RC 20286 (11/22/95)
Computer Science

IBM Research Report

Automatic Inlining

Paul R. Carini

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Automatic Inlining

Paul R. Carini*

November 16, 1995

Abstract

We present a method for performing automatic procedure integration which selects individual call sites for inlining based on simple hueristics. The hueristics are expressed as two cost functions. One cost function models each procedure and attempts to capture the cost of inlining a procedure. The other cost function models the region around each call site, and attempts to capture the benefit of inlining a particular call site. We describe the implementation of our method in a Fortran90 research compiler prototype and present experimental results. These results demonstrate that our method achieves equal or better execution performance than that obtained by user directed inline expansion on the Fortran subset of the SPECfp92 benchmarks without a significant increase in executable size or compilation time.

1 Introduction

Inline expansion, or Inlining, is the replacement of a procedure call with the body of the called procedure. Inlining is a well known [1, 14, 20, 19] compile time optimization which can substantially improve the execution performance of programs. The increasing popularity of object-oriented languages, such as C++, accelerates the trend toward programs created from smaller and more numerous procedures and provides greater opportunity for inlining to improve execution performance.

Inlining provides performance improvements for three principal reasons. First, the procedure linkage is removed, including the code to save and restore registers, allocate stack space, and the branch itself. Second, the code surrounding the call site can be improved, since the call itself, which is usually a barrier for many optimizations, is no longer present. Third, the procedure code that is substituted for the call can be optimized for the specific call context. In practice, inlining is not always beneficial [15, 12], and may degrade the execution performance of some programs. Inlining may also increase binary code size and compilation time. Thus, inlining cannot be performed indiscriminately.

Compilers typically provide inlining function via two different paradigms, user directed inline expansion, or userinline mode, and automatic inline expansion, or autoinline mode. Userinline mode requires that the user or programmer specify which procedures should be inlined. The compiler then attempts to inline the procedure at each of its call sites. Userinline mode typically requires repeated cycles of compilation and execution, as different sets of procedures are selected for inlining and the resulting execution performance measured. Profiling information can aid this process by identifying likely candidates for inlining. The attainment of peak performance with userinline mode usually requires considerable programmer time and effort.

In autoinline mode, the compiler determines which procedures should be inlined. Autoinline mode is more convenient for the user since it typically requires no more than a compile time switch. However, simple implementations which use a single characteristic, such as procedure size or nesting level, have not been able to match the performance that can be obtained with userinline mode.

*I.B.M. T.J. Watson Research Center, P.O. Box 704 Yorktown Heights, NY 10598, carini@watson.ibm.com, 914 784 7911

One approach to improve the effectiveness of automatic inline expansion relies on profiling information to select the call sites which should be inlined [8]. Profiling information can be used to obtain performance from autoinline mode which is equivalent to userinline mode but is less convenient for the user. The use of profiling information typically requires at least two compilations of the program, one which is used to generate the profiling information, and the other which is used to perform the automatic inlining. The performance improvement obtained with profiling based automatic inlining may also be sensitive to the input data set that was used to generate the profiling information.

We have implemented an automatic inlining method in a Fortran90 research prototype compiler which does not rely on profiling information (although profiling information can be used if it is available). Our method selects individual call sites for inlining based on simple hueristics. The hueristics are expressed as two cost functions. One cost function models each procedure and attempts to capture the cost of inlining a procedure. The other cost function models the region around each call site and attempts to capture the benefit of inlining a particular call site. Our automatic inlining mode obtains equivalent or *better* execution performance on the Fortran subset of the SPECfp92 [2] benchmarks than the best performance obtained from userinline mode. This level of performance is obtained with approximately the same code size and compilation time. We believe that we are the first to demonstrate an automatic inlining method which does not rely on profiling information and provides performance equivalent to that obtained by user directed inlining.

