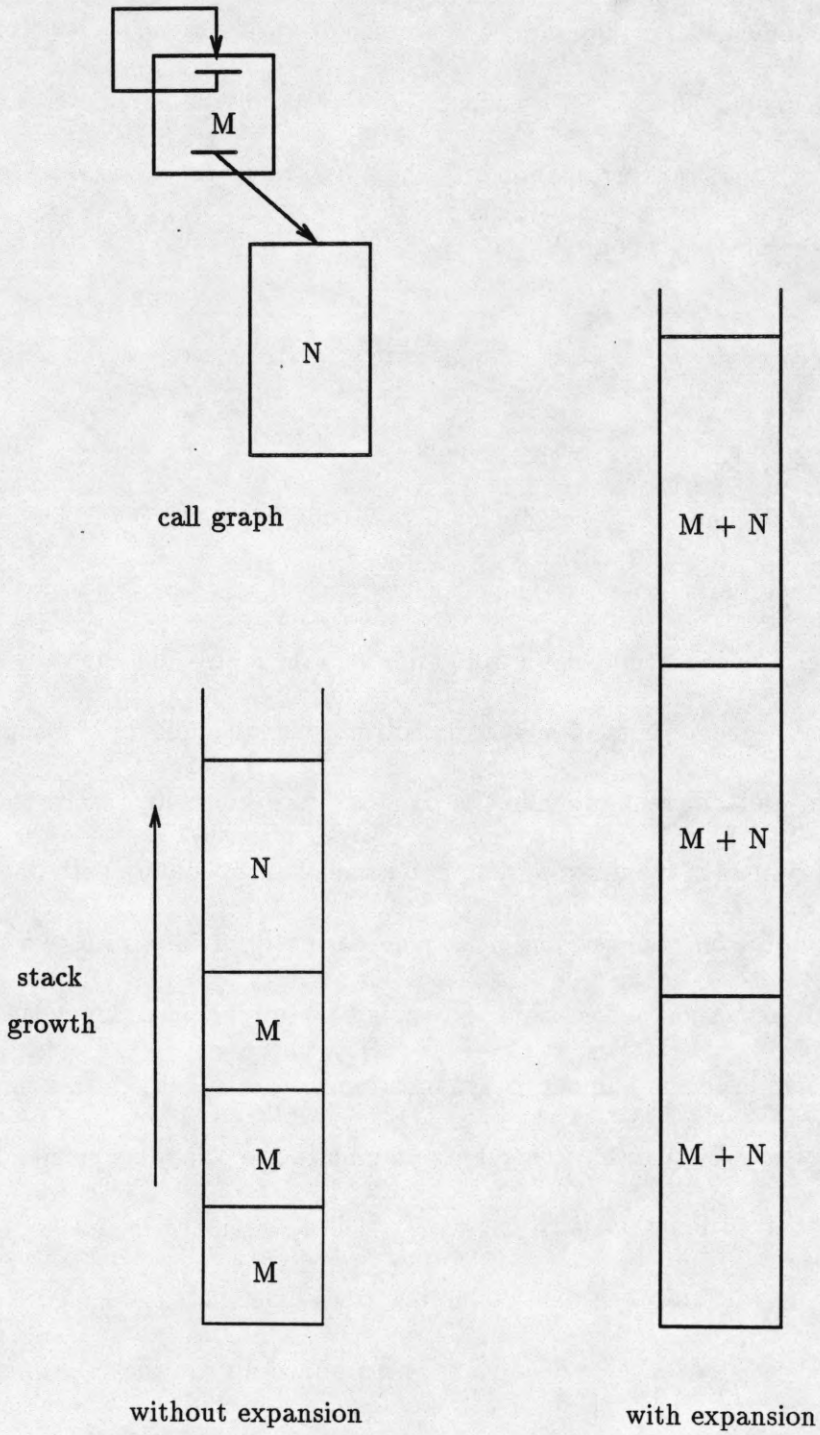

Figure 5: Activation stack explosion.

compiler specifies the limit on the activation stack usage of a function to be inlined into a (potential) recursion. Any functions which require more activation stack usage are excluded from being inlined into a (potential) recursion. Functions that expect variable number of parameters are also excluded from being inlined. All the arcs corresponding to these hazardous function calls are excluded from the consideration of inline expansion.

