SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 22(5), 349–369 (MAY 1992)

Profile-guided Automatic Inline Expansion for C Programs

POHUA P. CHANG, SCOTT A. MAHLKE, WILLIAM Y. CHEN
AND WEN-MEI W. HWU
Center for Reliable and High-performance Computing, Coordinated Science Laboratory, University of Illinois, 1101 W. Springfield Ave, Urbana-Champaign, IL 61801, U.S.A.

SUMMARY

This paper describes 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 production C programs.

KEY WORDS Mine expansion C compiler Code optimization Profile information

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 co-ordinate subtasks to perform the original task. Structured programming 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 per cent of all dynamic instructions are function 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. Eickemeyer and Patel reported a dynamic call frequency of one out of every 27 to 130 VAX instructions. [2] Gross *et al.*, cited a dynamic call frequency of one out of every 25 to 50 MIPS instructions. [3] Patterson and Sequin reported that function call is the most costly source language statement. [4] 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. [5] Scheifler implemented an inliner that takes advantage of profile infor-

0038–0644/92/050349–21$10.50

Received 15 April 1991
Revised 21 November 1991

mation 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. [6]

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. [7]

Many optimizing compilers can perform inline expansion. For example, the IBM PL.8 compiler does inline expansion of low-level intrinsic functions. [8] In the GNU C compiler, the programmers can use the keyword inline as a hint to the compiler for inline expanding function calls. [9] In the Stanford MIPS C compiler, the compiler examines the code structure (e.g. loops) to choose the function calls for inline expansion. [10] Parafrase has an inline expander based on program structure analysis to increase the exposed program parallelism. [11] It should be noted that the careful use of the macro expansion and language preprocessing utilities has the same effect as inline expansion, where inline expansion decisions are made entirely by the programmers.

Davidson and Holler have developed an automatic source-to-source inliner for C. [12,13] 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.

In the process of developing an optimizing C compiler, we decided to allocate about six man-months to construct a profile-guided automatic inliner. [14] 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-31OO workstations running UNIX operating systems.

CRITICAL IMPLEMENTATION ISSUES

The basic idea of inlining 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.

Integration into the compilation process

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.*

A major advantage of the separate compilation paradigm is that when one of the source files is modified, only the corresponding object file needs to be regenerated before linking the object files into the new executable file, leaving all the other object files intact. Because most of the translation work is performed at compile time, separate compilation greatly reduces the cost of program recompilation when only a small number of source files are modified. Therefore, the two-stage separate compilation paradigm is the most attractive for program development environments where programs are frequently recompiled and usually a small number of source

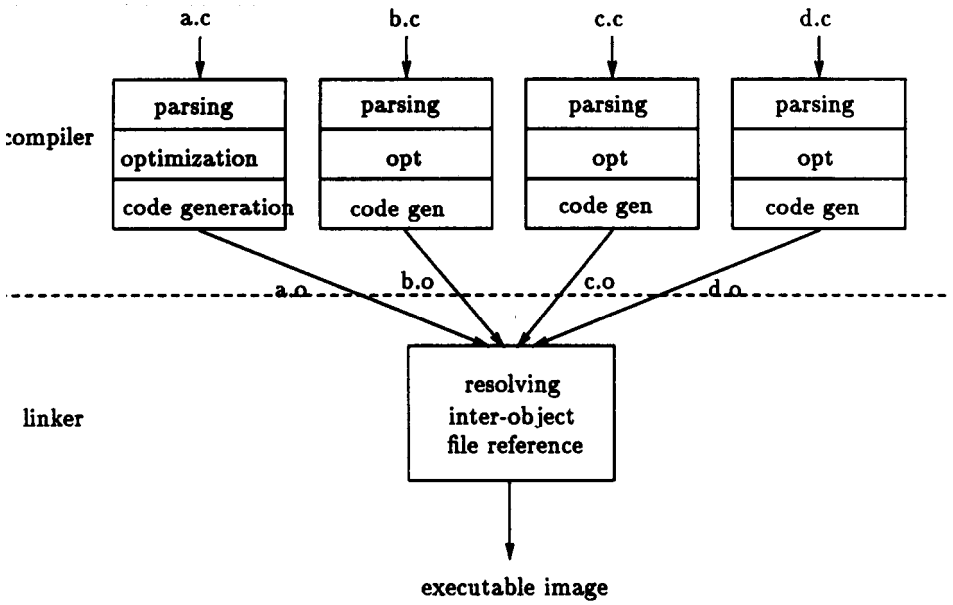

Figure 1. Separate compilation paradigm

files are modified between each recompilation. There are programming tools, such as the UNIX make program, to exploit this advantage.

