The Perils of Interprocedural Knowledge

Keith D. Cooper
Mary Hall
Linda Torczon

CRPC-TR90065
September, 1990

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

The Perils of Interprocedural Knowledge

Keith D. Cooper
Mary Hall
Linda Torczon

Department of Computer Science
Rice University
Houston, Texas 77251-1892

1. Introduction

In recent years, a large number of articles that deal with issues of interprocedural analysis and interprocedural optimizations have appeared in the literature.[1,2,4,6,7,9,11,12,14,15,16,17,18] Several of these articles have attempted to assess the practical value of interprocedural data-flow information or of specific cross-procedural transformations. We recently completed a study of the effectiveness of inline substitution in commercial FORTRAN optimizing compilers. During the course of the study, we came across an example that demonstrates quite clearly the intricate and interrelated problems that arise in the use of interprocedural techniques.

2. The Example

The famous Dongarra benchmark of numerical linear algebra operations, **linpackd**, was one of the eight programs used in our inlining study. As part of the study, we selected a set of call sites in **linpackd** for inlining and applied a source-to-source inliner to create a transformed version of the source. Next, we compiled and ran both versions of the code on several machines. In **linpackd**, we inlined forty-four percent of the call sites. This reduced the number of source language procedure calls by over ninety-nine percent. Thus, we eliminated effectively all of the procedure call overhead from the execution. Despite this, the running time on the MIPS M120/5 increased by eight and one-half percent after inlining.

We did not expect this behavior. Clearly, second-order effects in the compilation and optimization of **linpackd** overcame the reduction in procedure call overhead. Initially, we suspected that the problem was increased register pressure in the critical loop of the code—after all, **linpackd** spends the

Measurement	Original Source	After Inlining	Percent Change
loads	38,758,879	37,523,134	-3
stores	20,422,975	19,610,724	-4
calls	141,788	2,705	-98
nops	2,564,767	4,398,835	72
data interlocks	12,177,775	21,379,822	76
add interlocks	3,100	6,200	100
multiply interlocks	124,800	124,803	0
other fp interlocks	102	46,414	45503

Figure 1 — selected data from **pixstats**

```
      subroutine daxpy(n,da,dx,incx,dy,incy)
c
      double precision dx(1),dy(1),da
      integer i,incx,incy,ix,iy,m,mp1,n
      ...
      do 30 i = 1,n
        dy(i) = dy(i) + da*dx(i)
   30 continue
      return
      end
```

Figure 2 — relevant portions of **daxpy**

majority of its time inside a single loop. To investigate this behavior in more detail, we used the **pixstats** analyzer to look at detailed performance data. Figure 1 shows several of the important statistics.

Several things stand out in the performance data. First, both loads and stores decreased slightly. This suggests a decrease in register spills. Second, the number of call instructions executed dropped dramatically. Almost all the calls to source code routines went away; most of the remaining call instructions invoke run-time support routines. Third, the number of **nop** instructions executed nearly doubled. Finally, there was a significant rise in floating point interlocks. The ratio of interlocks to floating point operations rose from 0.62 to 1.1 after inlining.

Since **linpackd** executed almost twenty million floating point operations, the rise in interlocks was significant. The inlined code hit an additional nine million floating point data interlocks—a seventy-six percent increase. Interlocks on floating point adds doubled, to sixty two hundred. An additional forty six thousand other floating point interlocks occurred during execution of the inlined code.

Seeking to understand the increase in floating point interlocks, we looked more closely at the source code. Most of the floating point operations that occur in **linpackd** take place inside the routine **daxpy**. **Daxpy** is one of the BLAS routines, the basic linear algebra subroutines. It computes $y=ax+y$, for vectors x and y and scalar a. Thus, we began our search by examining the three call sites that invoke **daxpy**. The call to **daxpy** from inside **dgefa** passes two regions of the array **a** as actual parameters. The two actuals specify different starting locations inside **a** and a careful analysis shows that the two regions cannot overlap in any invocation of **daxpy**. Unfortunately, the level of analysis required to detect this separation is beyond what is typically performed in a compiler for a scalar machine.

Figure 2 shows an abstracted version of **daxpy** – the details that are relevant to **linpackd**'s performance. We have hidden the bulk of the code; it deals with the case where one or both of the strides, **incx** and **incy**, are not equal to one. After inlining the call site in **dgefa**, the critical loop takes on the following form.

```
      temp = n-k
      ...
      do 31 i = 1, temp
         a(i+k,j) = a(i+k,j) + t * a(i+k,k)
   31 continue
```

Now, the statement that comprises the loop body both reads and writes locations in the array **a**. Unless the compiler specifically inspects the subscripts with the intention of discovering this problem, it will be forced to generate code that allows the write of **a(i+k,j)** to clear memory before the read of **a(i+k,k)** for the next iteration can proceed. Few, if any, compilers for scalar machines invest compile time in this kind of subscript analysis.