Section 2 describes our compilation model, which fully supports interprocedural analysis and transformations, including automatic inlining and cloning. Section 3 describes our method for performing automatic inlining. We provide a detailed description of our routine and call site cost functions and describe their use. Measurements of the results obtained with our autoinline mode are presented in Section 4. Measurements are also provided for userinline mode and when no inlining is performed. These results are contrasted with the results for autoinline mode. Section 5 discusses related work, and Section 6 presents our conclusions.

2 Compilation Model

Our Fortran90 research compiler prototype was designed to support a full range of interprocedural analyses and transformations. Automatic inlining and cloning can be performed with or without interprocedural analysis (IPA). Many of the underlying mechanisms that were developed to support the IPA phase are also used in the autoinline mode. Both intra-file and inter-file (cross-file) inlining and cloning are supported. The compilation model is based on the *separate compilation* model, which first compiles source files to object files, and then links them. Our interprocedural compilation model is fully compatible with automated program generators and tools, such as the UNIX *make* and *sccs* facilities.

The compilation model provides an IPA collection phase, during which each procedure in the program is visited and the IPA inputs are collected and saved for later use. This information includes such items as procedure names and formal parameters, procedure call sites and arguments, global variables which are immediately modified or referenced, and an intermediate representation of the procedure. This information is stored in the object module, which is created for each source file. Also collected and saved during this phase are the inlining and cloning inputs that are used by the cost functions to characterize each procedure and call site. For the routine cost function, these include measurements of the size of the procedure, complexity, number of calls, and number of I/O calls, for each procedure. For the call site cost function, these include the size of the region containing the call, the loop nesting level of the call, the number of conditionals along the path to the call, and whether or not any of the parameters of the call are used in loop bounds, array bounds, or conditional statements of the called procedure. Figure 1 illustrates the steps of the compilation model.

After the complete program has been visited, (the model includes a provision for handling missing procedures), the interprocedural phase begins by constructing the *program call graph* (PCG). Each procedure is visited, the inlining inputs are collected, and an intermediate representation is generated,

1. Collect IPA inputs
2. Construct the Program Call Graph
3. Perform Reverse Topological Traversal

Figure 1: Compilation Model Overview

1. Inline Previously Selected Outgoing Edges
2. Select Incoming Edges for Inlining or Cloning
3. Perform Optimizations and Transformations
4. Generate Object Code

Figure 2: Reverse Topological Traversal

during the compilation phase of the separate compilation model. The interprocedural phase of our model is equivalent to the link phase of the separate compilation model. Procedures can be missing for various reasons, including the compilation of an incomplete program, calls to procedures written in another language, or calls to library procedures for which only the object code is available. The interprocedural phase detects missing procedures and reports them to the programmer. A compiler option is available which instructs the interprocedural phase to ignore all missing procedures. Alternatively, the programmer can specify a list of missing procedures that should be ignored. This option can also be used by the programmer to characterize the interprocedural behavior of the missing procedures.

If interprocedural analysis is selected, then it is performed in the following sequence. An interprocedural alias analysis [9, 6] is performed, which includes both reference parameter and pointer-induced alias analyses. This is followed by a phase that includes the computation of interprocedural MOD and REF information [3, 10, 4], which is implemented as a flow-insensitive traversal of the PCG. This analysis relies upon the IPA inputs that are initially collected. Next, interprocedural constant propagation [7] is performed, which uses the interprocedural MOD and REF information along with the IPA inputs. The interprocedural constant propagation consists of two separate steps, the computation of interprocedural constants, and the transformation of a program representation to reflect these constants.

