Pointer Target Tracking - An Empirical Study

Jon Loeliger, Robert Metzger, Mark Seligman, Sean Stroud

CONVEX Computer Corporation

Abstract

CONVEX Computer Corp. has developed a language-independent interprocedural optimizer that is now available to users of its C-series supercomputers. This paper documents the benefits of one particular feature of that optimizer, pointer target tracking.

This paper surveys the structure of the interprocedural optimizer and the goals of pointer target tracking. It describes the realities of scientific codes that had to be handled, and gives an overview of the algorithms used. It also provides detailed statistics on the opportunities for pointer tracking, the characteristics of pointer ranges, and the benefits to optimization of pointer tracking.

Introduction

CONVEX Computer Corp. has developed a language-independent interprocedural optimizer, which is now available to users of its C-series supercomputers. This optimizer is packaged together with existing CONVEX compilers in a product called the Application Compiler.

Currently, the Application Compiler processes programs containing FORTRAN and C source code. The FORTRAN front end accepts ANSI Standard source (ANSI X3.9-1978). It provides a high degree of compatibility with the extensions made by DEC or Cray, at the user's option. The C front end accepts ANSI Standard source (ANSI X3.159-1989). It also provides optional compatibility with common usage C.

The driver program for the Application Compiler determines which source files need to be (re)compiled, and applies the appropriate front end to all such source files. The front end performs lexical analysis, parsing, and semantic analysis. It writes to disk the symbol table, annotated parse tree, and miscellaneous information for each procedure.

After all source files that need to be (re)compiled have been processed, the driver invokes the interprocedural optimizer. The following analyses, which answer the questions shown, are performed as listed.

Interprocedural Call Analysis -
Which procedures are invoked by each call?
Interprocedural Alias Analysis -
Which names refer to the same location?
Interprocedural Pointer Tracking -
Which pointers point to which locations?
Interprocedural Scalar Analysis -
Which procedures (and subordinates) use and assign which scalars?
Interprocedural Constant Propagation -
Which globals and formals are constant on entry to which procedures?
Inline Analysis -
Which procedures should be inlined at which call sites?
Interprocedural Array Analysis -
Which procedures (and subordinates) use and assign which sections of arrays?

Each analysis algorithm reads from the program database and writes the information it generates back to that database.

After the interprocedural algorithms have completed, the driver program invokes the CONVEX common back end for each procedure in the application. The back end consists of a machine-independent optimizer and a machine-dependent code generator. In the Application Compiler, the optimizer has been modified to make use of information gathered by interprocedural analysis, rather than make worst-case assumptions about procedure side-effects.

The optimizer first performs the following procedure-scope scalar optimizations:

Constant propagation and folding
Redundant-assignment and -use elimination
Useless code elimination
Common subexpression elimination
Invariant code motion
Operator strength reduction
Induction value optimization

Then it performs automatic vectorization and parallelization:

Dependency analysis
Loop Distribution
Loop Interchange
Partial vectorization
Conditional vectorization
Parallelization

The code generator is table-driven and supports several similar architectures. The CONVEX C2 instruction set is an extension of the C1 instruction set, and there are other differences such as instruction timings.

The code generator performs the following machine dependent transformations:

Instruction selection
Register allocation
Instruction scheduling
Peephole optimization
Object generation

After the backend has processed all procedures, the Application Compiler driver invokes the linker, which creates an executable image from the generated objects.

Applying tracked pointers

Languages which permit liberal use of pointers, such as C and C++, pose special problems for writers of optimizing compilers. Because the language syntax does not specify, at a given use, where a pointer may point, the compiler writer often assumes the worst case: a pointer may point anywhere. Such "worst case" assumptions may severely restrict the amount of optimization that can be done.

In the absence of semantic information, a compiler must assume that a pointer may target any addressed variable in the program. When compiling an individual function, without knowledge of the rest of the program, this pessimistic estimate must include not only variables known to have been addressed, but also all globals: a global's address may be taken in another function and assigned to a pointer. With minimal interprocedural analysis this burden can be lightened somewhat: all addressed globals will be known, perhaps shrinking a pointer's set of possible targets.

