Improving Parallelism after Inline Substitution

Mary Hall

CRPC-TR90061
July, 1990

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Improving Parallelism after Inline Substitution

Mary Wolcott Hall *

Department of Computer Science
Rice University
P.O. Box 1892
Houston, Texas 77251
(713) 527-6077
marf@rice.edu

Abstract

This paper describes some properties of programs after inline substitution. Optimizations are outlined to take advantage of these properties in order to improve the parallelism of such programs. These optimizations can be incorporated into a compiler that already supports automatic or programmer-specified inlining to increase parallelism in programs with inlined procedure calls. The effectiveness of these optimizations is demonstrated with experimental results.

Keywords: inline substitution, optimization, parallelization, loop distribution, loop-invariant code, induction variable, range analysis, scalar expansion

*This research has been supported by National Science Foundation grants CCR 86-19893 and CCR 87-06229, Control Data Corporation and IBM Corporation.

1 Introduction

While using many short procedures is generally accepted as a preferred programming style, procedure calls may greatly impair the ability of the compiler to perform optimizations on the program. Because the compiler lacks knowledge about the effects of the procedure calls, it is forced to make worst-case assumptions about the variables used and modified in the called procedure. Procedure calls pose an even greater problem to parallelizing compilers since the inability to optimize around procedure calls can result in a significant loss in potential run-time performance improvements.

By performing *inline substitution*, first introduced as open procedure linkage in [AC72], the compiler is able to see the effects of the procedure call directly. A procedure call is substituted *inline* by replacing the call with the body of the called procedure, with the actual parameters of the call replacing the procedure's formal parameters.

Although inline substitution can produce substantial improvements in parallelizing modular programs, often in scientific programs, even after inlining procedure calls appearing in loops, too many dependences remain to gain improvement in parallelism [CHT90]. By examining such loops and the nature of the remaining dependences, some qualities of loops containing inlined call sites become apparent. Loops with inlined call sites exhibit properties unlike human-generated code, and so, optimizations that may not be profitable on code written by humans may be useful in improving the quality of code after inlining.

This paper presents observations about some unique properties of loops with inlined procedure calls. Based on these observations, we describe optimizations that promote parallelism in loops with such properties. This work suggests that a compiler that supports automatic or programmer-specified inline substitution can more effectively parallelize a program by performing these optimizations either before analyzing dependences between variables or, in cases where the optimizations can possibly cause execution-time performance degradation, during dependence analysis.

The organization of the paper is as follows: the next section discusses related work; section 3 presents the observed properties of loops with inlined calls and the optimizations exploiting these properties; section 4 demonstrates the value of these optimizations with experimental results; and section 5 concludes the paper.

2 Related Work

Inline substitution was first suggested as an optimization in [AC72]. It is most commonly used to reduce the overhead of saving and restoring state at procedure calls, and for this purpose, is often applied in compilers for Lisp and other languages where procedures are very short and call sites are prevalent. Several studies have tried to assess the value of inlining for this purpose, most notably Scheifler's, where significant improvements were observed [Sch77].

In addition to eliminating the overhead of procedure calls, a more interesting goal of inline substitution is to improve the compiler's ability to perform optimization. To this end, the Experimental Compiling System project used inlining as a key component of an optimizing compiler, expressing source-level operations as defining procedures in some intermediate representation and replacing the source with the body of the defining procedures as needed [ACF+80]. Hecht's design of a compiler for a restricted language is another example of inlining used to promote optimization [Hec77].

Studies to assess the value of inlining for enhancing optimization include Richardson and Ganapathi's study of inlining in a Pascal optimizing compiler [RG89], Husson's study of inlining to promote parallelism in Fortran for Parafrase [Hus82], and the study at Rice examining the benefits of inlining for both scalar and parallel optimizations in Fortran [CHT90]. In particular, our study showed that while inlining could provide substantial improvements in some cases, in others it was actually the

cause of performance degradation. In terms of enhancing parallelism, the improvements were only significant in a few cases.