The final phase of interprocedural analysis is a reverse topological traversal (*backward walk*) of the PCG, which is illustrated in Figure 2.[1] During this traversal each procedure (node) is visited once. The intermediate representation of the procedure is first restored to memory, if required. The autoinline mechanism determines whether any of the incoming edges (which represent calls of this procedure) should be inlined. The outgoing edges (which represent calls from this procedure) are checked and procedure inlining and cloning may be performed to eliminate these call sites. At this stage, the intermediate representation is transformed to reflect the results of interprocedural constant propagation, other traditional analyses and transformations are performed, upward exposed use is computed and saved, and optimized object code is generated for this procedure. The inline mechanism does not attempt to inline calls along back edges. This prevents the inlining of recursive calls.

3 Method

In this section we describe our method for performing automatic inlining. We describe the routine and call site cost functions and explain how the automatic inlining determination is performed.

The automatic inline expansion method has three main objectives. The first objective is to eliminate the call overhead. The removal of a call site eliminates the requirement for setting up the

[1] In the presence of PCG cycles, topological order is defined by the removal of back edges.

subroutine linkage on both sides of the call, including the storage and retrieval of register values and the allocation of stack space. The branch instruction is also eliminated. The second objective is to improve the optimization of the code in the caller which surrounds the call site. By removing the call site, which is a barrier for many optimizations, it is possible to perform better instruction scheduling, register allocation, etc. The third objective is to tailor the code of the callee to the call site. Some simplifications may result from optimizations such as unreachable code elimination or invariant if removal, when the arguments at this call site are propagated into the callee code.

3.1 Routine Selection

The routine cost function determines the suitability of a routine for inlining at any call site. This cost function selects routines which are typically short (small) and simple. Procedures which are large or contain lots of control flow, I/O, or call sites have a higher cost. Only procedures whose cost is below a threshold value are candidates for inlining.

The inline routine cost is computed with the following function:

$$inline\_routine\_cost = size/X1 + complexity/X2 + call\_count/X3 + iocall\_count/X4; \qquad (1)$$

Note that all of the quantities in the formula are integers, and X1, X2, X3, and X4 are tunable parameters. The *size*, *complexity*, *call_count*, and *iocall_count* values are all collected during the *IPA* collection phase for each procedure.

Size is determined by counting the number of statements in the intermediate representation of the procedure. This measure favors small or short procedures by increasing the cost based on the size of the procedure.

Complexity is determined by counting the number of control flow and loop statements in the procedure. Since a loop statement is a more complicated form of control flow which is likely to be repeated, it is given extra weight. In our implementation, each loop statement is counted as three control flow statements. This measure favors simple straight line procedures by increasing the cost of control flow statements, and especially loop statements.

Call_count is determined by counting the number of call sites in the procedure.

Iocall_count is determined by counting the number of I/O calls in the procedure. These measures favor procedures without call sites by increasing the cost of calls and I/O calls.

The threshold value is increased for procedures that are leaf procedures or have formal parameters which are used to determine control flow within a procedure, as an array bound, or as a loop bound. The amount of the increase is also a tunable parameter.

3.2 Call Site Selection

The call site cost function governs call site selection. This cost function enhances the selection of call sites which are on the execution path, are deeply nested within loops, and are contained within a small region of statements. The cost of call sites which are unlikely to be executed, or are in a large region and may potentially overflow the optimizer, are increased. Only call sites whose cost is below a threshold value are candidates for inlining.

The call site cost is computed by the following function for each call site:

$$call\_site\_cost = block\_size/Y1 + paths/Y2 - nesting/Y3; \qquad (2)$$

Note that all of the quantities in the formula are integers, and Y1, Y2, and Y3 are tunable parameters. The *block_size*, *paths*, and *nesting* values are all collected during the *IPA* input phase for each procedure.

Block_size is determined by counting the number of statements in the intermediate representation which are between statements of major control flow. This measure favors a call site surrounded by a small region by increasing the cost based on the size of the region. The intent is to avoid the creation of a region which is too large to be optimized effectively.

Paths is an estimate of the number of execution paths which may not traverse the call site. This is determined by counting the number of "open" conditional branches which are encountered before the call site. A conditional branch is no longer open if, for example, the THEN and ELSE bodies of a conditional IF have been passed. This measure favors call sites which are more likely to be on the execution path and could be easily improved through the use of profiling information.