These simple syntactical observations, however, do not provide a sufficient basis for performing aggressive automatic optimization. If the compiler could determine, for example, that a pointer was not aliased to an assignment made through a cast, it would know that the pointer and its target were of the same data type; this would relax the worst case assumption to all addressed variables of that data type. If, similarly, the compiler could determine that all assignments reaching a use of a pointer were from addressed variables, the less pessimistic assumption would then limit the range to just those variables. Pointer tracking uses just such information to limit the range of aliases a pointer may have.

The approach to pointer tracking presented in this paper is effective primarily for applications that traverse blocks of memory using pointers. Such blocks may be allocated statically or on the stack as arrays, or dynamically on the heap. Using pointers to process blocks of homogeneous numeric data is the distinguishing characteristic of scientific and engineering applications written in C. This style of programming differs greatly from system software written in C, which typically process recursive data structures such as lists, trees, and graphs.

Precise alias lists provide greater opportunity for optimizing loops which process memory blocks. If a loop contains two pointer dereferences, for example, the compiler no longer assumes that the two potentially alias the same memory location. Aliases cause the compiler to assume there are *loop-carried dependencies*, and when it can eliminate such dependencies, it can vectorize and parallelize the loop.

Realities of scientific C codes

Real applications used by scientists and engineers present a number of challenges and problems to a compiler designer. This section describes some of the realities of scientific codes that make implementing pointer tracking more interesting.

There are three aspects that C that make it less than ideal for scientific computing. [M89] The lack of *float complex* and *double complex* data types is the first problem. The most common solution is to define a structure that contains the real and imaginary parts.

The semantics of argument passing, which prevent compilers from assuming that pointer arguments aren't aliases (as in FORTRAN), is the second problem. Without being able to make such assumptions, C compilers must be very conservative in creating aliases between pointer variables. This inevitably results in poor automatic vectorization and parallelization, in the absence of assistance from the user in the form of pragmas or command line options. This second problem was one of the main motivations for implementing interprocedural pointer tracking. Pointer tracking can determine whether two argument pointers could ever be aliases, and if they can not be, it makes much more precise alias lists.

The third problem is in many ways the most serious — the lack of a way to declare argument arrays that have varying dimensions. FORTRAN has adjustable

and assumed-size arrays, and Pascal has conformant arrays. Legend has it that this feature was omitted from the original design of C because it would have created an incompatibility with the language BCPL. How short-sighted!

The problem is stated very well in [PFTV88] (p. 17): "The systems programmer rarely deals with two-dimensional arrays, and almost never deals with two-dimensional arrays whose size is variable and known only at run time. Such arrays are, however, the bread and butter of scientific computing. Imagine trying to live with a matrix inversion routine which could work with only one size of matrix!"

The inevitable result of a significant omission from a programming language is a lot of ugly hackery. C programmers have coped with the omission of varying-dimension arrays in several ways. These include explicit indexing, arrays of pointers, and array objects.

The first approach stores all multi-dimensional arrays as vectors. A macro like the following is defined to generate the address calculation code.

```
double *x;

#define SUB(i,j,c)  (i)*(c)+(j)

*(x + SUB(i, j, m)) = 0.0;
```

The second approach uses vectors of pointers to represent arrays with more than one dimension. First, the vector of pointers is allocated, then the vectors that contain actual data are allocated, as shown below.

```
double **x;
x = (double **) malloc(
     N * sizeof(double *));
x[i] = (double *) malloc(
     N * sizeof(double));

x[i][j] = 0.0;
```

The obvious advantage of the second approach over the first is that it uses the existing syntax of the language for element references. It has several disadvantages.

1) Multiple indirections introduce extra memory references that reduce performance with many compilers and architectures.

2) Extra storage is required for the pointers, and this can become quite substantial when dealing with arrays of more than two dimensions.

3) Assigning pointers into a vector whose own size is likely to be indeterminate at compile-time presents significant difficulties for a pointer tracking algorithm.