Our extension to the separate compilation paradigm to allow inlining at compile time is illustrated in Figure 2. Performing inline function expansion before the code optimization steps ensures that these code optimization steps benefit from inlining. For example, 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, optimization such as constant propagation, constant folding, and dead code removal can be expected to be more effective with inlining. *

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

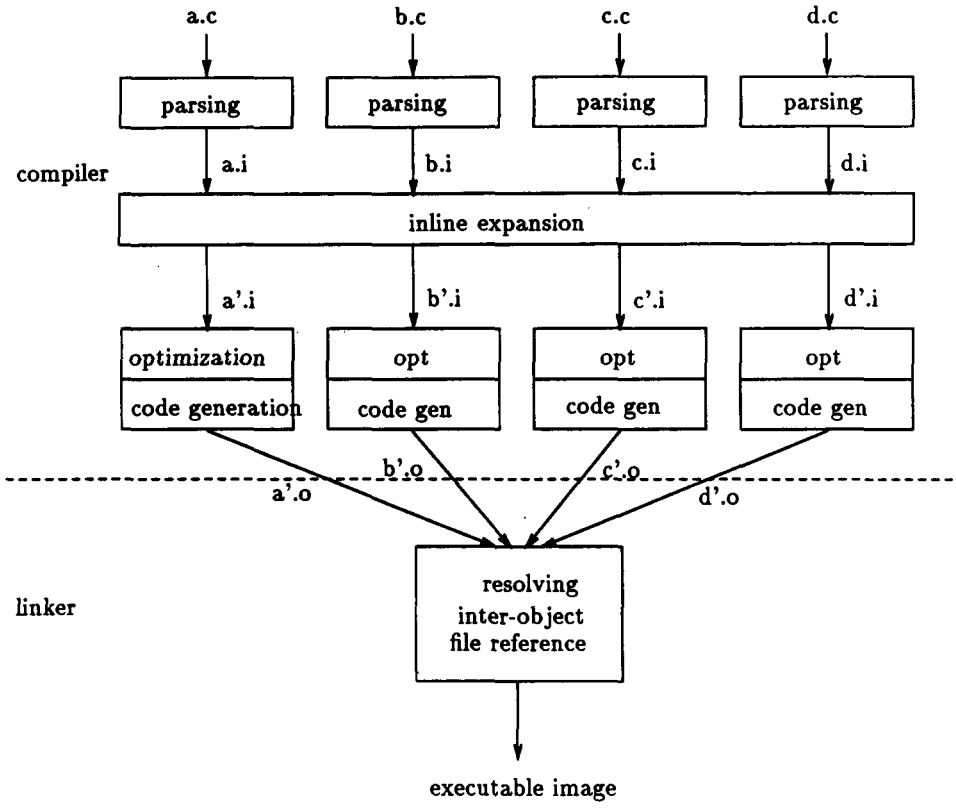

Figure 2. Inlining at compile time

* Note that it is possible to do link-time inline expansion and still perform global optimization. In the MIPS compiler, for instance, a link phase with function inlining can optionally occur before the optimizer and subsequent compilation phases. [15] The same effect is achieved: in line expansion is performed before the optimization steps.

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.

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 behaviour of the represented programs. Secondly, 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 behaviour of a program. A weighted call graph (a directed Multigraph), $G = (N, E, \text{main})$, 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 are determined 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; one is expected to execute one time and the other is expected to execute two times in a typical run.

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.

* To support program development, the inliner can generate a makefile that correctly recompiles the program when a source file is modified. The makefile specifies all the source files that an object file depends on after in lining. When a source file is modified, all the files that received function bodies from the modified source file will be recompiled by invoking the makefile. The problem of program development and debugging with inlining is beyond the scope of this paper and is currently being investigated by the authors. Currently, the inliner serves as a tool to enhance the performance of a program before its production use.