A final class of related work describes the types of optimizations most often enabled by inlining. Ball proposed a model for predicting the benefits of inlining a particular call site, considering only improvements arising from substituting constant actual parameters — constant propagation, elimination of unnecessary tests and unreachable code [Bal79]. Wegman and Zadeck also considered a major benefit of inlining to be the combination of constant propagation and eliminating unreachable code [WZ89]. Finally, the extended algorithms for loop-invariant code motion described in the next section were due to the observation that certain control structures occur frequently in loops after inlining is applied [CLZ86].

3 Properties of Inlined Programs

After inlining, some loops exhibited properties inhibiting optimization that would not likely appear in human-generated code. These properties can be categorized in the following way:

1. Unreachable code.
2. Loop-invariant code.
3. Bounds checking.
4. Partial parallelism.

This section describes how these properties arise from inlining and suggests optimizations to enhance parallelism by taking these properties into account. Most of the examples used in this section are taken from our sample programs.

3.1 Unreachable code.

Unreachable code can result from inline substitution when constant actual parameters appear at the inlined procedure call. As a result of replacing constant parameters in the procedure body, tests based on the value of these parameters can be evaluated at compile time. Thus, the tests themselves can be eliminated, and if they evaluate to false, code conditionally executed based on such tests can also be eliminated.

One obvious benefit of eliminating unreachable code is that less of the program needs to be examined for dependence analysis and other optimizations. However, a more important reason to eliminate unreachable code is because it can contribute control dependences that inhibit parallelism. Consider the following example:

```
do $i = 1, 10$
    if $(1 \neq 1)$ $j = i$
    y(i) = y(i) + j
enddo
```

With the conditional statement appearing in the loop, on any iteration of the loop it is unclear to the dependence analyzer whether j is receiving a new value or using the previous value. Thus, the assignment to $y(i)$ may not be run in parallel. After eliminating the conditional statement, it becomes obvious that j receives its value from outside the loop, and the assignment to $y(i)$ can then be parallelized. Note that if the condition instead evaluated to *true*, eliminating the test and leaving the assignment to j would also allow us to run the loop in parallel.

Unreachable code elimination occurs before dependence analysis in many compilers; however, by discussing it here, we emphasize its importance after inlining. Additionally, since the optimizations in this paper were applied by hand in order to gather the experimental results, eliminating unreachable code was essential to make the program understandable when applying the remaining optimizations.

3.2 Loop-invariant code

Because our goal is to be able to parallelize more loops after inlining, the call sites usually selected for inlining are those that appear within loops. When a procedure is called repeatedly within a loop, it is often the case that there is some initialization code appearing at the beginning of the procedure body that only needs to be executed once within the loop. Inlining exposes the opportunity to move this code outside of the loop body.

Moving loop-invariant code out of loops is profitable even within a scalar optimizing compiler because doing so can greatly reduce the amount of computation performed at run-time, especially when the loop executes a large number of iterations. However, there is an additional reason why it is profitable in a parallelizing compiler. Just like unreachable code, loop-invariant code can contribute dependences that inhibit parallelization. Consider the following variation of the previous example:

```
do $i = 1, 10$
    if $(x \neq 1)$ $j = 1$
    y(i) = y(i) + j
enddo
```

A typical dependence analyzer would detect the dependence of the assignment to $y(i)$ upon the conditional assignment to j in the previous statement. However, the value of the condition $(x \neq 1)$ is loop-invariant, so j will have the same value on each iteration of the loop. Thus, the conditional assignment to j can be moved outside the loop, and the loop can then be parallelized.

The traditional loop-invariant code motion algorithm is used to eliminate a single expression at a time. To eliminate control dependences and improve parallelism, compound statements such as loops and conditionals must be located and moved outside of the loop. The algorithms in [CLZ86] extend the traditional code motion algorithm to handle control structures within loops. We motivate the need for their extended algorithms in the paragraphs that follow.