Nesting is determined by computing the level of loop nesting which surrounds the call site. Since the value of a call site increases with increased nesting level, this parameter is subtracted and favors nested loop call sites. Nested loop call sites are more likely to provide significant improvement due to the reduction in call overhead.

During the *IPA* collection phase, formal parameters which are used to determine control flow within a procedure, as an array bound, or as a loop bound, are noted. The threshold value is increased for call sites which have constant arguments that are mapped to these "noted" formal parameters. The amount of the increase is also a tunable parameter. Interprocedural constant propagation can be used to increase the number of constant arguments within a program. The threshold value is also increased for call sites which are deeply nested.

3.3 Inlining Determination

Automatic inlining is performed during the *IPA* backward walk. As each procedure is processed during the backward walk, the routine cost function is computed. Then the call site cost function is computed for each incoming edge in the *PCG* and a separate inlining decision is made for each call site. Automatic inlining is performed at a call site only if both the routine cost function and the call site cost function are below their respective threshold values. This mechanism makes the inlining decision call site specific. Figure 3 presents our automatic inlining algorithm.

4 Results

Our experiment compares the effectiveness of automatic inlining with user directed inlining. We compare the execution performance, executable code size, and compilation time. All of our measurements were performed by using our prototype compiler with optimizations enabled, on an IBM POWER2 Model 590 machine, running AIX. In the accompanying tables, the only differences between the measurements displayed in any two columns is the amount or method of inlining that was performed. The same set of autoinline mode tunable parameters were used throughout the experiment. Our measurements were made on the Fortran subset of the SPECfp92 benchmarks[2] (except 047.TOMCATV, which has no call sites). See [7, 5, 17] for an analysis of interprocedural aspects of the FORTRAN subset of the SPEC benchmarks.

Since the current version of the prototype does not correctly compile the 013.spice2g6, 039.wave5, 089.su2cor, 093.nasa7, and 094.fpppp benchmarks with any form of inlining enabled, no results are available for these benchmarks. We present results for the remaining 6 SPECfp92 benchmarks. The results reported for userinline mode were obtained with the userinline options determined by a performance group at the IBM Santa Teresa Development Laboratory. Profiling information was used as a starting point, and many experiments were conducted by the performance group to determine the best set of procedures that should be inlined for each program, in order to obtain the best possible performance. Since userinline mode does not improve the performance of the 078.swm256 benchmark,

```
Collect ipa inputs, routine and call site cost function inputs;
Build PCG;
procedure auto_inline
    for each procedure p, in a backward topological traversal of the PCG
        /* Process the call sites within p first */
        for each out edge e of p
            Inline each edge marked INLINE;
            /* adjust the cost of p to reflect the inlined procedure */
            add the routine cost inputs of the inlined procedure to p;
        end loop
        Compute the routine_cost_function for p;
        if the routine_cost_function(p) ¡ threshold[p] then
            /* process the call sites that invoke p */
            for each in edge e of p
                Compute the call_site_cost_function for e;
                if the call_site_cost_function(e) ¡ threshold[e] then
                    mark edge e INLINE;
                end if
            end loop
        end if
    end loop
```

Figure 3: Procedure auto_inline

Selected Procedures					
program	Procedures	UserInline	Pct	AutoInline	Pct
015.doduc	40	13	32	15	38
034.mdljdp2	34	7	21	17	50
048.ora	1	1	100	1	100
077.mdljsp2	34	7	21	17	50
078.swm256	7	*	0	6	86
090.hydro2d	39	3	8	20	51
Total	155	31	20	76	49

Table 1: Number of Selected Procedures

Selected Call Sites					
program	Call Sites	UserInline	Pct	AutoInline	Pct
015.doduc	110	60	55	58	53
034.mdljdp2	124	15	12	34	27
048.ora	1	1	100	1	100
077.mdljsp2	124	15	12	34	27
078.swm256	18	*	0	17	95
090.hydro2d	98	7	7	68	69
Total	475	98	21	212	45

Table 2: Number of Selected Call Sites

we have used the non-inlined results for this benchmark to compute the userinline totals for the figures in this section.