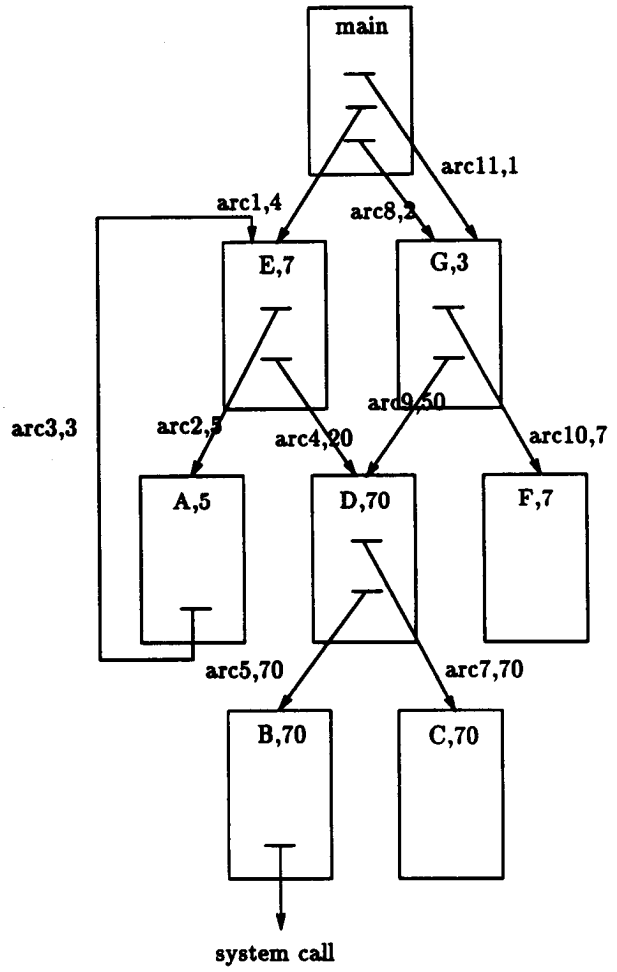

Figure 3. A weighted call graph

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 F 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.[16]

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.

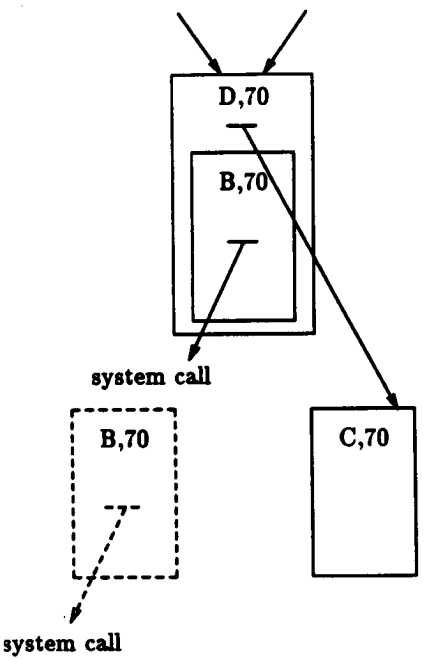

Figure 4. An inlining example

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.

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.

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 IF 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). In the current implementation of our compiler, these calls are excluded from being inlined. This is done by writing the names of this type of functions in a file, and specifying this file as a compiler option. *

* We are currently developing the program analysis required to efficiently handle varargs functions, which is important for programs such as graphics and windowing packages.