Extended algorithms. A loop-invariant statement is one that only uses variables defined outside of the loop. However, by the traditional algorithm for loop-invariant code motion, in order to safely move a loop-invariant expression outside of the loop, three conditions must hold:

1. the statement must be executed whenever the loop is entered,
2. any variables defined by the statement must not be defined in any other statements within the loop, and
3. all uses within the loop of a variable defined by a loop-invariant statement must not have any other definitions of the variable that reach them. [ASU86]

One deficiency with this technique is illustrated in the following example:

```
do $i = 1, 100$
    if *loop-invariant condition* then
        $x$ = *loop-invariant expression*
        ⋮
    else
        $x$ = *different loop-invariant expression*
        ⋮
    endif
    $y = x$
enddo
```

Since the conditions for execution of the assignments to x are loop-invariant, as is the value assigned to x, we know that x will have the same value on each iteration of the loop. Therefore, we can move the assignments to x, along with the conditionals guarding their execution, outside of the loop.

However, by considering each statement individually to determine if it is loop-invariant, we observe that a single assignment to x cannot be moved out of the loop since the use of x in the assignment $y = x$ can be reached by two different definitions (by rules 2 and 3 above). Also, if the assignments to x remain in the loop, the assignment to y must also remain in the loop. The problem is that the traditional

algorithm does not take into account the independence of the code in the two halves of an *if* when the condition is loop-invariant.

In the above example, we can move the *if-then-else* statement out of the loop if all statements following each guard are loop-invariant. If only some of them are loop-invariant, instead of moving the conditions out of the loop, we copy them outside of the loop and move only the loop-invariant statements with the copies. The conditions also remain within the loop guarding the statements that are not loop-invariant.

Another motivation for the extended algorithm is to permit moving code in groups of statements if the group of statements is interdependent but not dependent on the rest of the loop. For example, an entire inner loop may be loop-invariant within an outer loop, but without moving the loop control structure and the statements as a group, no single statement can be moved out of the outer loop.

Loop unswitching. A similar technique, *loop unswitching* [AC72], can be applied when a condition is loop-invariant, but the code guarded by the condition is not. An example using unswitching is given below.

```
do $i = 1, 10$
    if loop-invariant condition then
        $x = f(i)$
    else
        $x = g(i)$
    endif
enddo
```

Before unswitching.

```
if loop-invariant condition then
    do $i = 1, 10$
        $x = f(i)$
    enddo
else
    do $i = 1, 10$
        $x = g(i)$
    enddo
endif
```

After unswitching.

For an *if-then-else* clause within a loop, we make two copies of a loop, one guarded by the *if* condition with the condition and the code guarded by the *else* clause all removed. Similarly, the second copy of the loop is guarded by the *else* condition, and the *if* condition and the code guarded by the condition are removed from the loop.

3.3 Bounds checking

When call sites are inlined within a loop, a lot of optimization opportunities involving loop induction variables can arise. When calls appear in loops, the current problem size, a function of the loop induction variable, may be passed as a parameter. Then, using this information we can eliminate unnecessary conditionals to reduce control dependences and improve parallelism in the loop, as was our goal when eliminating unreachable and loop-invariant code.

In a loop with a call site where some function of the loop induction variable is passed as an actual parameter, there may be a test within the called procedure of the value of the parameter to insure that it is within the range of the array bounds or within some other suitable bounds (e.g., greater than 0). Because the bounds of the induction variable and variables whose values are based on the induction variable can be determined directly from the bounds of the condition for loop execution, tests involving such variables can often be eliminated, as in the following example:

```
do $k = 1, n - 1$
    /* before inlining, call $p(n - k, \ldots)$ appeared here */
    $t = n - k$
    if $(t \leq 0)$ then
        ⋮
    endif
    ⋮
enddo
```

Because the loop induction variable k ranges from 1 to $n - 1$, the value of t ranges from $n - 1$ on the first iteration, to $n - (n - 1) = 1$ on its final iteration. Thus, the

test for $t \leq 0$ will always evaluate to *false*. So, the test and the code guarded by the test can be eliminated.

