Unexpected Side Effects of Inline Substitution: A Case Study

Keith D. Cooper, Mary W. Hall, and Linda Torczon

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

The structure of a program can encode implicit information that changes both the shape and speed of the generated code. Interprocedural transformations like inlining often discard such information; using interprocedural data-flow information as a basis for optimization can have the same effect.

In the course of a study on inline substitution with commercial FORTRAN compilers, we encountered unexpected performance problems in one of the programs. This paper describes the specific problem that we encountered, explores its origins, and examines the ability of several analytical techniques to help the compiler avoid similar problems.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors — compilers, optimization

Additional Keywords and Phrases: inline substitution, interprocedural analysis, interprocedural optimization

The final version of this paper was published in *ACM Letters on Programming Languages and Systems*, 1(1), March, 1992, pages 22-32.

Unexpected Side Effects of Inline Substitution: A Case Study

Keith D. Cooper, Mary W. Hall, and Linda Torczon

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

The structure of a program can encode implicit information that changes both the shape and speed of the generated code. Interprocedural transformations like inlining often discard such information; using interprocedural data-flow information as a basis for optimization can have the same effect.

In the course of a study on inline substitution with commercial FORTRAN compilers, we encountered unexpected performance problems in one of the programs. This paper describes the specific problem that we encountered, explores its origins, and examines the ability of several analytical techniques to help the compiler avoid similar problems.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors — compilers, optimization

Additional Keywords and Phrases: inline substitution, interprocedural analysis, interprocedural optimization

1 Introduction

In recent years, many articles dealing with issues of interprocedural analysis and interprocedural optimization have appeared in the literature [1, 2, 5, 6, 7, 10, 12, 14, 17, 18, 19, 20, 21]. Several of these articles have attempted to assess the practical value of interprocedural data-flow information or of specific cross-procedural transformations. Ganapathi and Richardson point out that inline substitution can be viewed as an upper limit on the improvement available through use of interprocedural data-flow information [17]. Put succinctly, their point is that inlining splits off copies of nodes along some path through the call graph. This, in turn, allows interprocedural data-flow analysis to derive a sharper image of the program's real behavior because it eliminates points of confluence — places where the data-flow analyzer must merge sets from different paths. At the merge points, the analyzer can only assume the set of facts that occur along all entering paths. This set is often weaker than the individual sets that enter the merge.

We recently completed a study of the effectiveness of inline substitution in commercial FORTRAN optimizing compilers [9]. During the course of the study, we came across an example that demonstrates the kind of problems that can arise in the use of interprocedural transformations like inlining. Similar problems will arise in a compiler that bases optimization on the results of interprocedural data-flow analysis. Our experience suggests that compilers that use interprocedural transformations or interprocedural data-flow information will require stronger intraprocedural analysis than would be needed otherwise.

2 The Example

The famous Dongarra benchmark of numerical linear algebra operations, `linpackd`, was one of the eight programs used in our inlining study. `Linpackd` consists of ten routines containing four hundred seventeen non-comment lines of code. Figure 1 shows its call graph. Each arrow represents one or more call sites.

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 the original and inlined versions of the code on several machines. We inlined forty-four percent of the call sites, measured statically. This reduced the number of dynamically executed calls by roughly ninety-eight percent. This includes over ninety-nine percent of the calls that correspond to subroutine calls in the source code; most of the calls that remained in the executable invoke run-time support routines. Thus, we eliminated almost all

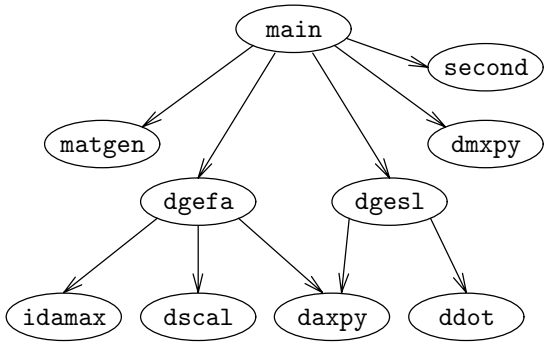

Figure 1 call graph for `linpackd`

of the procedure call overhead that occurred in the original program. Despite this, the running time on the MIPS R2000 increased by eight and one-half percent after inlining. We did not expect this behavior.[1]

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 majority of its time inside a single loop. To investigate this behavior in more detail, we used `pixie` and `pixstats` — the instruction level execution analysis tools provided with the MIPS machine — to look at detailed performance data. Figure 2 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`s executed nearly doubled. Finally, there was a significant rise in the number of floating-point interlocks — stall cycles introduced by the hardware to allow completion of a pipelined floating-point operation. The ratio of interlocks to floating-point operations rose from 0.62 to 1.1 after inlining. Nearly all these interlocks were data interlocks — stalls on loads and stores.