2.4 Sequence control

The fourth issue regarding inline function expansion is how the sequence of inlining should be controlled to minimize unnecessary computation and code expansion. In this step, we do not consider the hazardous function calls. The sequence control in inline expansion determines the order in which the arcs in the weighted control graph, i.e., the static function calls in the program, are inlined. Different sequence control policies result in different numbers of expansions, different code size expansion, and different reduction in dynamic function calls. All these considerations affect the cost-effectiveness of inline expansion, and some of them conflict with one another.

The sequence control of inline expansion can be naturally divided into two steps: selecting the function calls for expansion and actually expanding these functions. The goal of selecting the function calls is to minimize the number of dynamic function calls subject to a limit on code size increase. The goal of actual expansion control is to minimize the computation cost incurred by the expansion of these selected function calls. Both steps will be discussed in this section.

In this section, we will limit the discussion to a class of inline expansion with the following restriction. If a function F has a callee L and L is to be inlined into F, then all functions absorbing F will also absorb L. Note that this restriction can cause some extra code expansion, as illustrated in the following example. Function F calls L (100 times) and is called by A (990 times) and B (10

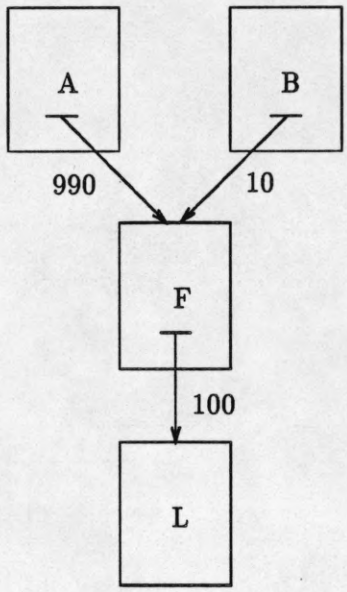

Figure 6: An example of restricted inlining.

times) (see Figure 6). In this call graph, there is not enough information to separate the number of times F calls L when it is being invoked by A and by B. Assume F is to be absorbed into both A and B. If F calls L 99 times when it is invoked by A and 1 time when by B, then L should be absorbed into A but not B (see Figure 7). With our restriction, however, L will be absorbed into both A and B (see Figure 7). Obviously absorbing L into B is not cost-effective in this case.

The problem is, however, that, there is not enough information in the call graph to attribute the F→L weight to A and B separately. Therefore, the decision to absorb L only into A would be based on uncertain information. Also, to accurately break down the weights, one needs to duplicate each arc as many times as the number of possible paths via which the arc can be reached from the main function. This will cause an exponential explosion of the number of arcs in the weighted call graph.

Because all the hazards due to recursion have been handled by the technique described in Section 2.3, the call graph can be simplified by breaking all the cycles. The cycles in the call graph

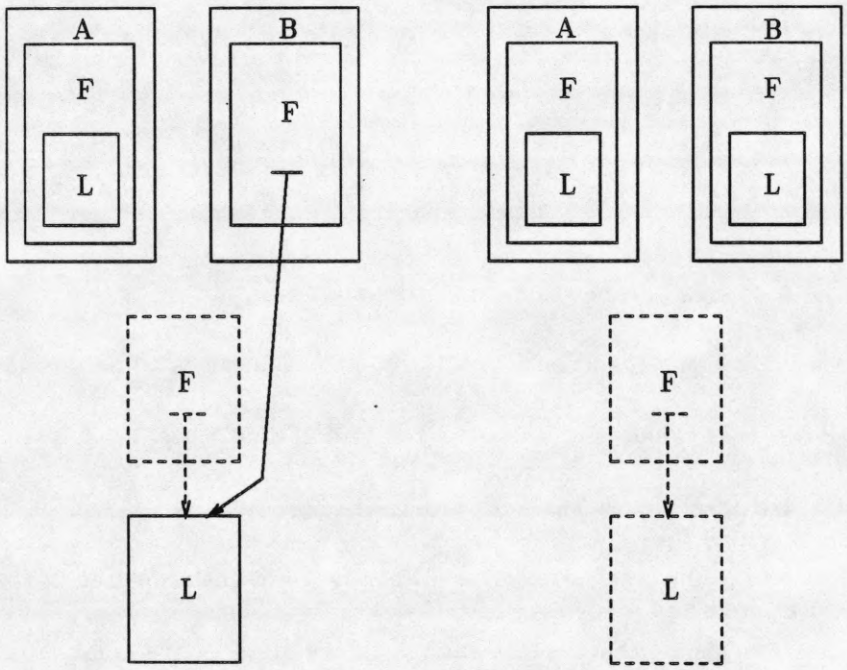

Figure 7: Lost opportunity.