A similar opportunity arises when the test only evaluates to *true* on the first or last iteration of the loop (or the first few or the last few). By peeling off the first or last iteration and removing the test and its accompanying code within the loop, control flow is simpler, possibly exposing parallelism within the loop. However, peeling iterations is not as valuable as removing the test completely, particularly when the number of iterations is small. In this case, the increased loop overhead associated with loop peeling may offset the improvement in parallelism.

Improving Dependence Analysis. Considering the range of the loop induction variable and alternate induction variables can also be useful in determining the independence of two array subscript expressions. The following example illustrates this idea:

```
do $k = 1, n - 1$
    do $j = k + 1, n$
        $t(j) = a(k, j)$
        do $i = 1, n - k$
            $a(i + k, j) = a(i + k, j) + t(j) * a(i + k, k)$
        enddo
    enddo
enddo
```

Knowing that j is always greater than k in an iteration of the outermost loop, we can determine that $a(i + k, k)$ will never reference the same location as $a(i + k, j)$ in any iteration of the j loop. Also, since $i \geq 1$, we can determine that $a(i + k, j)$ and $a(k, j)$ will never reference the same location on any iteration of the i loop. Thus, the j loop can be run in parallel.

Note that the triangular version of Banerjee's dependence test would also prove the independence of these subscript expressions [Ken86] [Ban88]. However, this test is not commonly performed in commercial compilers. [1]

[1]The only systems we know about that perform this test are PFC and the IBM VS Fortran 2.4 compiler.

Locating variables whose values are based on the induction variable. Because variables that are functions of the induction variable may themselves be induction variables, we can locate them using a variant of *induction variable elimination* [ASU86]. The algorithm requires that a particular induction variable's only definition within the loop be of the form $i_2 = \pm i_1 \pm c$, where i_1 is some predetermined induction variable, and c is some expression that is constant within the loop. With our example, this expression contains numerical constants and the loop upper bound.

Once these variables are located, we must determine their possible ranges. If we know that i is a loop induction variable ranging from *lb* to *ub*, and j is an induction variable expressed in terms of i, we can substitute *lb* for i in the expression for j's value to determine the lower bound of j. Similarly, we can substitute *ub* for i to get the upper bound of j. Note that if $-i$ appears in the expression for j's value, substituting *lb* for i instead gives the upper bound of j, and substituting *ub* gives the lower bound of j.

This is a simplification of the techniques *range analysis* and *range propagation* [Har77]. These techniques track the range of values for all variables in a program. Although much more precise, tracking ranges of all variables is too expensive for most practical compilers. Here, we have limited ourselves to only tracking ranges of induction variables because often their ranges are explicitly declared in the loop body. This knowledge has proven to be especially useful in the context of optimization after inline substitution.

A different optimization dealing with induction variable ranges. A further potential optimization involving the loop induction variable appeared in the inlined version of *linpackd.* Here is an excerpt of code:

```
do $k = 1, n - 1$
    $t = n - k + 1$
    $r = 1$
    do $i = 2, t$
        if *condition* then
            $r = i$
            ⋮
        endif
    enddo
    $l = r + k - 1$
enddo
```

Because r is either 1 or some possible value of the induction variable i, the value of r lies somewhere between 1 and $n - k + 1$. Thus, the value of l is at least k, and in fact, no greater than n. This knowledge was useful in parallelizing a loop in which l appeared in a subscript expression. However, it is not clear that this optimization is generally applicable.

3.4 Partial parallelism

A final property of loops after inlining is that they are sometimes very complex. With long, complex loops it is difficult to avoid dependences and other structures in the code that the dependence analyzer cannot handle. Even after applying the optimizations described above, a loop may contain dependences that make it inherently sequential. To parallelize such loops, it is necessary to locate parallel portions of the loop and *distribute* the loop among the parallel and sequential portions of the loop.

Loop distribution separates a loop into groups of statements, forming a distinct loop with a copy of the loop header for each group of statements. It may be performed as long as all statements involved in a cycle of dependences remain in the same loop [KKL$^+$81].