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

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 2 Selected data from `pixie`

[1] A common perception is that increased executable size often leads to increased page faults or instruction cache faults. We assumed that this would not be significant with `linpackd` because of its size. The inlined code had 2.37 times as many source code statements, but the executable produced by the MIPS compiler was nine percent smaller than the original executable. Both our study and Holler's study suggest that inlining rarely leads to thrashing or instruction cache overflow [9, 13].

```
      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 3 abstracted code for `daxpy`

other floating-point interlocks occurred during execution of the inlined code. All of these numbers suggest that the instruction scheduler was less effective on the inlined program than on the original program. In the original, it was able to mask these idle cycles with useful computation. The additional `nop` instructions are a symptom of the same problem.

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 part of the BLAS library, the basic linear algebra subroutine library. 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`. One is in `dgefa`; there are two in `dgesl`. The call in `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 3 shows an abstracted version of `daxpy` — the details that are relevant to `linpackd`'s performance. 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
```

The more complex subscript expressions arise from inlining; the array references must be expressed in terms of variables known in the calling procedure.

Now, the single statement in the loop body both reads and writes locations in the array `A`. Unless the compiler can prove that the subscripts are always disjoint, 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. This requires moderately sophisticated subscript analysis — deeper analysis than most scalar compilers perform. Without such analysis, the compiler is forced to schedule the loads and stores of `A` so that their executions cannot overlap. This limits the freedom of the scheduler to such a degree that it can no longer hide memory and pipeline latency. The result is increased time for each iteration of the loop.

The program `linpackd` contains two other calls to `daxpy`. They both occur inside the routine `dgesl`. Both of them pass unique arguments to `daxpy`'s parameters `dx` and `dy`. Thus, inlining them does not introduce the same problems that occur at the call from `dgefa`. The compiler can generate good code for the post-inlining code — that is, code that is not artificially constrained by memory accesses. Unfortunately, these two call sites account for just 5,174 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. A program that generates an alias is not standard conforming; the behavior of the resulting code is not defined. In practice, aliases do occur, often as a by-product of FORTRAN's lack of dynamic allocation. Programmers declare work arrays high in the call graph and then pass them down through the call graph. The remaining aliases fall into two categories: (1) carefully considered uses where the programmer understands that no sharing occurs, and (2) unintended sharing. The call to `daxpy` from `dgefa` falls in the first category. No aliasing really occurs; the authors can argue that the program conforms to the standard's requirements.

Common practice in the field is to rely on the standard's restriction. This lets the compiler assume that no aliases exist and generate code that runs faster but produces unexpected results if aliases do exist. For example, IBM's VS FORTRAN and FORTRAN H compilers, and the FORTRAN compilers from Cray Research, MIPS Computer, and Stardent Computer all do this. Other compilers, like DEC's VMS FORTRAN compiler and our own research compiler, ignore the standard's restriction and compile code that will produce the expected results in the case when variables actually are aliased. This results in more predictable behavior when aliasing does occur.

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]. Assuming that aliasing happens infrequently, the cost of providing consistent behavior in this way is small. The Convex Application Compiler performs alias analysis and has a compile-time flag that lets the user dictate how to handle the issue.

In trying to understand the slowdown in `linpackd` described in Section 2, we asked ourselves 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.

This discussion has implications for languages other than FORTRAN. The standard's prohibition on aliasing allows the compiler to generate good code for procedures that access both call-by-reference formal parameters and global variables. Without the restriction, the compiler would be forced to generate slower code. In the example, this introduced many wasted cycles in the form of interlocks and `nop`s. Thus, compilers for languages that use call-by-reference for array and structure variables could profit from knowledge about aliasing that involves their formal parameters.

3.1 Classical Alias Analysis

Classical interprocedural alias analysis deals with arrays as homogeneous objects [8]. A reference to any element of an array is treated as a reference to the whole array. Thus, a classical 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 implementing the scheme suggested earlier — computing interprocedural information and using it to determine where conservative code is needed — would generate the slower code for all the executions of `daxpy`. Thus, it would create a single copy of `daxpy`. That instance 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. 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* [8].

With this strategy, the calls in `dgesl` would invoke a copy of `daxpy` that was compiled with the assumption 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`.[2] 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 [11]. 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.