The third approach is to use structures to represent array objects of different types. The structure contains the dimensions of the array, and a pointer to the vector containing the data.

```
typedef struct DBLMAT {
  int rows, cols;
  double *data;
} x;
x.cols = N;
x.data = (double *) malloc(
     N*N * sizeof(double));

#define REF(x,i,j) x.data[j+i*x.cols]

REF(x, i, j) = 0.0;
```

There are two advantages of the third approach over the first:

1) Writing an element reference requires less effort, reducing the likelihood of error.

2) The compiler automatically generates the code to pass and use the column length field.

Both the first and third approaches are equally amenable to the pointer tracking algorithm presented in this paper. The second approach can obscure the flow of pointers in the general case.

Description of the algorithm

Procedural pointer tracking

Pointer tracking is performed after the scalar optimizer has converted the input procedure into a graph representation and has performed data-flow analysis on it. Each basic block is represented as a directed acyclic graph. Within basic blocks, *dataflow* arcs connect operators to operands. *Use-definition* arcs constrain the order in which memory references occur. Basic block DAG's are the nodes in a control flow graph. Within the control flow graph, arcs show possible control flow between blocks and identify loop structures. The control flow graph is reducible and contains use-definition information. Such information tells, for a given use of a variable, what definitions of that variable have a path that reaches the use.

Targets for use and assignment nodes are kept in a data structure called a *range*. A pointer range is a set of symbols and a set of flags. The symbols are the names of variables whose addresses are contained in the pointer at some time. Special symbols are created during interprocedural analysis for heap storage. The flags indicate whether the range must be treated as a worst-case target (anywhere) or whether the range may be

affected by the behavior of a procedure call.

The algorithm for tracking pointers in a procedure is essentially the same both before and after interprocedural (IPO) analysis has taken place. The differences between the two schemes concern the ranges assigned to formal and global pointers and how ranges are revised by a call.

When a pointer is assigned from an expression involving an addressed symbol, the assignment's range is set to that symbol. Assignments from expressions involving a pointer receive the range of that pointer. A use of a pointer receives as its range the union of the ranges of all assignments reaching it.

Assignments made to a pointer from other expressions, such as from an array or a pointer dereference are not tracked; they are identified as default instances. Similarly, pointer uses that are reached by function calls are identified as being affected by a call. In the absence of interprocedural information these ranges are assigned the default worst case. If inter-procedural tracking has taken place, though, there may be useful knowledge of the effect of the call on the pointer's range.

In either tracking pass, each node in the graph is visited. Nodes representing assignments to or uses of unsubscripted pointers have their range information set as described above. If loops are present in the graph, the nodes within each loop must be visited multiple times to enable targets discovered in the loop to propagate to the loop head. Because the graph is reducible, the algorithm visits each nested loop (loop depth + 1)-many times [ASU86].

As the first tracking pass begins there is no knowledge of the ranges of either global or formal pointer variables. Uses of these pointers which can be reached from outside the procedure are therefore assigned the default (worst-case) range. At the start of the post-IPO tracking pass, however, the ranges for such pointers are initialized using information gleaned from the interprocedural pointer tracking. Post-IPO tracking passes, likewise, use range information for individual calls. When tracking is complete, unsubscripted uses and assignments of pointers are annotated with range information.

For the purposes of dependence analysis, pointer references are represented internally in the compiler as subscripted references to a mythical array that represents all of memory. For example `*p = *q;` is represented as `$MEM[*p] = $MEM[*q];` Another pass over the graph replaces each pointer dereference by a dereference of a new variable corresponding to its range.

The compiler creates new symbols to represent the various combinations of aliasing encountered in the ranges. For example, if some range is found to target two distinct variables, a new symbol is made with those two variables on its alias list. Clearly, the number of combinations encountered may be large. The compiler computes default symbols for those cases whose aliases are all addressed variables of the pointer's type or, if the pointer has been assigned the default (worst-case) range, uses the symbol aliasing all addressed variables.

In the previous example, if the ranges of p and q can be determined to be completely distinct, the representation will change to `$MEM1[*p] = $MEM2[*q];` When the dependence analyzer sees two array references with different names, and no aliases between them, it does not bother inspecting the subscripts since it knows there can be no dependence. This results in improved vectorization and parallelization.