can be broken by excluding the least important arc from each cycle in the call graph. If the least important arc is excluded from inlining to break a cycle involving N functions, one can lose the opportunity to eliminate up to $1/N$ of the dynamic calls involved in the recursion. This is usually acceptable for N greater than 1.

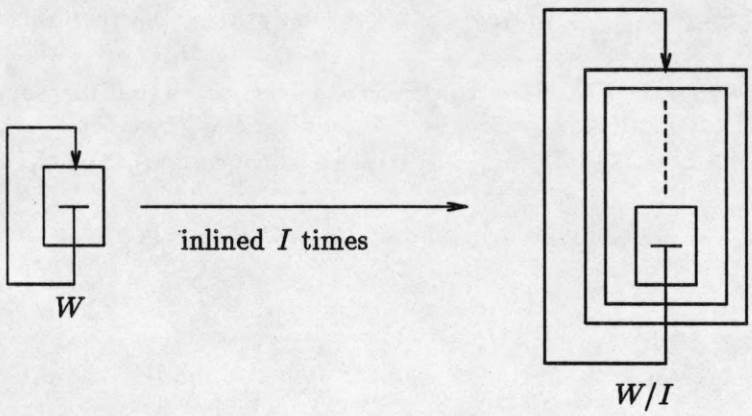

Figure 8: Handling single-function recursions.

If N is equal to 1, breaking the cycle will eliminate all the opportunity of reducing the dynamic calls in the recursion. If the recursion happens to be the dominating cause of dynamic function calls in the entire program, one would lose most of the call reduction opportunity by breaking the cycle. There is, however, a simple solution to this problem (see Figure 8). One can inline the recursive function call I times before breaking the cycle. In this case, one loses only $1/I$ of the call reduction opportunity by breaking the cycle.

The weighted call graph becomes a directed acyclic graph after all the cycles are broken. All the following discussions assume this property.

It is desirable to expand as many frequently executed function calls (heavily weighted arcs in the call graph) as possible. However, unlimited inline expansion causes code size expansion. In order to expand a function call, the body of the callee must be duplicated and the new copy of the

callee must be absorbed by the caller. Obviously, this code duplication process in general increases program code size. Therefore, it is necessary to set an upper bound on the code size expansion. This limit may be specified as a fixed number and/or as a function of the original program size. The problem with using a fixed limit is that the size of the programs handled varies so much that it is very difficult to find a single limit to suit all the programs. Setting the upper limit as a function of the original program size tends to perform better for virtual memory and favor large programs. 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.

Code size expansion increases the memory required to accommodate the program and reduces instruction memory hierarchy performance. Precise costs can not be obtained during inline expansion because the code size depends on the optimizations to be performed after inline expansion. The combination of copy propagation, constant propagation, and unreachable code removal will reduce the increase in code size. A rough estimate of the code size increase can be derived from the intermediate code size of each function. Because the sizes of the functions change during inline expansion, it is important to keep track of the up-to-date size of each function.

Accurate benefits of inline expansion are equally difficult to obtain during inline expansion. Inline expansion improves the effectiveness of register allocation and algebraic optimizations, which reduces the computation steps and the memory accesses required to execute the program. Because these optimizations are performed after inline expansion, the precise improvement of their effectiveness due to inline expansion can not be known during inline expansion. Therefore, the benefit of inline expansion will be judged only by the reduction in dynamic function calls, which in turn reduces execution time of the program for each computer architecture. Using call frequency

reduction rather than execution time reduction allows the inline expander to be independent of architectures.