To prove that these two array subscripts are disjoint requires some moderately sophisticated analysis. Typically, compilers for scalar machines have not performed a deep enough analysis of array subscript expressions to demonstrate independence in this case. In the absence of this kind of analysis, the compiler is forced to schedule the reads and writes in such a way that their executions cannot overlap. This prevents the compiler from masking any memory latency behind the floating point arithmetic, and increases the time required for each iteration.

Fortunately, the other two call sites pass unique actual parameters to each of the array positions. Thus, in the inlined version of the code, the inner loop reads and writes a single location in **b** and reads another location in **a**. Because the two references to **b** are textually identical, any moderately sophisticated compiler will recognize that they reference a single location. Thus, the compiler generates good code for these two loops—that is, code that is not artificially constrained by memory accesses. Unfortunately, these two call sites account for just 5174 of the calls to **daxpy**; the call site in **dgefa** accounts for 128,700 calls.

3. Interpretation

The question remains, why doesn't the same problem arise in the original code? Certainly, the sequence of array references made by the two versions of **linpackd** are identical. How can the compiler generate faster code for the original version of the program? The answer lies in the idiosyncrasies of the FORTRAN 77 standard.

Whenever a program can reference a single memory location using more than one variable name, those names are said to be aliases. Aliases can be introduced to a program in several ways. A call site can pass a single variable in multiple parameter positions. A call site can pass a global variable as a parameter. In languages that provide pointer variables, they can usually be manipulated to create multiple access paths to a single location.

The FORTRAN 77 standard allows the compiler to assume that no aliasing occurs at call sites. In the case of the call site in **dgefa**, no aliasing really occurs; the authors can argue that the program conforms to the standard's requirements. Common practice in the field is to assume that no aliases exist and generate code that may give an unexpected result if aliases do exist. This allows the compiler to support separate compilation and to avoid performing the kind of interprocedural analysis that would be required to detect aliases if they did exist.

Some compilers, like the VMS FORTRAN compiler, ignore the standard's restriction and compile code that will produce the expected results in the case when variables actually are aliased. In previous papers, we have suggested that the compiler should perform interprocedural alias analysis and use that information to provide the friendliness of the VMS FORTRAN compiler with the additional speed that results from understanding which parameters are not aliased.[8] These compilers assume that aliasing happens infrequently, so the cost of providing consistent behavior is small. In trying to understand the slowdown in **linpackd** after inlining on the MIPS, we naturally asked the question: could interprocedural alias analysis have helped the situation? The answer provides some interesting, albeit anecdotal, insight into the relative power of different types of analysis.

3.1. Conventional Alias Analysis

Conventional interprocedural alias analysis deals with arrays as homogeneous objects. A reference to any element of an array is treated as a reference to the whole array. Thus, a conventional analysis of **linpackd** would show that **dx** and **dy** can be aliases on entry to **daxpy**. Because of the flow-insensitive nature of the information, all that the compiler can assume is that there exists a path to **daxpy** that results in an invocation where **dx** and **dy** refer to the same base array. It does not assert that the path generating the alias is necessarily executable; neither does it assert that any references to **dx** and **dy** necessarily overlap in storage.

Given this aliasing information, a compiler following the "friendly" scheme suggested earlier would compile code that treated the two potentially aliased variables conservatively. Thus, it would compile a single copy of the body of **daxpy**, and that body would contain the code necessary to ensure that reads and writes to **dx** and **dy** in the loop bodies had sufficient time to clear through memory. The compiler would generate the slow code for all three call sites. This would simply slow down the 5,174 calls that ran quickly in the inlined version in our example.

3.2. Cloning

To regain the speed on the calls from **dgesl**, the compiler could generate two copies of **daxpy**'s body. If it examined the contribution that each call site made to the alias set for **daxpy**, it would determine that two of the calls involved no aliases while the third produced the alias between **dx** and **dy**. This information suggests compiling two copies of **daxpy** and connecting the call sites appropriately—a transformation known as *cloning*.[5]

With this strategy, the calls in **dgesl** would invoke a copy of **daxpy** that was compiled with the knowledge that no aliases occur. The call in **dgefa** would invoke a copy that assumed an alias between **dx** and **dy**. This strategy would produce roughly the same code that the MIPS compiler produced from the inlined version of **linpackd**.* Thus, interprocedural alias analysis coupled with cloning could get us back to the point where inlining got us. It would not, however, get back the cycles that we lost from the original code, compiled with the FORTRAN 77 standard's restriction on aliasing.