On reentering the optimizer after interprocedural analysis, moreover, use-definition information is recomputed using the new alias variables. Because pointer dereferences tend to have fewer aliases now, assignments through a pointer tend to generate fewer new definitions. This results in longer use-definition chains, which makes possible:

- better redundant-use and redundant-assignment elimination,
- basic-block DAG's with fewer constraints,
- longer life for values in registers.

Interprocedural pointer tracking

The interprocedural pointer tracking algorithm synthesizes new target range sets using the initial pointer ranges developed by the procedural pointer tracking algorithm. For each function complete range sets are determined for every global or formal pointer variable referenced directly by the function or by any function it calls directly or indirectly.

The first phase of the interprocedural algorithm converts all the ranges to refer to symbols in the program symbol table, rather than the individual function symbol tables. It also creates ranges for statically initialized pointer variables that are associated with the main procedure.

The second phase of the interprocedural algorithm inspects call sites and adds items to the ranges of argument pointer variables based upon the actual arguments used. This phase also creates, for each function, a list of pointer variables and functions that return pointers whose ranges must be resolved.

The third phase converts all of the pointer ranges associated with function calls and returns into a single directed graph. Nodes represent pointers and pointees, arcs represent a binding between a pointer and an object pointed at, or between formal and actual arguments.

The heart of the algorithm computes the transitive closure of this reference graph. Once computed, the resolved pointer ranges are constructed, and associated with two sets — ranges known on entry to each function, and ranges known on exit from each function.

The results of this algorithm have limitations. In particular, the range for a global variable on entry to a function is shown as the union of all ranges associated with that variable during the execution of *all* functions. Originally this was implemented as an iterative algorithm that gave exact results for global variables at each call site. It was abandoned because it was far too slow to process real applications. The resulting degradation in optimization has been minimal, since most codes do not use global pointers in vectorizable loops. The current implementation does not handle more complex memory tracking due to either array-of-pointer references or multiple pointer dereferences.

Empirical studies

Assumptions

The application codes analyzed for this paper were obtained from public domain sources and from CONVEX customers. The size of our published sample was limited by the fact that there is much less scientific C code available than scientific FORTRAN code, and by privacy requirements of our customers.

To help isolate the effects of interprocedural pointer tracking, all other interprocedural optimizations were disabled when the data in this paper were generated. In practice, all of the interprocedural algorithms would be enabled and the interprocedural pointer tracking would only be a part of a much larger optimization effort.

As the LINALG code is a library, we cannot supply statistics on it directly because there is no complete application. Instead, we compiled 9 separate example drivers that used the library. We extracted from the library only those routines which were needed to compile each of the examples independently. The figures we present for this application are a mean of these 9 example applications. Statistics are shown for only those routines of the library which are used somewhere in one of the examples. There is some duplication of the lower level routines from the library.

Two versions of the TRACE application are shown in the data. The TRACE1 application has its own implementation of a standard library memory allocation scheme, directly providing the three routines *malloc()*, *realloc()*, and *free()*. The TRACE2 version uses the memory management routines from the standard library.

Applications

Table 1 provides general characteristics of the applications on which the effects of interprocedural pointer tracking was studied.

The first column of Table 1 gives an identifier for the application and the type of the application. The second column lists the total number of functions in the application. The third column gives the total number of source lines in the application. The last two columns list the mean and maximum number of function calls per procedure.

The codes range from 5 to 173 functions, with a median of 23 functions. They range from 411 to 11442 source lines, with a median of 931 lines. The maximum number of calls per function ranges from 16 to 194. This high number is largely attributable to the fact that in C input and output are done through calls to functions in libraries, rather than through features that are a part of the language.

Opportunities for pointer tracking

The next two tables provide background information on the opportunities for pointer tracking. Table 2 shows potential pointer target data while Table 3 shows data about the pointers themselves.

The first column of Table 2 identifies the application. The second, third, and fourth columns give the total, mean and maximum number of local arrays within a function. The fifth column shows the total number of global arrays within the entire application. The final three columns characterize an additional target for pointer tracking, showing the total, mean and maximum number of calls to heap allocation routines.