In each table, the first column lists the program. The second column lists characteristics measured or results obtained when no inlining is performed. The next two columns report the results obtained from userinline mode. The measurements are listed first; and the results are presented as a percentage of the first column. The last two columns report the results obtained from our implementation of autoinline mode, again the measurements are presented first; and the results are presented as a percentage of the first column.

The first column in Table 1 lists the total number of procedures that are reachable from the main procedure in each program. The main procedure is not counted. Block Data subroutines are also excluded. The remaining columns list the number and percent of procedures inlined by each mode. The number of procedures that are inlined by the autoinline mode is much larger than the number of procedures specified in userinline mode, 49% to 20%, as expected.

The first column in Table 2 lists the total number of call sites in each program. The remaining columns list the number and percent of call sites inlined by each mode. The number of call sites that are expanded by the autoinline mode is much larger than the number expanded by userinline mode, 45% to 21%, but the difference is slightly smaller than the procedure difference. Autoinline mode chooses to inline 58 call sites of 15 procedures in 015.doduc, while 60 call sites of 13 procedures are inlined in userinline mode. This result clearly shows the call site specific nature of autoinline mode.

The first column in Table 3 lists the optimized execution performance of each benchmark. The reported measurements are in seconds, and correspond to the user time reported by the *time* command, in multi-user mode. We consider times that differ by .1 second to be equivalent. With the exception of the inline option, the same options were used in each mode (for each column) to facilitate a fair comparison. The highest level of optimization was used. The execution times reported here may not be the best that have been obtained by the prototype, but do provide a basis for comparing the

Execution Times					
program	Optimized	UserInline	Pct	AutoInline	Pct
015.doduc	18.4	17.2	93	17.3	94
034.mdljdp2	31.7	31.2	98	31.3	99
048.ora	37.8	35.4	94	35.5	94
077.mdljsp2	49.4	49.1	99	48.5	98
078.swm256	73.3	*	100	72.6	99
090.hydro2d	84.4	84.2	100	83.7	99
Total	295.0	290.4	98.4	288.9	97.9

Table 3: Execution Times

Executable Size					
program	Optimized	UserInline	Pct	AutoInline	Pct
015.doduc	211930	228706	108	242258	114
034.mdljdp2	241460	238134	99	237015	98
048.ora	10941	10349	95	10349	95
077.mdljsp2	203744	199658	98	198519	97
078.swm256	3729897	*	100	3728989	100
090.hydro2d	696413	706245	101	712685	102
Total	5094385	5112989	100.4	5129815	100.7

Table 4: Executable Code Size

performance of the different modes. The remaining columns list the execution performance observed with each mode. These measurements indicate that inlining provides measurable performance gains in the 6 benchmarks for which results have been obtained. Autoinline mode provides the same improvement as Userinline mode for the first 3 benchmarks, and provides better performance for the remaining three benchmarks: 077.mdljsp2, 078.swm256, and 090.hydro2d. The results in the totals column indicate that the improvement obtained by autoinline mode is 25% better than that achieved by userinline mode, a 2.1% versus a 1.6% improvement. The largest improvement, 6%, was obtained in 015.doduc.