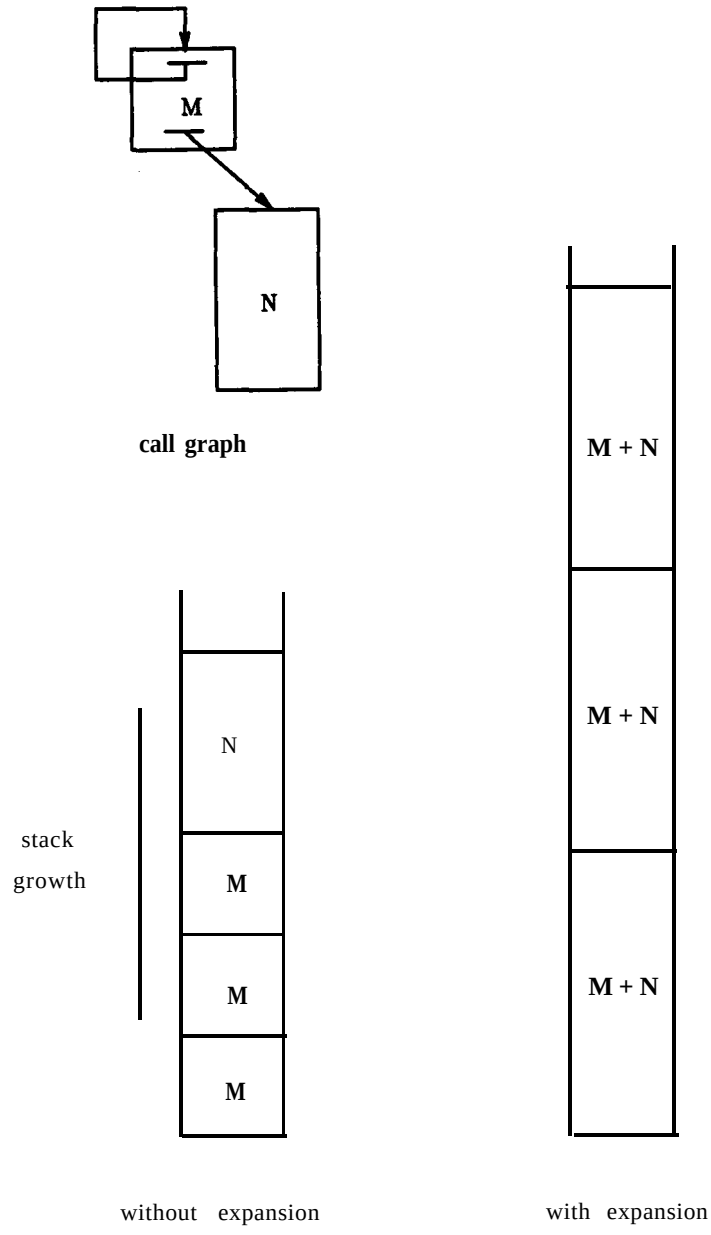

Figure 5. Activation stack explosion

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 expansion, 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 expansion sequence 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 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 $F \rightarrow L$ weight to A and B, separately. Therefore, the decision to absorb L only into A would be based on non-existing 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.

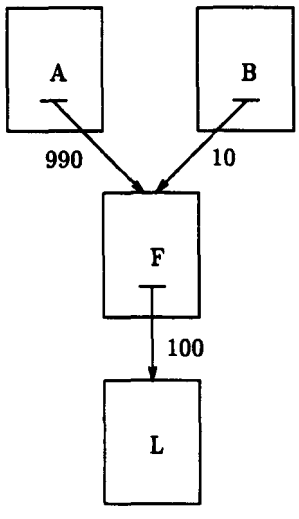

Figure 6. An example of restricted inlining

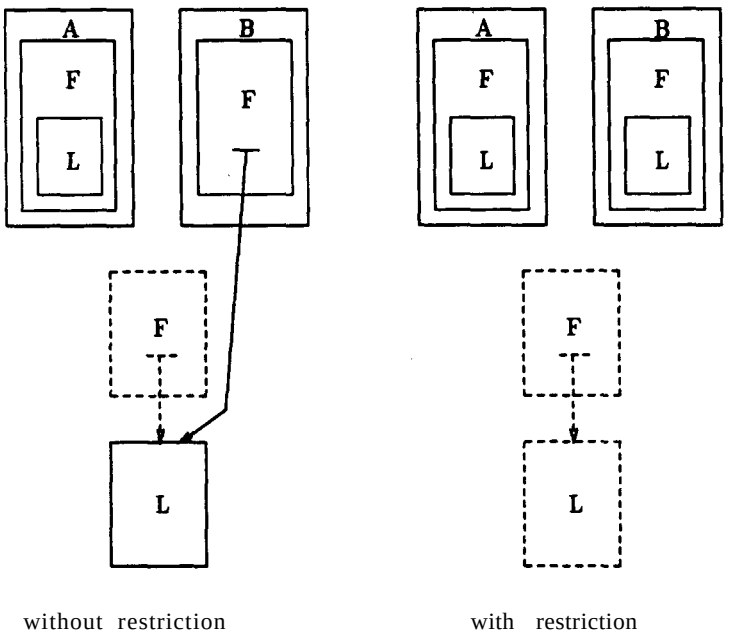

Figure 7. Lost opportunity

After detecting all the hazards due to recursion, the call graph can be simplified by breaking all the cycles. The cycles in the call graph 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.