The first column of Table 3 identifies the application. The following four columns list the mean and maximum number of local and argument pointer variables per procedure. The last column gives the number of global pointer variables in the application.

The total number of local arrays ranges from 29 to 356, with a median of 45. The maximum number of local arrays declared in any one function ranges from 8 to 99, with a median of 24. The total number of global arrays ranges from 1 to 160, with a median of 5. The total number of calls to library functions that allocate storage from the heap ranged from 0 to 135, with a median of 10.

The maximum number of local pointer variables in any one function ranges from 1.9 to 22, with a median of 10. The maximum number of argument pointer variables in any single function ranges from 4 to 13, with a median of 4. The total number of global pointer variables ranges from 0 to 45, with a median of 8.

Pointer tracking effectiveness

The next two tables combined demonstrate the effectiveness of the pointer tracking algorithm by contrasting the same statistics obtained with and without interprocedural pointer tracking.

Table 4 shows the effectiveness of determining to which memory locations a pointer variable refers. The first column identifies the application. The second column gives the total number of dereferences in the application. The third and fourth columns give the mean and maximum number of dereferences per procedure. The fifth column lists the total number of aliases present at all dereferences. The next two columns list the mean and maximum aliases present per dereference. The final column lists the number of dereferences that had exact ranges, e.g. those whose target sets did not include the canonical worst-case target (anywhere).

Table 5 shows the same analysis using interprocedural pointer tracking.

During the normal procedural compilation of a function, only the symbols named within a file are potential aliases of a dereference. Using interprocedural information causes the compiler to represent explicitly program-wide aliases for each dereference.

Generally, an application has more global variables than are named in any one source file of the application. In many cases, a procedure compiler will assume that a pointer is an alias for all the global variables that it knows about. When interprocedural information is introduced, there are more global variables known, and potentially more global variables for which a pointer can be an alias. In these applications interprocedural pointer tracking may result in adverse optimization effects. If the interprocedural pointer tracking algorithm isn't good enough, the transformation to program-wide aliases list for each dereference will introduce more aliases than would otherwise be present in the normal procedural compilation. This may cause a more constrained basic block, and fewer opportunities for vectorization.

The BANDED application is an example where interprocedural pointer tracking failed to produce shorter alias lists. The procedural compiler must only contend with an average of 6.2 aliases per dereference, while the interprocedural pointer tracking algorithm has caused an average of 14.8 aliases per dereference. The effects of this increase are apparent by noting that there are now 386 dereferences instead of the original 374. Due to more constrained basic blocks, fewer redundant references were eliminated.

Using the standard memory manager enabled the Application Compiler to increase the number of exact pointer ranges from the 615 in the TRACE1 application to 967 in the TRACE2 application. This resulted in reducing the number of dereferences by 10 and the average aliases per dereference from 29.7 to 25.3. The increase in the total dereference aliases in these applications does not have a negative effect on optimization. On the contrary, scalar optimization was able to reduce the number of dereferences by 19 in TRACE1 and 16 in TRACE2. This shows that pointer tracking is not just a question of reducing aliases, but of putting them where they really belong. If key assumed aliases are removed by pointer tracking, optimization improves despite the increase in actual aliases on other dereferences.

The best case occurs when the interprocedural pointer tracking algorithm creates fewer aliases at each dereference despite having been transformed into program-wide alias lists. The applications SOLVE, SPEECH, LINALG, SURFACE, and CRYPTO have fewer dereferences in the entire application due to better basic block optimizations. These applications and SIMPLEX all show a significant decrease in the average and maximum number of aliases present at dereferences.

Resulting optimizations

Table 6 shows the result of using pointer tracking. The first column identifies the application. The next two columns indicate how many loops were fully or partially vectorized by the most recent release of the CONVEX C compiler, when no command line options or source directives are used to guide vectorization. The following two columns indicate how many loops were fully or partially vectorized by the Application Compiler. The last two columns show the theoretical maximum — how many loops could be vectorized by manually inserting pragmas and making trivial changes to the source code.