The problem of selecting functions for inline expansion can be formulated as an optimization problem that attempts to minimize dynamic calls given a limited code expansion allowance. In terms of call graphs, the problem can be formulated as collecting a set of arcs whose total weight is maximized while the code expansion limit is satisfied. It appears that the problem is equivalent to a knapsack problem defined as follows: There is a pile of valuable items each of which has a value and a weight. One is given a knapsack which can only hold up to a certain weight. The problem is to select a set of the items whose total weight fits in the knapsack and the total value is maximized. The knapsack problem has been shown to be NP-complete [Garey 79]. However, this straight forward formulation is unfortunately incorrect for inlining. The code size of each function changes during the inlining process. The code size increase due to inlining each function call depends on the decision made about each function call. The decision made about each function call, in turn, depends on the code size increase. This dilemma is illustrated in Figure 9.

If L is to be inlined into F, the code expansion due to inlining F into A is the total size of F and L. Otherwise, the code expansion is just the size of F. The problem is that the code increase and the expansion decision depend on each other. Therefore, inline expansion sequencing is a even more difficult than the knapsack problem. Nevertheless, we will show that a selection algorithm based on the call reduction achieves good results in practice.

The arcs in the weighted call graph are marked with the decision made on them. These arcs are then inlined in an order which minimizes the expansion steps and source file accesses incurred.

Different inline expansion sequences can be used to expand the same set of selected functions. For example, in Figure 10, Function D is invoked by both E and G. Assume that the selection

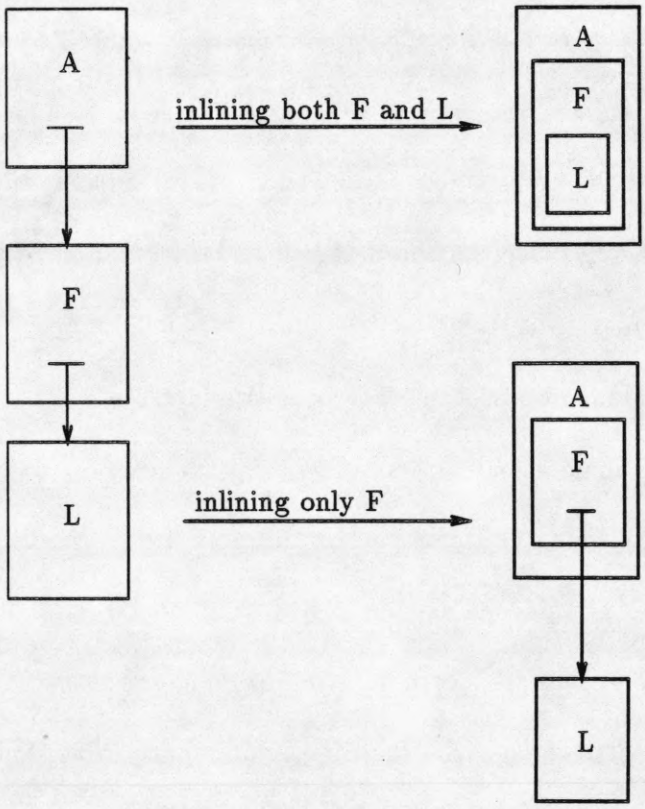

Figure 9: Inter-dependence between code size increase and sequencing.