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

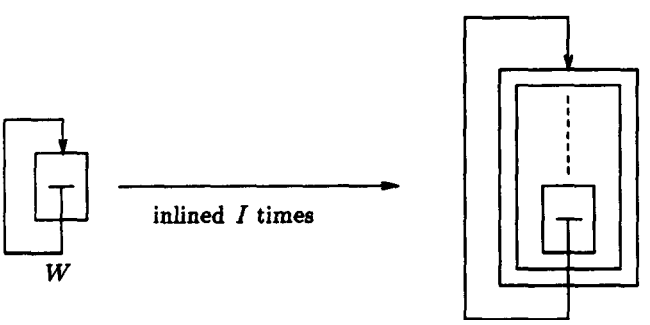

Figure 8. Handling single-function recursions

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, i.e. heavily weighted arcs in the call graph, as possible. However unlimited inline expansion may cause code size explosion. 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 favour large programs. We chose to set the limit as a percentage of the original program size through a compiler option.

Code size expansion increases the memory required to store the program and affects instruction memory hierarchy performance. Precise costs cannot 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. Some original function bodies may become unreachable from the main function and can be eliminated after inlining. Also, a detailed evaluation has shown that code expansion due to inlining does not necessarily reduce the instruction cache performance. [17] Therefore, the cost of inlining is estimated based on the intermediate code size increase rather than the accurate effect on the instruction memory system performance.

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 cannot be known during inline expansion. Therefore, the benefit of inline expansion will be estimated only by the reduction in dynamic function calls.

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 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. [18] However, this straightforward 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.

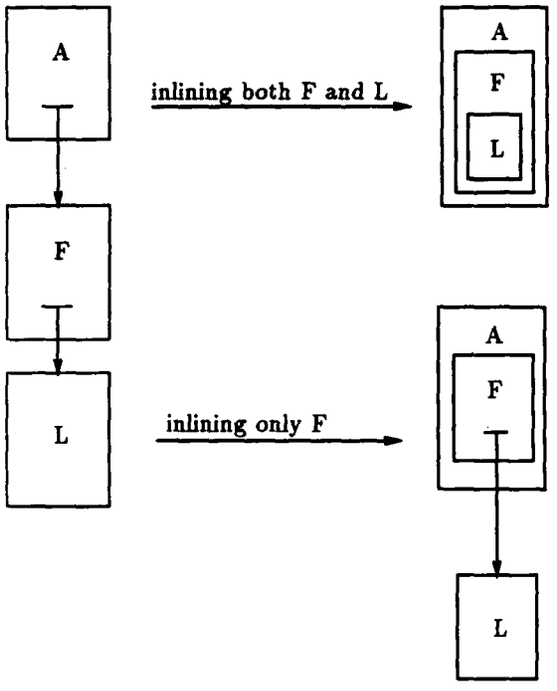

Figure 9. Inter-dependence between code size increase and sequencing

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 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 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 $E \rightarrow D$ and $G \rightarrow D$ are eliminated first. Note that by absorbing D into both E and G (and therefore eliminating $E \rightarrow D$ and $G \rightarrow D$ in two expansion steps), four new arcs are created: $E \rightarrow B$, $E \rightarrow C$, $G \rightarrow B$, and $G \rightarrow C$. 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 six 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 $D \rightarrow B$ and $D \rightarrow C$. Function D, after absorbing B and C, is then absorbed into E and G. This further eliminates $E \rightarrow B$ and $E \rightarrow C$. Note that it only takes a total of four expansion steps to achieve the original goal. The general