In all cases there was an improvement. It varied from the minimal (one loop partially vectorized) to the theoretical maximum achievable by manual editing of the source. In half of the applications, the additional benefit of pointer tracking enabled the compiler to fully vectorize all the loops that were vectorizable.

Future work

Multi-level dereferences

We are currently investigating extending our initial implementation to handle multi-level pointer dereferences. We plan to create a single pointer range for vectors of pointers, which receives targets from any of the elements of that array. This will enable to us to determine whether there are common targets between two

different arrays of pointers. It will not solve the more general problem of determining whether there are common targets between two elements of the same array of pointers. We doubt that this problem can be solved at compile-time.

Runtime pointer checking

Pointer Tracking causes a larger number of loops to be automatically vectorized and parallelized than before. Even though it is a definite improvement, it still does not cause all potential vectorizable loops to be vectorized. The particular coding style of an application can have a dramatic effect on the number of loops vectorized. In order to obtain the same level of vectorization for C applications as is already available for FORTRAN applications, we intend to implement runtime pointer checking.

As a last resort, the Application Compiler will use runtime pointer checking to select potentially vector or parallel loops. Two versions of every potentially vector or parallel loop are created. One contains vector or parallel code, the other scalar code. At runtime, one of the pair of loops is selected, based on tests applied to each of the pointer variables used in the loop. The tests determine whether any pair of pointer references (where one reference is an assign), can point to the same location on different iterations of the loop. If not, the vector or parallel version of the loop may be used.

This approach is a last resort because it can add significant overhead to the loop execution. If simplifications can't be made at compile time, the runtime test may include several arithmetic and comparison operations *for each pair of pointers*. These calculations must be recomputed each time the loop preheader is executed.

Related Work

Our work proceeds in the spirit of [W80], [C86], [ASU86] and [G88]. Aho et al. [ASU86] suggests a special pass over the control-flow graph to summarize pointer uses. We feel it is easier to employ the use-definition information we have already computed. We compute pointer targets at individual uses, as Guarna does [G88], but we do not use tree-matching algorithms which, the author concedes, may be quite expensive. We currently track pointers of multiple indirection *type*, for example "***char"; we can associate ranges with such pointers, although not to dereferences of them. See the section on Future Work above. Weihl [W80] discusses tracking the actual-formal binding of pointer parameters; we track this as well as the behavior of pointer globals in a function call.

Recent work on tracking linked data structures in heap storage is described in [LH88], [HPR89] and [CWZ90]. Horwitz et al. [HPR89] give semantic characterizations of data dependencies and uses these to develop algorithms to compute safe approximations of dependencies when heap-allocated storage is manipulated. We have avoided this type of analysis, concentrating instead on tracking pointers to arrays, records and scalars, which we feel comprise a significant portion of the pointer manipulations found in numerical software. We do attempt to identify pointers referencing heap-allocated storage, however, in order to track references to heap-based arrays. The assumptions made by Chase et al. [CWZ90] demonstrate the difference between analyzing pointers for Lisp-like languages and for the C language. They "do not allow the unrestricted pointer arithmetic that is possible in languages such as C, and we do not deal with aliasing between variables." We are primarily interested in pointer arithmetic code, with the blocks of array-like homogeneous data implied in such situations, and our main goal is to reduce aliasing between variables.

[AJ88] and [GGL90] discuss problems involved in attempting to parallelize C programs. Allen and Johnson [AJ88] point out that some aliasing problems associated with actual-formal binding of pointer parameters can be avoided by function inlining. Gannon et al [GGL90] give examples of codes that need an analysis of pointer usage to guarantee parallelizability.

Conclusions

Scientists and engineers are increasingly choosing C as a language for application development. Such applications need the same high performance as those coded in FORTRAN. Interprocedural pointer tracking and runtime pointer tracking can enable optimizing C compilers to provide the same level of performance that FORTRAN programmers have previously enjoyed on high-performance computers. If C programmers follow a few simple stylistic suggestions, they can increase the likelihood that an optimizing C compiler will produce an efficient executable.

Use standard library functions for memory allocation.