The first column in Table 4 lists the executable code size of the optimized binary for each benchmark. The measurements are reported in bytes, and are the total from the *size* command. The remaining columns list the executable code size obtained by the various modes. Both userinline and autoinline mode decrease the size of 3 of the 6 binaries; 2 become larger, and 1 essentially remains the same. The largest decrease in both userinline mode and autoinline mode is observed in 048.ora, and is 5% in both cases. The largest increase in both userinline and autoinline mode is observed in 015.doduc, and is 8% and 14%, respectively. The Total column shows that both userinline and autoinline modes have a negligible impact on object code size, with respective increases of .4% and .7%, respectively. To provide some perspective, the sum of the object code size of the first 4 benchmarks increases by 8% (from 612755 to 668075) when the optimization level is increased from O4 to O5, the highest level.

Note that all but one of the call sites in 078.swm256 is expanded by the autoinline mode. If the remaining call site is also inlined, then the object code size decreases by 288 bytes, and the execution time increases by .2 seconds.

The first column in Table 5 lists the optimized compilation time in seconds for each benchmark. The remaining columns list the compilation times obtained by the various modes. Both userinline and autoinline mode increase the compilation time by 50%, as the totals indicate. The numbers clearly

Compilation Times					
program	Optimized	UserInline	Pct	AutoInline	Pct
015.doduc	209	460	220	346	165
034.mdljdp2	60	50	83	89	148
048.ora	6	6	100	7	116
077.mdljsp2	56	55	98	84	150
078.swm256	27	*	100	26	96
090.hydro2d	176	205	116	250	142
Total	534	803	150.3	802	150.2

Table 5: Compilation Times for the SPECfp92 Benchmarks

indicate that our implementation of autoinline mode does not have an additional compilation time cost above userinline mode.

5 Related Work

Chang et al.[8, 19] describe the IMPACT-I system, an implementation of a profile guided automatic inline expansion system for C programs. Since the only mode of operation of their system incorporates profiling, it is not directly comparable. Our compilation model is functionally very similar to theirs, in that we also perform the inline expansion before optimization and code generation.

Richardson and Ganapathi[20] compare the benefits that can be obtained from procedure integration and interprocedural analysis. They provide measurements which indicate that procedure integration can provide substantial performance improvements.

Davidson and Holler[15] develop a formalism for characterizing the effects of inlining on execution performance. They show that the increased size of the inlined program does not have detrimental effects on performance. Davidson and Holler also show that inlining is not always beneficial, primarily due to an increase in register pressure. Davidson and Holler[14] describe INLINER, an implementation of a C source to source automatic inlining method within a source file which achieves impressive performance gains for some programs. No comparison is provided with the performance that can be obtained from user directed inlining.

Cooper et al.[11, 13, 12] present a study which characterizes the interaction of procedure integration with high levels of compiler optimization. They report disappointing results for the benefits of inlining in Fortran programs, and suggest that procedure cloning is a more effective technique for improving the performance of Fortran programs.

Kaser et al.[18] describe a method for converting indirect to direct recursion. Their compilation model is similar to ours, in that they also use a program call graph. We avoid recursion by preventing inlining along back edges in the PCG. They also argue that original-version inlining is superior to the usual method, current-version inlining, that we employ.

Dean and Chambers[16] describe a method for performing automatic inlining in SELF that is based on a constantly updated database of profiling information.

6 Conclusions

This paper describes a method for performing automatic inlining that is based on procedure and call site cost functions. The two cost functions permit the compiler to perform an individual inlining determination at each call site, which leads to better performance. Automatic inlining provides the same function as user directed inlining, but is more convenient in practice, and requires less user effort and time.

We describe our implementation in a Fortran90 research prototype compiler and present experimental results which demonstrate that this automatic inlining method is more effective than user directed inlining. Our automatic inlining mode provides a performance improvement which is equal or better than that which is obtained from user directed inlining for the Fortran subset of the SPECfp92 benchmarks. The executable code size showed a negligible increase and remained comparable both to those obtained by userinline mode on these benchmarks, and to the optimized code size. The compilation time cost for autoinline mode remained the same as the compilation time for userinline mode, both of which are 50% longer than the optimized mode.