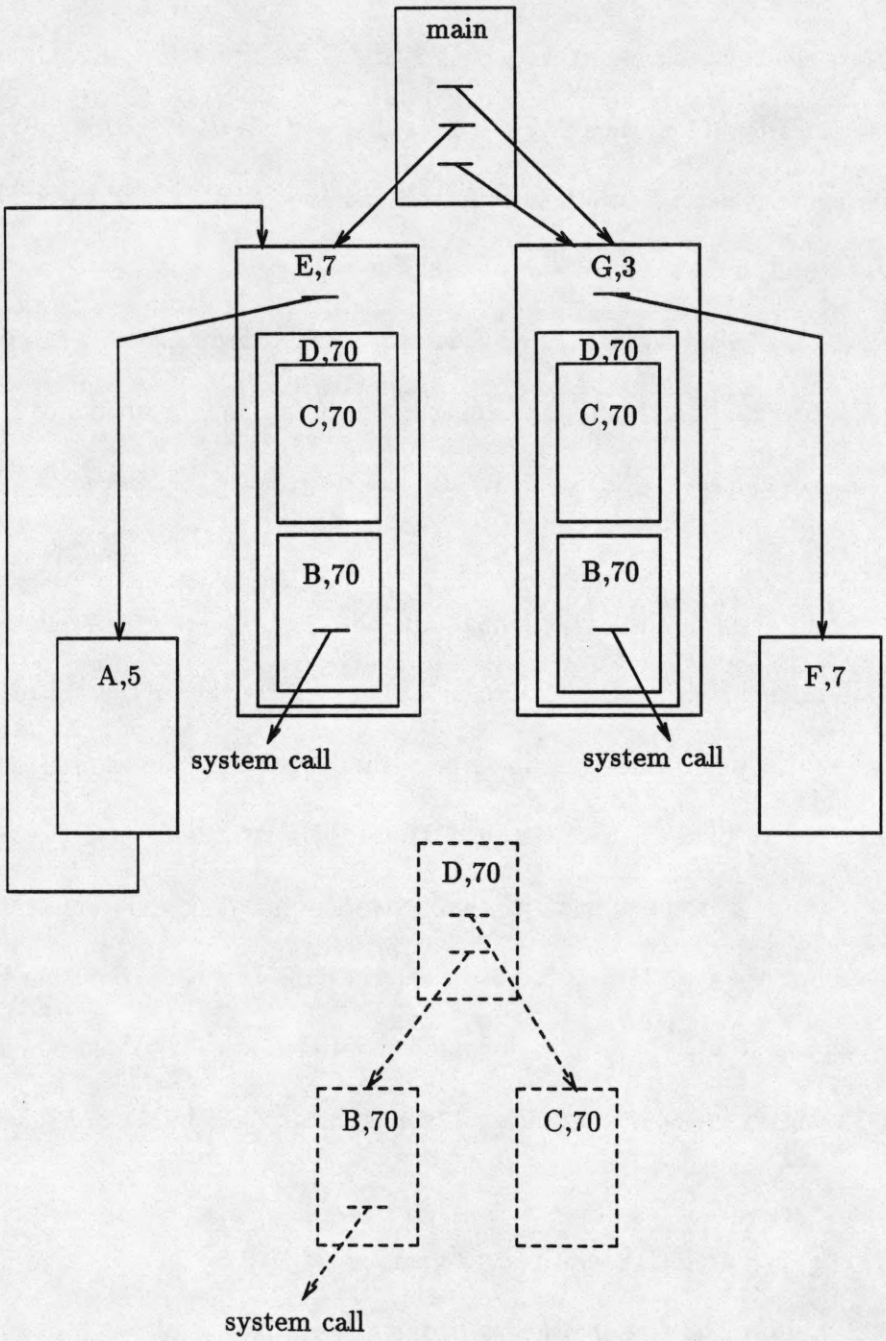

Figure 10: Inlining a function before absorbing its callees.

step decides to absorb D, B, and C into both E and G. There are at least two sequences which can achieve the same goal. One sequence is illustrated in Figure 10, where E→D and G→D are eliminated first. Note that by absorbing D into both E and G (and therefore eliminating E→D and G→D in two expansion steps), four new arcs are created: E→B, E→C, G→B, and G→C. It takes four more steps to further absorb B and C into both E and G to eliminate all these four new arcs. Therefore, it takes a total of 6 expansion steps to achieve the original goal.

A second sequence is illustrated in Figure 11, where B and C are first absorbed into D, eliminating D→B and D→C. Function D, after absorbing B and C, is than absorbed into E and G. This further eliminates E→B and E→C. Note that it only takes a total of 4 expansion steps to achieve the original goal.

The general observation is that if a function is to be absorbed by more than one caller, inlining this function into its caller before absorbing its callees can increase the total steps of expansion.

For the class of inlining algorithms considered in this paper, the rule for minimizing the expansion steps can be stated as follows: If a function F is absorbed into more than one caller, all the callees to be inlined into F must be already inlined. It is clear that any violation against this rule will increase the number of expansions. It is also clear that an algorithm conforming to this rule will perform N expansion steps, where N is the number of function calls to be inlined. Therefore, an algorithm conforming to the rule is an optimal one as far as the number of expansion steps is concerned.