A compiler that performs interprocedural pointer tracking will know that certain functions allocate and deallocate heap storage. If a code uses its own allocator, the compiler may not be able to determine this.

Explicitly declare functions that return pointers.

Pointer tracking needs to collect information about all carriers of addresses, both variables and function

returns. If a function appears to return an integer (the default), the compiler may not collect all the information it needs prior to interprocedural pointer tracking.

Avoid global pointers to storage used in vector or parallel loops.

Iterative solutions to the pointer tracking problem can determine exact pointer ranges for global variables at any point in the program. Our experience, however, is that such solutions are far too expensive for compiling real applications. If a loop can be vectorized or parallelized, it should use argument or local pointers.

Represent multi-dimensional arrays without multiple indirections.

This paper includes three approaches to multi-dimensional arrays, two of which work well with our pointer tracking algorithm. Regardless of pointer tracking, some compiler/hardware combinations will give better performance with explicit address arithmetic, and others will do better with the extra indirections introduced by vectors of vectors. Choosing an approach to this problem depends on the target hardware for the application.

Acknowledgements

Randall Mercer developed the initial design for the interprocedural optimizer. The final design and implementation of the interprocedural optimizer were done by the authors and Matthew Diaz.

Presley Smith, Frank Marshall, and Steve Wallach believed we could make it work and gave us the time and resources to do so.

References

[ASU86] Aho, A., Sethi, R., Ullman, J. *Compiler: Principles, Techniques, and Tools.* Addison-Wesley, 1986, pp. 648-660.

[AJ88] Allen, R., and Johnson, S. *Compiling C for Vectorization, Parallelization, and Inline Expansion. In* Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation, July 1989, pp. 241-249.

[CWZ90] Chase, D., Wegman, M., and Zadeck, F. Analysis of Pointers and Structures. In *Processing of SIGPLAN '90 Conference on Programming Language Design and Implementation,* June 1990. pp. 296-310.

[C86] Coutant, D. Retargetable High-Level Alias Analysis. in *ACM Symposium on Principles of Programming Languages,* January, 1986, pp. 110-118.

[GGL90] Gannon, D., Guarna, V., and Lee, J. Static Analysis and Runtime Support for Parallel Execution of C in *Languages and Compilers for Parallel Computing,* MIT Press, 1990.

[G88] Guarna, V. A Technique for Analyzing Pointer and Structure References in Parallel Restructuring Compilers. *Proceedings of International Conference on Parallel Processing - 1988* pp. 212-220, August 1988.

[HPR89] Horwitz, S., Pfeiffer, P., and Reps, T. Dependence analysis for pointer variables. *Proceedings of SIGPLAN '89 Conference on Programming Language Design and Implementation,* June 1988, pp. 28-40.

[LH88] Larus, J., and Hilfinger, P. Detecting conflicts between structure accesses. *Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation,* July 1989, pp. 21-34.

[L90] Larus, J. Parallelism in Numeric and Symbolic Programs. Technical Report issued by the U. of Wisconsin - Madison.

[M89] Metzger, R. Using C for Supercomputing. *Proceedings of Software Development '89,* pp 107-124.

[PFTV88] Press, W., Flannery, B., Teuklosky, S., Vettering, W. *Numerical Recipes in C.* Cambridge University Press, 1988.

[W80] Weihl, E. Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables. *Seventh Annual ACM Symposium on Principles of Programming Languages,* 1980, pp. 83-94.

Table 1
General Application Characteristics

Application	Total Functions	Total Source Lines	Calls per Function avg	Calls per Function max
BANDED: Banded equation solver	22	864	7.2	42
SOLVE: Various linear system solvers	8	590	8.9	45
SIMPLEX: Simplex linear programming	9	931	10.6	51
FITCOEF: Fit coefficients	5	411	7.2	16
TRACE1: Ray tracing	173	11442	3.7	74
TRACE2: Ray tracing, modified	170	11090	3.8	74
SPEECH: Speech analysis	99	3774	7.8	194
LINALG: Linear algebra library	23.4	814	19.8	52.9
SURFACE: 2-D graphics	16	1725	9.6	72
CRYPTO: Applied number theory	23	3311	5.9	35