3.3. More Complex Analysis

In the original code for **linpackd**, the call site boundary between **dgefa** and **daxpy** serves two purposes. First, it provides the modularity intended by the designers of the BLAS routines.[10] Second, by virtue of the FORTRAN 77 standard's prohibition on aliasing, the call site acts as an assertion that all of the parameters at the call site occupy disjoint storage.

Introducing classical interprocedural aliasing information tells the compiler that the two parameters, **dx** and **dy**, may actually be aliases. Can the compiler, through deeper analysis, derive equivalent information that will allow it to conclude that no aliasing exists? To understand this issue, we will examine two possible techniques: regular section analysis and dependence analysis.

---

* The codes would differ in that the cloned version would have the overhead associated with the individual calls while the inlined version would avoid it. Furthermore, the codes might well differ in the code generated for the inner loops as a result of inlining—for example, the amount of register pressure seen in the inner loop in the inlined and cloned versions might differ substantially.

```
      subroutine dgefa(a,lda,n,ipvt,info)
      integer lda,n,ipvt(1),info
      double precision a(lda,1)
c
      ...
      nm1 = n - 1
      ...
      do 60 k = 1, nm1
         kp1 = k + 1
         ...
         do 30 j = kp1, n
            ...
            call daxpy(n-k,t,a(k+1,k),1,a(k+1,j),1)
30       continue
         ...
60    continue
      ...
      end
```

Figure 3 - abstracted code for **dgefa**

3.3.1. Regular Section Analysis

Classical interprocedural summary and alias analysis provides a superficial treatment of arrays. If any element of an array is modified, the analysis reports that the array has been modified. Similarly, if two disjoint subsections of an array are passed as arguments at the same call site, alias analysis will report that the corresponding formal parameters are potential aliases.

Regular section analysis provides more precise information about the portions of an array involved in some interprocedural effect.[3,11] In the case of side-effect information, the single bit representing modification or reference is replaced with a value taken from a finite lattice of reference patterns–the lattice of regular sections. To make this discussion more concrete, consider the regular sections actually produced for the call from **dgefa** to **daxpy** by the PFC system.[11] Figure 3 shows the context that surrounds the call site.

PFC computes two kinds of regular section information for the call, a MOD set that describes possible modifications to variables and a REF set that describes possible references to variables. The MOD set contains a single descriptor, **a[(k+1):n,j]**. This indicates that a call to **daxpy** may modify elements **k+1** through **n** of the **j**th column of **a**. The REF set contains two descriptors, **a[(k+1):n,j]** and **a[(k+1):n,k]**. These indicate that columns **j** and **k** can be referenced by **daxpy**, both in positions from **k+1** to **n**.

Given this information, could the compiler have determined that the two subranges of **a** are disjoint? To show independence, it needs to realize that **j** is always strictly greater than **k** and that **n** is smaller than the column length of **a**. In fact, both of these statements are true. While our example contains enough information to allow a compiler to intersect these regular sections fairly easily, the general case is more difficult. In general, computing the intersection of two regular sections is equivalent to applying an array subscript independence test. Thus, we hold out little hope that a regular section-based alias analysis could have solved our problem.

3.3.2. Dependence Analysis

To avoid the disastrous problems with interlocks introduced by the appearance of an alias in the inlined code, the compiler must show that the two sets of references in the critical loop are disjoint. In the previous subsection, we showed that this is equivalent to showing that the regular sections **a[(k+1):n,j]** and **a[(k+1):n,k]** don't intersect. This problem arises regularly in compilers that restructure programs for parallel or vector execution. Such compilers rely on a technique known as dependence analysis to show the independence of pairs of array references.[13]

The critical loop nest describes a triangular iteration space. To prove independence, the analyzer must perform some symbolic analysis and a triangular form of one of the dependence tests. While this sounds complex, a quick survey showed that KAP from Kuck and Associates, the Convex FORTRAN compiler, the Stardent FORTRAN compiler, and the PFC system from Rice all were able to prove independence in this case. Thus, the necessary analysis is clearly both understood and implementable.

With this kind of dependence analysis, the compiler could have generated code for the inlined version of **linpackd** that was as good as the code for the original program. Unfortunately, it appears that it will take this much work to undo the damage done by inlining the call from **dgefa** to **daxpy**.