Many of the dependences remaining in the loop after inlining are on scalar variables. The technique commonly used to eliminate dependences on scalar variables is *scalar expansion* [KKL+81]. A scalar r is expanded in a loop with induction variable i by replacing accesses to r with accesses to an array element $r(i)$, where the new array r has length at least as great as the number of iterations of the loop. This optimization is always safe, but the compiler must insure that uses of the expanded scalar are translated to correspond to either the previous or the current iteration, whichever is appropriate. However, scalar expansion is not usually performed unless doing so allows the loop to be parallelized.

We suggest scalar expansion even when it does not automatically allow the loop to be parallelized. Then, the combination of scalar expansion and loop distribution may allow portions of the loop to run in parallel. Two such opportunities arose in our sample programs.

First of all, it may be possible to expand a scalar and calculate its value for all iterations in parallel, even if the rest of the loop is sequential. Calculating the scalar values sequentially and expanding them into array values for every iteration so that the rest of the loop can be run in parallel is another possibility. Both of these cases are illustrated in this example:

```
do $i = 2, ny$
    $yn = (i - 1.5) * hy$
    do $j = lb, ub$
        $r = 0.$
        if $(yn = y(j))$ $r = dy(j)$
        if $(yn > y(j-1)$ and $yn < y(j))$ then
            $r = (dy(j) - dy(j-1)) * (yn - y(j))$
            $y(j+1) = yn$
        endif
    enddo
    $d = d + r * hy$
enddo
```

For the variable yn, scalar expansion permits calculating all of the values for yn in parallel. However, the inner loop is inherently sequential. Thus, yn is an example of

scalar expansion to permit the scalar to be calculated in parallel. Now, since the inner loop is sequential, we must calculate values for the variable r sequentially. However, scalar expansion of r using an array element for each outer loop iteration allows the value of d to be calculated in parallel using a *sum reduction.*[2] Thus, expansion of r is an example of scalar expansion to permit use of the scalar in a parallel loop. Here is the parallelized version of the above loop:

```
doall $i = 2, ny$
      $yn(i) = (i - 1.5) * hy$
enddo
do $i = 2, ny$
      do $j = lb, ub$
         $r(i) = 0.$
         if $(yn(i) = y(j))$ $r(i) = dy(j)$
         if $(yn(i) > y(j-1)$ and $yn(i) < y(j))$ then
            $r(i) = (dy(j) - dy(j-1)) * (yn(i) - y(j))$
            $y(j+1) = yn(i)$
         endif
      enddo
enddo
doall $i = 2, ny$
      $d = d +$ sum_reduction$(r(i) * hy)$
enddo
```

This particular pair of optimizations should be applied with great care since they may carry with them a substantial overhead. The overhead of loop distribution arises from the cost of duplicating loop control structures combined with the overhead of parallelizing a loop. Thus, the number of loop iterations and the number of statements appearing in the distributed loops must be sufficiently large to justify the increased overhead. We also observed poor cache performance after scalar expansion because calculating all values for a scalar in parallel greatly separated definitions of the variable from its uses. We give examples of slowdown after loop distribution and scalar expansion in the next section.

[2]A reduction operation uses special hardware to accumulate the results of certain operations applied across an array of values, even though there is a dependence. To compute the same result as if run in scalar, reduction operations should be commutative and associative.

4 Experimental Results

The optimizations discussed in this paper came directly from a study of inline substitution [CHT90]. After observing that increased parallelism after inlining was not as great as expected, examination of loops in the inlined code revealed the properties explained in this paper.

The optimizations were applied by hand to the 8 programs from the inlining study. On 4 of the programs, these optimizations yielded improvements in parallelism. After optimization, we executed versions of the inlined program with and without optimization on the Stardent Titan. The code was instrumented to measure the time spent in the optimized portions of the programs. The results are summarized in Figure 1 and discussed in the remainder of this section.