In a directed acyclic call graph, the optimal rule can be realized by an algorithm manipulating a queue of terminal nodes. The terminal nodes in the call graph are inlined into their callers if desired and eliminated from the call graph. This produces a new group of terminal nodes which are inserted into the queue. The algorithm terminates when all the nodes are eliminated from the

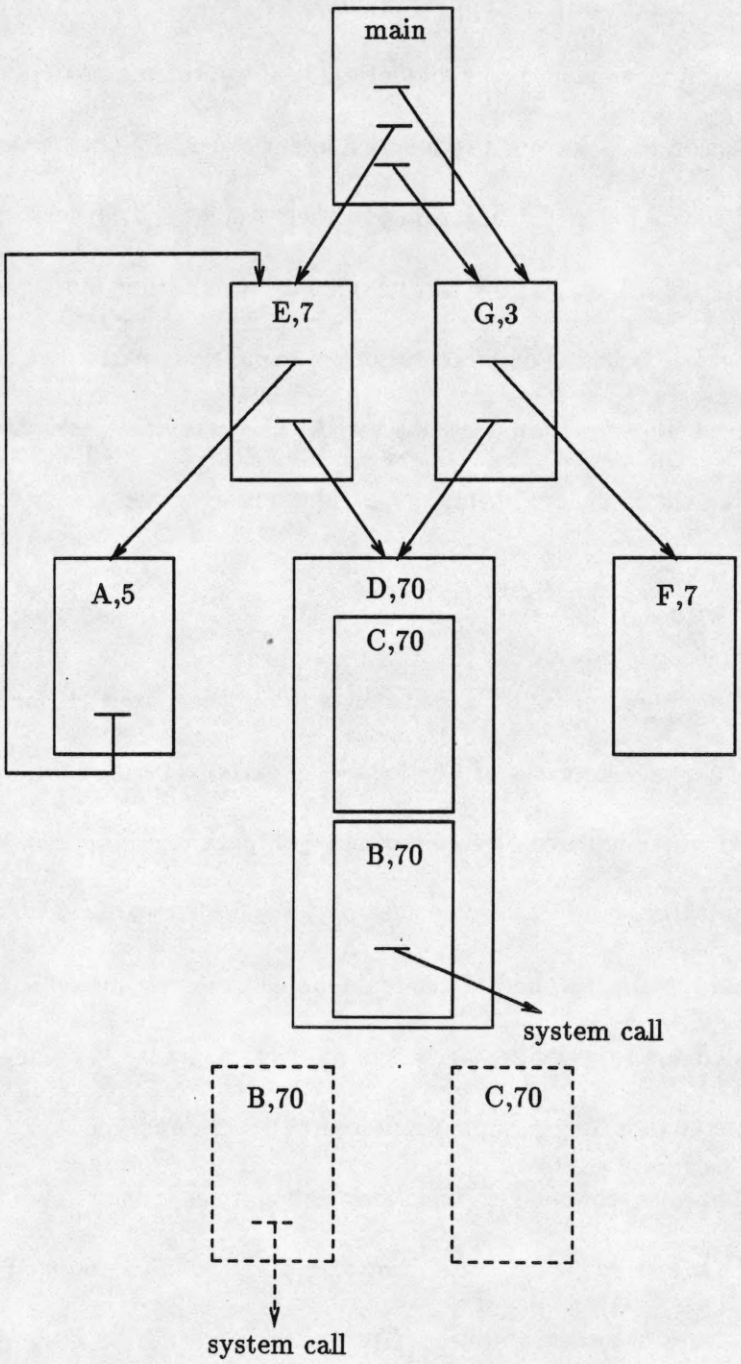

Figure 11: Inlining a function after absorbing its callees.

call graph. The complexity of this algorithm is $O(N)$, where N is the number of function calls in the program (arcs in the call graph) eligible for inlining.

We implemented a simpler sequence control method that approximates the optimal queue-based algorithm. Inline expansion is constrained to follow a linear order. The functions (nodes in the call graph) are first sorted into a linear list according to their weights. The most frequently executed function leads the linear list. A function X can be inlined into another function Y if and only if X appears before Y in the linear list. Therefore, all inline expansions pertaining to function X must already have been done before function Y is processed. The rationale is that functions which are executed frequently are usually called by functions which are executed less frequently.