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

```
      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 4 abstracted code for `dgefa`

[2]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.

disjoint subsections of an array are passed as arguments at the same call site, classical alias analysis will report that the corresponding formal parameters are potential aliases.

Regular section analysis is a technique that provides more precise information about the portions of an array involved in some interprocedural effect [4, 12]. 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 [12]. Figure 4 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`. Both of these statements are true. A compiler that performed interprocedural regular section analysis and incorporated a strong test for intersecting regular sections could determine that the two subscript expressions never take on the same value.

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 [15]. Dependence analysis, as discussed here, is an intraprocedural technique that can be applied to the inlined program. In our example, the loop nest that causes problems is wholly contained in a single procedure.

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, in the fall of 1990, 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, the compiler should mark source statements that result from inlining. Using these marks, the compiler could assert independence of specific references based on the implicit assertion represented by the original call site.

Several commercial compilers use this strategy. In particular, both the Cray and Convex compilers retain enough information to remember that the original program asserted independence for its parameters. Unfortunately, this tactic may have relatively limited application. The more general solution, implementing the necessary dependence analysis, will both cure the idiosyncratic FORTRAN problem and allow the compiler to generate good code for other languages where call-by-reference parameter passing of aggregate structures introduces potential aliases.

4 Conclusions

In this paper, we presented a problem that arose from inlining in a FORTRAN program. We examined several analytical techniques to determine if they could have given the compiler enough information to

allow it to avoid the problem. Three of the techniques would have worked: regular section alias analysis coupled with an appropriate intersection test, dependence analysis, and keeping appropriate information in an optimization history. Of the three, dependence analysis may be the most attractive. While regular sections and optimization histories would have solved the problem we encountered, dependence analysis has more general utility. It serves as a basis for many other optimizations [3, 16]. Furthermore, implementing the intersection test for regular sections is equivalent to implementing a dependence test.

The `linpackd` benchmark is only a single program. Nonetheless, we feel that it sheds light on several issues that arise when interprocedural transformations are used. The problem that we encountered relied on an idiosyncracy in the FORTRAN standard. In practice, the structure of a program encodes important information in subtle ways. For example, inlining a call site often takes a complex name (*e.g.*, an array element or a base-offset pair for a global variable in some data area) and forward substitutes it into the procedure body. If the formal parameter is a scalar variable (a single register name), this replaces the inexpensive name with the more complex name, increasing the competition for registers and, perhaps, replicating some computation. As another example, a call site constitutes the broadest possible hint to the register allocator — it tells the allocator to break all caller-saved live ranges and spill them. Any transformation that eliminates the call site, like inlining, takes that hint away from the allocator.

A compiler that employs interprocedural transformations may need more powerful analysis methods, like regular section analysis, dependence analysis, or optimization histories, to ensure that the compiler does not generate worse code than it would have produced for the original program, and to ensure that the compiler can capitalize fully on the opportunities exposed by the interprocedural technique.

5 Acknowledgements

We owe our colleagues in the ParaScope group at Rice a debt of thanks for building the tools in ParaScope that were used in the underlying study. We are grateful to MIPS for including the `pixie` and `pixstats` tools in their standard software distribution. More manufacturers should provide this kind of tool to their users. The comments and insights provided by the three reviewers and the editor led to substantial improvements in both the exposition and contents of this paper.

References

[1] M. Burke, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. Technical Report RC 15968 (70983), IBM Research Division, July 1990.

[2] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 23(7), pages 47–56. ACM, July 1988.

[3] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. *SIGPLAN Notices*, 25(6):53–65, June 1990. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[4] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5:334–358, Aug. 1987.

[5] F. C. Chow. Minimizing register use penalty at procedure call. *SIGPLAN Notices*, 23(7):85–94, July 1988. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[6] K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Languages Design and Implementation,* SIGPLAN Notices23(7), pages 57–66. ACM, July 1988.

[7] K. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49–59. ACM, Jan. 1989.

[8] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the $\mathbf{R}^n$ environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, Oct. 1986.

[9] K. D. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software – Practice and Experience*, 21(6):581–601, June 1991.

[10] J. W. Davidson and A. M. Holler. A study of a C function inliner. *Software – Practice and Experience*, 18(8):775–790, Aug. 1988.

[11] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. LINPACK *User's Guide*. SIAM, 1979.

[12] P. Havlak and K. Kennedy. Experience with interprocedural analysis of array side effects. In *Proceedings of Supercomputing 90*, pages 952–961. IEEE Computer Society Press, Nov. 1990.

[13] A. M. Holler. *A Study of the Effects of Subprogram Inlining*. PhD thesis, Univ. of Virginia, Charlottesville, VA, May 1991.

[14] W. W. Hwu and P. P. Chang. Inline function expansion for inlining C programs. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 24(7), pages 246–257. ACM, July 1989.

[15] D. J. Kuck. *The Structure of Computers and Computations*, volume 1. Wiley, 1978.

[16] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN Notices*, 23(7):318–328, July 1988. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[17] S. Richardson and M. Ganapathi. Interprocedural analysis versus procedure integration. *Information Processing Letters*, 32(3):137–142, Aug. 1989.

[18] S. Richardson and M. Ganapathi. Interprocedural optimization: Experimental results. *Software—Practice and Experience*, 19(2):149–169, Feb. 1989.

[19] V. Santhanam and D. Odnert. Register allocation across procedure and module boundaries. *SIGPLAN Notices*, 25(6):28–39, June 1990. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*.

[20] P. A. Steenkiste and J. L. Hennessy. A simple interprocdural register allocation algorithm and its effective for lisp. *ACM TOPLAS*, 11(1):1–32, Jan. 1989.

[21] D. Wall. Register windows vs. register allocation. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 23(7), pages 67–78. ACM, July 1988.