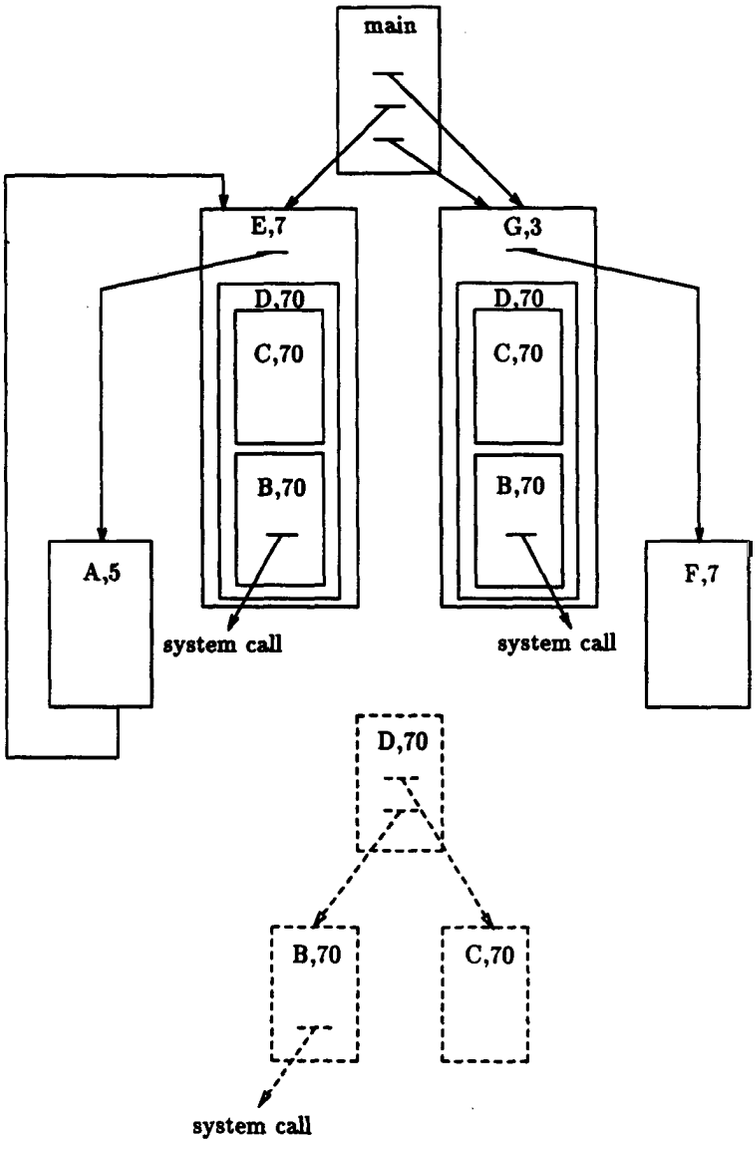

Figure 10. Inlining a function before absorbing its callees

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

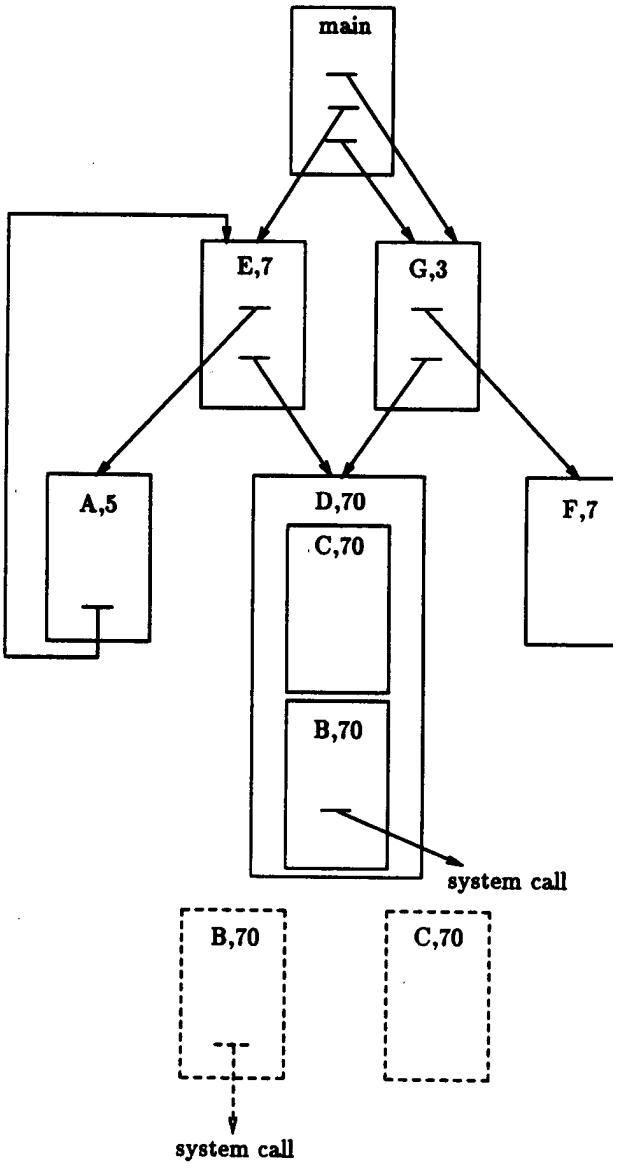

Figure 11. Inlining a function after absorbing its callees

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 call graph. The complexity

of this algorithm is *O(N)*, where *N* is the number of function calls in the program 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 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.

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 between the local, parameter, and static variables of the caller and those of the callee, our C compiler creates a global name space for the entire program in the intermediate representation. This is achieved by a renaming mechanism invoked before inlining.

The inliner handles formal parameters by assigning the actual parameter values to them. The return value has to be assigned to new local temporary variables so that it can be used by the caller. These assignments are often eliminated later by constant and copy propagation and dead code elimination.

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.

EXPERIMENTS

Table I shows the set of classic local and global code optimization that we have implemented in our prototype C compiler. These code optimizations are common in commercial C compilers. We have also implemented a priority-based global register allocator which uses profile information to allocate important variables into processor registers. This register allocator assigns variables to caller-save and callee-save registers intelligently to remove part of the function calling overhead.

Table II 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 III describes the input data that we have used for profiling. The ‘Runs’ column lists the number of inputs for profiling each benchmark program. The ‘Description’ column briefly describes the nature of these input data. Executing each benchmark program with an input produces a profile data file. For each benchmark

Table I. Code optimizations

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 II. Benchmarks

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 III. Characteristics of profile input data

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	Iexers for C, Lisp, Pascal, awk, and pic
tbl	20	ditroff files (1004000 lines)
xlisp	5	gabriel benchmarks
yacc	10	grammars for C, Pascal, pie, eqn, awk, etc.