3.4. Optimization Histories

A final option is available. Several colleagues have suggested that the compiler "remember" the original shape of the code—that is, that it mark source statements that result from inlining. Then, the compiler might be able to assert independence based on the implicit assertion represented by the original call site.

Unfortunately, this tactic is unlikely to be practical. A real implementation would require tagging pairs of references as independent to ensure that the information survived various forms of code motion. Then, each phase in the compiler would be taught to use such information. Because such information has no natural representation in the source code, this strategy effectively rules out a source-level inliner. Nonetheless, in an inliner that operated on some intermediate representation, this scheme could be used.

This scheme requires substantial work to address a relatively rare problem—one that arises from a FORTRAN idiosyncrasy. Furthermore, it does not help in the case that the programmer actually wrote the loop in this form, within a single procedure. It is probably simpler to implement the dependence analysis that would be required to generate good code for the loop, whether or not it arose as a result of inlining.

4. Conclusions

The **linpackd** benchmark is only a single program. Nonetheless, we felt that it shed light on several issues that arise in compiling a program in the presence of interprocedural knowledge. In the example, aliasing happens infrequently. Unfortunately, under some types of analysis, the program gives the appearance of a potential alias at one critical point—the call from **dgefa** to **daxpy** that accounts for the majority of the time spent in the entire program.

Simply adding interprocedural facts to the optimizer will not guarantee improved run-time performance. In our work in this area, we have discovered many cases where the profitable use of interprocedural information requires modifications to both the implementation and the philosophy of the optimizer. The **linpackd** benchmark is an excellent example of how giving the compiler a slightly different set of knowledge can have a major impact on the code that results.

5. References

(1) M. Burke. "An interval-based approach to exhaustive and incremental interprocedural data-flow analysis", *ACM TOPLAS* 12(3), July 1990.

(2) D. Callahan. "The program summary graph and flow sensitive interprocedural data flow analysis", *Proceedings SIGPLAN 88 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 23(7), July, 1988.

(3) D. Callahan and K. Kennedy. "Analysis of interprocedural side effects in a parallel programming environment", *Proceedings of the First International Conference on Supercomputing*, Athens, Greece, June, 1987.

(4) F.C. Chow. "Minimizing register usage penalty at procedure calls", *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 23(7), July 1988.

(5) K.D. Cooper. Interprocedural optimization in a programming environment. Ph.D. Thesis, Rice University Department of Computer Science, May, 1983.

(6) K.D. Cooper and K. Kennedy. "Interprocedural side-effect analysis in linear time", *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation.* SIGPLAN Notices, 23(7), July, 1988.

(7) K.D. Cooper and K. Kennedy. "Fast interprocedural alias analysis", *Proceedings of the Sixteenth POPL*, January, 1989.

(8) K.D. Cooper, K. Kennedy and L. Torczon. "The impact of interprocedural analysis and optimization in the $\mathbb{R}^n$ programming environment", *ACM TOPLAS* 8(4), October, 1986.

(9) J.W. Davidson and A.M. Holler. "A study of a C function inliner", *Software—Practice and Experience* 18(8), August 1988.

(10) J.J. Dongarra, J.R. Bunch, C.B. Moler, and G.W. Stewart. *LINPACK Users' Guide*, SIAM, Philadelphia. 1979.

(11) P. Havlak and K. Kennedy. Experience with interprocedural analysis of array side effects. *Proceedings of Supercomputing 90*, November, 1990. (also available as Technical Report 90-124, Department of Computer Science, Rice University).

(12) W.W. Hwu and P.P. Chang. "Inline function expansion for inlining C programs", *Proceedings of the SIGPLAN 89 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 24(7), July 1989.

(13) D.J. Kuck. *The Structure of Computers and Computations*, Vol. 1, Wiley, New York, 1978.

(14) S. Richardson and M. Ganapathi. "Interprocedural optimization: experimental results", *Software—Practice and Experience* 19(2), February 1989.

(15) S. Richardson and M. Ganapathi. "Interprocedural analysis versus procedure integration", *Information Processing Letters* 32, 24 August 1989.

(16) V. Santhanam and D. Odnert. "Register allocation across procedure and module boundaries", *Proceedings of the SIGPLAN 90 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 25(6), June 1990.

(17) P.A. Steenkiste and J.L. Hennessy. "A simple interprocedural register allocation algorithm and its effectiveness for LISP", *ACM TOPLAS* 11(1), January 1989.

(18) D.W. Wall. "Register windows versus register allocation", *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 23(7), July, 1988.