4.1 Explanation of Results.

In Figure 1, line 1 displays the number of loops where inlining may be applied to improve parallelism. Since we used a heuristic to determine when to inline a call site, not every call site appearing in a loop was inlined. In particular, calls to large procedures (> 175 lines) were not inlined, and calls where the types of actual parameters did not match the types of the called procedure's formals. Line 2 gives

	efie304	wave	dogleg	linpackd
1. DO loops with procedure calls	15	20	23	10
2. loops with no calls after inlining	14	15	22	8
3. loops improved after inlining	1	1	1	1
4. additional loops improved by opts	4	10	1	1*
5. execution time improvement in optimized loops	51%	77%	18%	-2%
6. execution time improvement for program	6%	4%	4%	-2%

* The loop from line 4 for *linpackd* is the same loop as in line 3.

FIGURE 1: Results of applying optimizations to inlined programs.

the number of loops from line 1 in which all calls were inlined. The calls remaining in loops in *wave* and *dogleg* either are to large procedures, or are calls where the types of parameters do not match. In *linpackd*, the 2 loops contain calls to a built-in timing routine, so they cannot be eliminated.

Line 3 shows the number of loops from line 2 that are fully or partially parallelized after inlining. This represents the improvement in optimization gained from inlining. Line 4 gives the number of additional loops from line 2 with partial parallelism after application of the optimizations described in this paper. In the case of *linpackd*, the loop from line 4 is the same loop as the one from line 3. That is, the loop partially parallelized by inlining is further parallelized by applying these optimizations.

Lines 5 and 6 represent the percentage decreases in execution time for the optimized portion of the code, and for the entire program, respectively. While the number of loops improved and execution time improvements within the optimized loops are significant, the overall execution time improvements are not as impressive. Comparing the inlined versions of these 3 programs before and after optimization, the overall execution time improvements have been no greater than 6%. With the exception of *linpackd*, this is because the time spent in the optimized loops is a very small percentage of the program execution time. This is probably because the Fortran programmers are concerned about the inefficiency of procedure calls, and therefore, avoid placement of calls in frequently executed parts of their programs.

4.2 Problems

Throughout these experiments, there were two problems that interfered with program performance after optimization. The first was the effect of scalar expansion on cache performance, and the second was the extensive use of loop distribution by the Titan compiler. We present these two problems, along with a detailed explanation of the results for *linpackd*, in the paragraphs that follow.

Cache performance after scalar expansion. After the combination of scalar expansion and loop distribution, executing a certain optimized loop in *wave* increased the execution time in the optimized portion of the code by a factor of 3. However, it was clear from previous experiments that the size of the loop and the number of iterations indicated that loop distribution and parallelization would be profitable. The next most obvious possibility for the dramatic increase in execution time was poor memory performance. We observed that the number of page faults did not change between the optimized and unoptimized versions of the program, so we then considered that cache performance might have been affected by our optimizations.

Apparently, the size of the expanded scalars combined with the distance between definition and use of the variables caused some or all of the variables to not be available in the cache at their uses. By rearranging statements so that definitions and uses were as close together as possible, we were able to achieve the 77% improvement presented in Figure 1.

Loop distribution. In section 3.4, we suggest loop distribution as a means of partially parallelizing a loop when full parallelism is not possible. Unfortunately, loop distribution may slow down the execution of the loop, particularly if the number of loop iterations is small or the loop bodies have few statements.

This problem is exacerbated by the code generation phase of the Titan compiler. During code generation, loops are distributed around the smallest number of statements that preserve all of the variable dependences. Then, after parallelization, these loops are fused back together whenever possible. However, loops containing conditionals are not fused with any other loops, even if fusing preserves all of the dependences [All90].

As we discuss in the next paragraph, unnecessary loop distribution combined with the loop distribution introduced by our optimizations resulted in performance degradation in *linpackd.* The same thing happened on *dogleg*, making execution time

in the optimized portion of the code 25% greater than before optimization. The optimized loop was distributed into 4 separate loops, 2 of which consisted of only a single statement. By making these 2 short loops run in vector mode rather than combined parallel and vector execution, we obtained the 18% improvement shown in Figure 1.

More on *linpackd*. First attempts to optimize *linpackd* resulted in performance degradations greater than 20%. During our optimization, we applied *loop unswitching* breaking the optimized loop into 2 loops. One of the loops was completely parallel, while the other was parallel if distributed into 2 loops. However, both loops contained a conditional, so both loops were distributed into 2 loops.