The increasing popularity of object-oriented programming languages, such as C++, and the resulting trend toward more numerous, smaller, procedures, make automatic inlining an even more important tool for compiler optimization.

7 Acknowledgements

We thank Fran Allen for suggesting that we examine this problem. We thank Rajeev Gulati for his implementation of userinline mode. We acknowledge the countless measurements of the STL performance group, including Joe Brownsmith and Kalyan Muthukumar. We thank Chin Li for his assistance with the tuning of autoinline mode. We are grateful to Michael Burke, Michael Hind, Kalyan Muthukumar, Lee Nackman, and Laureen Treacy for suggested improvements to earlier drafts of this paper.

References

[1] F.E. Allen and J. Cocke. A catalogue of optimizing transformations. In Randall Rustin, editor, *Design and Optimization of a Compiler*, chapter 1, pages 1–30. Prentice-Hall, Inc., 1972.

[2] Subra Balan and Walter Bays editors. Spec announces new benchmark suites cint92 and cfp92. Technical report, Systems Performance Evualation Cooperative, March 1992. *SPEC Newsletter* 4(1).

[3] John Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. *6th Annual ACM Symposium on the Principles of Programming Languages*, pages 29–41, January 1979.

[4] Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[5] Michael Burke and Paul Carini. Compile-Time Measurements of Interprocedural Data-Sharing in FORTRAN Programs. RC 17389 76684, IBM – T.J. Watson Research Center, November 1991.

[6] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *7th Workshop on Languages and Compilers for Parallel Computing*, 1994. Extended version published as Research Report RC 18546, IBM T. J. Watson Research Center, September, 1994.

[7] Paul R. Carini and Michael Hind. Flow-Sensitive Interprocedural Constant Propagation. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, 1995. *To Appear.*

[8] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen mei W. Hwu. Profile-guided Automatic Inline Expansion for C Programs. *Software – Practice and Experience*, 22(5):349–369, May 1992.

[9] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induces aliases and side effects. In *20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 232–245, January 1993.

[10] Keith Cooper and Ken Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *SIGPLAN '84 Symposium on Compiler Construction*, volume 19, pages 247–258, June 1984. *SIGPLAN Notices* 19(6).

[11] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. A Methodology For Procedure Cloning. *Computer Languages*, 19(2):105–117, February 1993.

[12] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Software – Practice and Experience*, 21(6):581–601, June 1991.

[13] Keith D. Cooper, Mary W. Hall, and Linda Torczon. Unexpected side effects of inline substitution: A case study. *ACM Letters on Programming Languages and Systems*, 1(1):22–32, March 1992.

[14] Jack W. Davidson and Anne M. Holler. A Study of a C Function Inliner. *Software – Practice and Experience*, 18(8):775–790, August 1988.

[15] Jack W. Davidson and Anne M. Holler. Subprogram Inlining: A Study of its Effects on Program Execution Time. *IEEE Transactions on Software Engineering*, 18(2):89–102, February 1992.

[16] Jeffrey Dean and Craig Chambers. Training compilers to make better inlining decsisions. Technical report, Department of Computer Science and Engineering, University of Washington, 1993. Technical Report 93-05-05.

[17] Michael Hind, Michael Burke, Paul Carini, and Sam Midkiff. An emperical study of precise interprocedural array analysis. *Scientific Programming*, 3(3):255–271, 1994.

[18] Owen Kaser, C. R. Ramakrishnan, and Shaunak Pawagi. On the Conversion of Indirect to Direct Recursion. *ACM Letters on Programming Languages and Systems*, 2(1):151–164, March 1993.

[19] Wen mei W. Hwu and Pohua P. Chang. Inline Function Expansion for Compiling C Programs. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24, pages 246–255, June 1989. *SIGPLAN Notices* 24(6).

[20] Stephen Richardson and Mahadevan Ganapathi. Interprocedural analysis versus procedure integration. *Information Processing Letters*, 32(3):137–142, August 1989.