program, its profile data files are summarized into one profile data file, which is used to guide the automatic inline expander. To evaluate performance, we use a different input data than those used for profiling to measure the execution time of the compiled program.

Table IV describes the static or compile-time characteristics of function calls. *

* We report call sites that are visible to the compiler,

Table IV. Static characteristics of function calls

Name	External	Pointer	Intra-file	Inter-file	Inlined
cccp	143		191	4	23
compress	104	1	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

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 ‘lnter-file’column shows the number of static call sites that call functions in a different source file. The ‘Inlined’ column shows the number of static call sites that are inlined expanded. Table V describes the dynamic or execution-time characteristics of function calls.

Note that several benchmark programs have large fractions 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 reduce the relative importance of external function calls. Our inliner can inline call sites that are shown in both the ‘Inter-file’ and ‘Intra-file’ columns. Tables IV and V show that inlining a small percentage of static call sites removes a large percentage of dynamic calls. Since the number of static call sites inlined directly correspond to the compile time spent on inlining, this result clearly shows that profile information allows the compiler to spend only a small portion of the compile time budget to eliminate most of the dynamic calls.

Table VI 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.

Table V. Dynamic characteristics of function calls

Name	External	Pointer	Intra-file	Inter-file	Inlined
cccp	1,015	140	1,414	3	1,183
compress	25	0	4,283	0	4,276
eqn	5,010	0	6,959	33,534	37,440
espresso	728	60,965	55,696	925,710	689,454
lex	13,375	0	63,240	4,675	56,991
tbl	12,625	0	9,616	37,809	35,504
xlisp	4,486,885	479,473	10,308,201	8,453,735	14,861,487
yacc	31,751	0	34,146	3,323	33,417

Table VI. Code expansion (DEC-3100)

Name	Global	Global + inline	Ratio
cccp	172,564	215,420	1·25
compress	72,300	73,228	1·00
eqn	130,376	157,528	1·21
espresso	311,544	338,508	1·09
lex	156,148	165,468	1·06
tbl	181,064	214,036	1·18
xlisp	267,268	354,092	1·32
yacc	141,268	164,584	1·17

The ‘Ratio’ column shows the code expansion ratios. The average code expansion ratio for the benchmark programs is about 1·16.