We then focused on preventing the parallel loop from being distributed. We observed that eliminating the conditional did not change the result calculated in the loop, it only potentially increased the amount of computation. By eliminating the conditional, the compiler no longer distributed the parallel loop, and from this change came the 2% performance degradation reported in Figure 1. Either the increased overhead associated with distributing the partially parallel loop or the increase in computation in the parallel loop caused the execution time increase.

5 Conclusion

This paper has presented optimizations that can be incorporated into a compiler supporting automatic or programmer-specified inline substitution. Because the optimizations are motivated by properties of inlined programs, they can enhance parallelism beyond what is possible with inlining alone.

The optimizations were validated with experimentation. We applied the optimizations by hand to 8 programs, and observed improvements in parallelism in 4 of these. However, the effects of loop distribution overhead and cache performance caused performance degradation in 3 of the programs. For the most part, we corrected

these problems, and the programs exhibited execution time improvements. However, a compiler that performs these optimizations should anticipate such problems and avoid them.

The experiments measured execution time improvements in the optimized portions of the programs, as well as in overall program execution time. The execution time improvements within the optimized portions of the programs were significant. However, the improvements in the overall program execution times were only moderate since time spent in optimized portions of most of the programs was only a small percentage of program execution time. This is perhaps due to a perception by the programmers that procedure calls are expensive.

Thus, while these optimizations were not as successful at improving program execution time as we might have liked, the improvements on just the optimized loops are an indication that significant execution time improvements may be possible on code written in a more modular style (i.e., making use of more procedures and procedure calls). Accordingly, if their compilers support optimizations across procedures such as inline substitution combined with the optimizations presented in this paper, programmers may feel more comfortable about using procedure calls in frequently executed portions of their programs.

References

[AC72] F.E. Allen and J. Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers.* Prentice-Hall, Englewood Cliffs, N.J., 1972.

[ACF+80] F.E. Allen, J.L. Carter, J. Fabri, J. Ferrante, W.H. Harrison, P.G. Loewner, and L.H. Trevillyan. The experimental compiling system. *IBM Journal of Research and Development*, 24(6), November 1980.

[All90] R. Allen. Private communication, March 1990.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass., 1986.

[Bal79] J.E. Ball. Predicting the effects of optimization on a procedure body. In *Proceedings of the SIGPLAN '79 Symposium on Compiler Construction.* ACM, August 1979.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, 1988.

[CHT90] K.D. Cooper, M.W. Hall, and L. Torczon. An experiment with inline substitution. Technical Report TR90-128, Rice University, 1990.

[CLZ86] R. Cytron, A. Lowry, and K. Zadeck. Code motion of control structures in high-level languages. In *Proceedings of the 13th Annual Symposium on Principles of Programming Languages.* ACM, January 1986.

[Har77] W. Harrison. Compiler analysis for the value ranges of variables. *IEEE Transactions on Software Engineering*, SE-3(5), May 1977.

[Hec77] M. Hecht. *Flow Analysis of Computer Programs.* American Elsevier, North Holland, 1977.

[Hus82] C.A. Huson. An inline subroutine expander for parafrase. Technical Report UIUCDCS-R-82-1118, University of Illinois, Urbana-Champaign, 1982.

[Ken86] K. Kennedy. Triangular banerjee inequality. Supercomputer Software Newsletter 8, Rice University, October 1986.

[KKL+81] D.J. Kuck, R. Kuhn, B. Leasure, D. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th Annual Symposium on Principles of Programming Languages.* ACM, January 1981.

[RG89] S. Richardson and M. Ganapathi. Interprocedural analysis versus procedure integraton. *Information Processing Letters*, 32(3), August 1989.

[Sch77] R. Scheifler. An analysis of inline substitution for a structured programming language. *CACM*, 20(9), September 1977.

[WZ89] M. Wegman and K. Zadeck. Constant propagation with conditional branches. Technical Report CS-89-36, IBM T.J. Watson Research Center, May 1989.