Table 2
Opportunities for Pointer Target Tracking

Application	Total Local Arrays	Local Arrays per Function avg	Local Arrays per Function max	Total Global Arrays	Total Allo-cations	Allocations per Function avg	Allocations per Function max
BANDED	38	1.7	27	1	18	0.8	18
SOLVE	24	30	17	2	10	1.3	5
SIMPLEX	43	4.8	34	1	25	2.8	19
FITCOEF	29	5.8	15	1	0	0	0
TRACE1	122	0.7	21	52	0	0	0
TRACE2	122	0.7	21	50	135	0.8	39
SPEECH	356	3.6	99	160	8	0.1	4
LINALG	40.9	1.9	35.3	0.9	2.7	0.1	1.0
SURFACE	45	2.8	24	5	0	0.0	0
CRYPTO	47	2.0	8	11	17	0.7	5

Table 3
Opportunities for Pointer Tracking Pointees

Application	Local ptr vars avg	Local ptr vars max	Argument ptr vars avg	Argument ptr vars max	Global ptr vars
BANDED	1.3	4	1.5	4	20
SOLVE	1.9	10	1.8	3	0
SIMPLEX	1.8	10	2.2	13	8
FITCOEF	1	5	1.6	2	0
TRACE1	0.9	22	1.9	8	38
TRACE2	0.9	22	1.9	8	38
SPEECH	0.8	6	0.4	2	46
LINALG	0.3	1.9	2.0	3.8	0
SURFACE	3.9	17	2.7	5	0
CRYPTO	1.8	9	1.6	4	4

Table 4
Characteristics of Dereferences Without Tracking

Application	Total Number of Dereferences	Dereferences per Function		Total Dereference Aliases	Aliases per Dereference		Exact Pointer Ranges
		avg	max		avg	max	
BANDED	374	17	119	2313	6.2	60	0
SOLVE	164	20.5	62	2005	12.2	26	0
SIMPLEX	196	21.8	40	6163	31.4	50	0
FITCOEF	23	4.6	10	216	9.4	26	0
TRACE1	3011	17.4	184	61943	20.6	79	0
TRACE2	2998	17.6	184	61836	20.6	79	0
SPEECH	143	1.4	9	30613	214.1	293	0
LINALG	172.2	7.1	23.9	1340	6.3	8.0	0
SURFACE	874	54.6	152	12343	14.1	49	0
CRYPTO	949	41.3	100	18504	19.5	45	0

Table 5
Characteristics of Dereferences With Tracking

Application	Total Number of Dereferences	Dereferences per Function		Total Dereference Aliases	Aliases per Dereference		Exact Pointer Ranges
		avg	max		avg	max	
BANDED	386	17.5	126	5723	14.8	45	126
SOLVE	157	19.6	62	1146	7.3	25	32
SIMPLEX	196	21.8	40	1913	9.8	49	28
FITCOEF	25	5.0	12	283	11.3	24	17
TRACE1	2992	17.3	184	88886	29.7	57	615
TRACE2	2982	17.5	184	75415	25.3	57	967
SPEECH	136	1.4	9	4669	34.3	158	69
LINALG	165.2	6.8	26.8	1300	7.5	14.9	129.1
SURFACE	832	52.0	152	4198	5.0	43	479
CRYPTO	301	13.1	67	2451	8.1	23	108

Table 6
Loop Optimizations Resulting from Pointer Target Tracking

Application	CONVEX C 4.1		Application Compiler		Manual Modification	
	Total Full vector	Total Partial vector	Total Full vector	Total Partial vector	Total Full vector	Total Partial vector
BANDED	1	0	1	1	9	3
SOLVE	3	1	5	1	15	1
SIMPLEX	2	0	16	0	16	1
FITCOEF	4	0	6	1	6	1
TRACE1	1	0	4	0	10	4
TRACE2	1	0	6	2	10	4
SPEECH	38	14	46	14	46	14
LINALG	10.3	0.2	14.6	1.2	NA	NA
SURFACE	6	0	7	1	7	1
CRYPTO	6	4	17	0	17	0