Table VII shows the speed-ups of the benchmark programs. The speed-up is calculated based on the real machine execution time on a DEC-3100 workstation. The ‘Global +inline’ column is computed by dividing the execution time of non-inlined code by the execution time of inlined code. Note that intelligent assignment of variables to caller-save and callee-save registers has already removed part of the overhead of function calls in non-inlined code. The average speed-up for the benchmark programs is about 1·11.

Table VIII shows the speed-up comparison of code produced by MIPS CC* and GNU CC† with our final inlined code. The code produced by MIPC CC and GNU CC is slightly slower than our inline code. The speed-up is calculated based on the real machine execution time on a DEC-3100 workstation. Note that MIPS CC performs link-time inline expansion and GNU CC performs intra-file inline expansion, both before code optimization. The purpose of Table VIII is to calibrate our final inlined code with the code generated by other optimizing compilers. Because the compilers have different optimization capabilities, the speed-up comparison should not be used to compare the inline capabilities of these compilers.

Table VII. Speed-ups (DEC-3100)

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

* MIPS CC release 2.1-O4

† GNU CC release 1.37.1-O

Table VIII. Speed comparison with other compilers (DEC-3100)

Name	IMPACT global +inline	MIPS-04	GNU-O
cccp	1·00	0·93	0·92
compress	l·00	0·98	0·94
eqn	1·00	0·92	0·91
espresso	l·00	0·98	0·87
lex	l·00	0·99	0·96
tbl	1·00	0·98	0·93
xlisp	l·00	0·88	0·76
yacc	1·00	1·00	0·90

CONCLUSION

An automatic inliner has been implemented and integrated into an optimizing C compiler. The inliner consists of 5200 lines of commented C statements and accounts for about 3 per cent of the source code in our 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 speed-up for a set of production C programs.

ACKNOWLEDGEMENT

The authors would like to acknowledge Nancy Warter and all members of the IMPACT research group for their support, comment, and suggestions. Special thanks to the anonymous referees whose comments and suggestions helped to improve the quality of this paper significantly. This research has been supported by the U.S. National Science Foundation (NSF) under Grant MIP-8809478, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, Matsuhita Electric Corporation, 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).

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. R. J. Eickemeyer and J. H. Patel, 'Performance evaluation of multiple register sets', *Proceedings of the 14th Annual International Symposium on Computer Architecture,* Pittsburgh, Pennsylvania, 2-5 June 1987, pp. 264–271·
3. T. R. Gross, J. L. Hennessy, S. A. Przybylski, and C. Rowen, 'Measurement and evaluation of the MIPS architecture and processor', *ACM Trans. Computer Systems,* August 1988, pp. 229–257.
4. D. A. Patterson and C. H. Sequin, 'A VLSI RISC', *IEEE Computer,* September 1982, pp. 8–21.
5. 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,* June 1988.
6. R. W. Scheifler, 'An analysis of inline substitution for a structured programming language', *Communications of the ACM,* **20,** (9), (1977).

7. S. Richardson and M. Ganapathi, 'Code optimization across procedures', *IEEE Computer,* February 1989.
8. M. Auslander and M. Hopkins, 'An overview of the PL.8 compiler', *Proceedings of the SIGPLAN Symposium on Compiler Construction,* June 1982.
9. R. M. Stallman, Internals of GNU CC, 1988.
10. F. Chow and J. Hennessy, 'Register allocation by priority-based coloring', *Proceedings of the ACM SIGPLAN Symposium on Compiler Constructions,* June 1984.
11. C. A. Huson, 'An in-line subroutine expander for parafrase', University of Illinois. Champaign-Urbana, 1982.
12. J. W. Davidson and A. M. Holler, 'A study of a C function inliner', *Software—Practice and Experience,* **18,** (8), 775–790 (1988).
13. 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.
14. 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,* June 1989.
15. MIPS Inc., 'MIPS C compiler reference manual'.
16. R. E. Tarjan, *Data Structures and Network Algorithms,* SIAM, Philadelphia, PA., 1983.
17. W. Y. Chen, P. P. Chang, T. M. Conte and W. W. Hwu, 'The effect of code expanding optimizations on instruction cache design', *CRHC-91-17,* Center for Reliable and High-Performance Computing, University of Illinois, Urbana-Champaign, May 1991.
18. 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.