2.5 Program modifications

The fifth issue regarding function inline expansion is what the essential operations for inlining a function call are. This task consists of the following parts: 1) callee duplication, 2) variable renaming, 3) parameter handling, and 4) elimination of unreachable functions.

To avoid conflicts with the caller's local variables, the callee's local variables must be renamed before inserting the code into the caller. This could be achieved by introducing a new scope for these local variables. This is especially easy in the modern structure languages such as C where provisions have been made to allow multiple scopes within each function.

The callee's formal parameters must also be renamed before code insertion. This again could be achieved by introducing a new scope for these formal parameters. The renamed formal parameters can then receive the actual parameter values. The return value has to be buffered by new local temporary variables so that it can be used by the caller.

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 serve as an exception handler, or be activated by some external functions. In the C language, this can be detected by identifying all functions whose addresses are used in computations.

3 Experiments

local	*global*
constant propagation	constant propagation
copy propagation	copy propagation
common subexpression elimination	common subexpression elimination
redundant load elimination	redundant load elimination
redundant store elimination	redundant store elimination
constant folding	loop invariant code removal
strength reduction	loop induction strength reduction
constant combining	loop induction elimination
operation folding	global variable migration
dead code removal	dead code removal
code reordering	loop unrolling

Table 1: Code optimizations.

Table 1 shows the set of classic local and global code optimizations that we have implemented in our prototype C compiler. These code optimizations are common in commercial C compilers.

Table 2 shows a set of eight C application programs that we have chosen as benchmarks. The *size* column indicates the sizes of the benchmark programs in terms of number of lines of C code. The *description* column briefly describes each benchmark program.

Table 3 describes the input data that we have used for profiling. The *runs* column lists the number of inputs for each benchmark program. The *description* column briefly describes the nature

name	*size*	*description*
cccp	4787	GNU C preprocessor
compress	1514	compress files
eqn	2569	typeset mathematical formulas for troff
espresso	6722	boolean minimization
lex	3316	lexical analysis program generator
tbl	2817	format tables for troff
xlisp	7747	lisp interpreter
yacc	2303	parsing program generator

Table 2: Benchmarks.

name	*runs*	*description*
cccp	20	C source files (100-5000 lines)
compress	20	C source files (100-5000 lines)
eqn	20	ditroff files (100-4000 lines)
espresso	20	boolean minimizations (original espresso benchmarks)
lex	5	lexers for C, Lisp, Pascal, awk, and pic
tbl	20	ditroff files (100-4000 lines)
xlisp	5	gabriel benchmarks
yacc	10	grammars for C, Pascal, pic, eqn, awk, etc.

Table 3: Characteristics of profile input data.

of these input data. Executing each benchmark program with an input produces a profile data file. For each benchmark program, its profile data files are summarized into one profile data file, which is used to guide the automatic inline expander.

name	*external*	*pointer*	*intra-file*	*inter-file*	*inlined*
cccp	143	1	191	4	23
compress	104	0	27	0	1
eqn	192	0	81	144	17
espresso	289	11	167	982	19
lex	203	0	110	234	6
tbl	310	0	91	364	46
xlisp	91	4	331	834	28
yacc	218	0	118	81	14

Table 4: Static characteristics of function calls.

name	*external*	*pointer*	*intra-file*	*inter-file*	*inlined*
cccp	1015	140	1414	3	1183
compress	25	0	4283	0	4276
eqn	5010	0	6959	33534	37440
espresso	728	60965	55696	925710	689454
lex	13375	0	63240	4675	56991
tbl	12625	0	9616	37809	35504
xlisp	4486885	479473	10308201	8453735	14861487
yacc	31751	0	34146	3323	33417

Table 5: Dynamic characteristics of function calls.

Table 4 describes the static (compile-time) characteristics of function calls.[1] The *external* column shows the numbers of static call sites that call functions whose source codes are not available to the compiler. The *pointer* column shows the number of static call sites that call through pointers. The *intra-file* column shows the number of static call sites that call functions in the same source file. The *inter-file* column shows the number of static call sites that call functions in a different

[1] We report call sites that are visible to the compiler.

source file. The *inlined* column shows the number of static call sites that are inlined expanded. Table 5 describes the dynamic (execution-time) characteristics of function calls.

Note that several benchmark programs have large numbers of calls to *external* functions, such as cccp, xlisp, and yacc. Currently, we do not have access to the source code of the C library functions. Including these C library functions in inline expansion will increase the numbers in the *intra-file* and *inter-file* columns. Our inliner can inline call sites that are shown in the *inter-file* and *intra-file* columns. Tables 4 and 5 show that inlining a small percentage of static call sites removes a large percentage of dynamic calls. This shows that profile-guided inline expansion is highly effective.

name	*global*	*global+inline*	*ratio*
cccp	172564	215420	1.25
compress	72300	73228	1.00
eqn	130376	157528	1.21
espresso	311544	338508	1.09
lex	156148	165468	1.06
tbl	181064	214036	1.18
xlisp	267268	354092	1.32
yacc	141268	164584	1.17

Table 6: Code expansion (DEC-3100).

Table 6 indicates the code expansion ratios of the benchmark programs. The *global* column shows the program sizes in bytes before inline expansion. The *global+inline* column shows the program sizes in bytes after inline expansion. The *ratio* column shows the code expansion ratios. The average code expansion ratio for the benchmark programs is about 1.16.

Table 7 shows the speedups of the benchmark programs. The *global+inline* column is computed by dividing the execution time of non-inlined code by the execution time of inlined code. The average speedup for the benchmark programs is about 1.11.

name	*global*	*global+inline*
cccp	1.00	1.06
compress	1.00	1.05
eqn	1.00	1.12
espresso	1.00	1.07
lex	1.00	1.02
tbl	1.00	1.04
xlisp	1.00	1.46
yacc	1.00	1.03
average	1.00	1.11

Table 7: Speedups (DEC-3100).

4 Conclusion

An automatic inliner has been implemented and integrated into an optimizing C compiler. In the process of designing and implementing this inliner, we have identified several critical implementation issues: integration into a compiler, program representation, hazard prevention, expansion sequence control, and program modification. In this paper, we have described our implementation decisions. We have shown that this inliner eliminates a large percentage of function calls and achieves significant speedup for a set of realistic C programs.

References

[Allen 88] R. Allen and S. Johnson, "Compiling C for Vectorization, Parallelism, and Inline Expansion", Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta, Georgia, June 1988.

[Auslander 82] M. Auslander and M. Hopkins, "An Overview of the PL.8 Compiler", Proceedings of the SIGPLAN Symposium on Compiler Construction, June 1982.

[Chow 84] F. Chow and J. Hennessy, "Register Allocation by Priority-based Coloring", Proceedings of the ACM SIGPLAN Symposium on Compiler Constructions, June 1984.

[Davidson 88] 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.

[Davidson 89] J. W. Davidson and A. M. Holler, "A Model of Subprogram Inlining", Computer Science Technical Report TR-89-04, Department of Computer Science, University of Virginia, July 1989.

[Emer 84] 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.

[Garey 79] M. R. Garey and D. S. Johnson, Computers and Intractability, A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, New York, 1979.

[Huson 82] C. A. Huson, "An In-line Subroutine Expander for Parafrase", University of Illinois, Champaign-Urbana, 1982.

[Hwu 89] W. W. Hwu and P. P. Chang, "Inline Function Expansion for Compiling Realistic C Programs", Proceedings, ACM SIGPLAN'89 Conference on Programming Language Design and Implementation, Portland, Oregon, June 1989.

[Patterson 82] D. A. Patterson and C. H. Sequin, "A VLSI RISC", IEEE Computer, pp.8-21, September 1982.

[Richardson 89] S. Richardson and M. Ganapathi, "Code Optimization Across Procedures", IEEE Computer, February 1989.

[Scheifler 77] R. W. Scheifler, "An Analysis of Inline Substitution for a Structured Programming Language", Communications of the ACM, vol.20, no.9, September 1977.

[Stallman 88] R. M. Stallman, Internals of GNU CC, 1988.

[Tarjan 83] R. E. Tarjan, Data Structures and Network Algorithms, SIAM, Philadelphia, PA., 1983.

This research has been supported by the National Science Foundation (NSF) under Grant MIP-8809478, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS).