Goal-Directed Interprocedural Optimization

Preston Briggs
Keith D. Cooper
Mary W. Hall
Linda Torczon

CRPC-TR90102
November, 1990

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

Goal-Directed Interprocedural Optimization

Preston Briggs
Keith D. Cooper
Mary W. Hall
Linda Torczon

Rice University
Houston, TX 77251-1892

1 Introduction

Interprocedural optimization is not a new idea. In the early sixties, the ALPHA system looked across call sites to find opportunities for generating customized procedure linkages [Yer66]. Since that time, work in this area has generally fallen into two distinct categories: application of interprocedural transformations and computation of interprocedural data-flow information. The former attempts to improve code quality directly by specializing procedures to the calling context; the latter attempts to improve code quality indirectly by improving the quality of data-flow information at procedure entry and exit, and around call sites. Both schools of thought rely on an implicit assumption that the interprocedural work improves the final code.

Attempts to verify this experimentally have shown mixed results. Scheiffler found small improvements from inline substitution in CLU [Sch77]; Davidson and Holler found consistent small improvements using inline substitution in C programs [DH88]. Ganapàthi and Richardson found that their Pascal compiler required drastically longer compile times to achieve modest improvements with inlining – compile times increased by up to five hundred percent [RG89]. Our own study of inline substitution in commercial FORTRAN optimizing compilers found that secondary effects in the compiler often obscured *any* benefits from inlining [CHT90]. At best, these studies have shown modest improvements. In some cases, compile times have grown so rapidly that they make the techniques impractical.

We are building a compiler that performs interprocedural data-flow analysis and applies interprocedural transformations. We have spent several years looking for heuristics that allow the optimizer to determine where to apply these techniques profitably. Originally, our focus was on estimating execution frequencies for call sites and the improvements that accrue from each decision. The underlying notion was that execution time could be decreased by the accumulation of small improvements at individual call sites. We feel that the studies in this area have demonstrated two fundamental points: (1) the execution-time improvements are, at best, modest, and (2) secondary effects in the optimizer can negate these improvements. This paper proposes a new way of looking at the problem – a *goal-directed* approach.

Our strategy is to use interprocedural transformations to enable the application of *high-payoff* optimizations – optimizations that routinely improve the program by integer factors. In this paper, we describe an experiment using interprocedural techniques to create opportunities for cache and register blocking [CCK90]. We feel, however, that the strategy can be extended to other high-payoff optimizations, like automatic detection of parallelism.

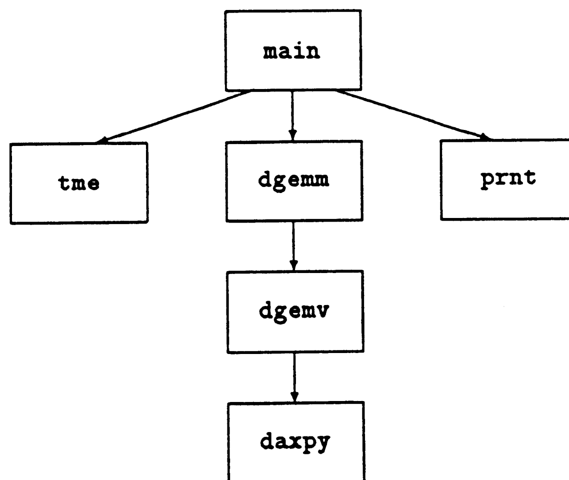
2 The Experiment

The strategy that we advocate in this paper does not resemble our early ideas. Instead, it is the end product of a somewhat circuitous trip. Early in 1990, one of our colleagues gave a talk at Stanford describing the use of *strip mining* to improve cache blocking, *scalar replacement* to place array values in registers, and *unroll and jam* to adjust both register pressure and loop balance [CCK90]. A researcher at SUN Microsystems became interested in these techniques and their effectiveness on the SPARC microprocessor.[1] We sent him a simple matrix-matrix multiply, both the original and the transformed version. The transformations produced an improvement on the SPARC of almost a factor of two. Rather than satisfying his curiousity, however, this simply whetted his appetite.

A short while later, our colleague at SUN asked us to apply the same techniques to the well-known SPEC benchmark `matrix300`. By applying a combination of interprocedural transformations and the memory-management transformations, we were able to achieve significant improvements on two popular workstations. The success of the experiment suggests a promising approach to interprocedural optimization. In the next three sections, we introduce `matrix300`, motivate and explain the transformations, and present measurements illustrating the performance improvements.

2.1 Matrix300

The call graph of `matrix300` is shown below.



[1] SPARC is a trademark of SUN Microsystems, Inc. MIPS is a trademark of MIPS Computer Systems.

The bulk of the computation is carried out in dgemm and its components.

- dgemm computes a matrix-matrix product, using dgemv.
- dgemv computes a matrix-vector product, using daxpy.
- daxpy computes $\vec{y} \leftarrow \vec{y} + \alpha\vec{x}$.

All of the floating-point computations are actually performed in daxpy. Its form is very simple:[2]

```
subroutine daxpy(m, x, A, ia, Y, iy)
dimension A(ia, m), Y(iy, m)
do i = 1, m
   Y(1, i) = Y(1, i) + x * A(1, i)
enddo
```

The unusual form of the subscripts is simply a notational convenience, allowing the original authors a handy way of expressing access to non-unit stride vectors.

2.2 Improving Matrix300

Examining the code for daxpy, we notice several things:

- The loop body contains two loads, one store, a floating-point add, and a floating-point multiply.
- The loads will probably cause a large number of cache misses, especially if ia and iy are greater than one.

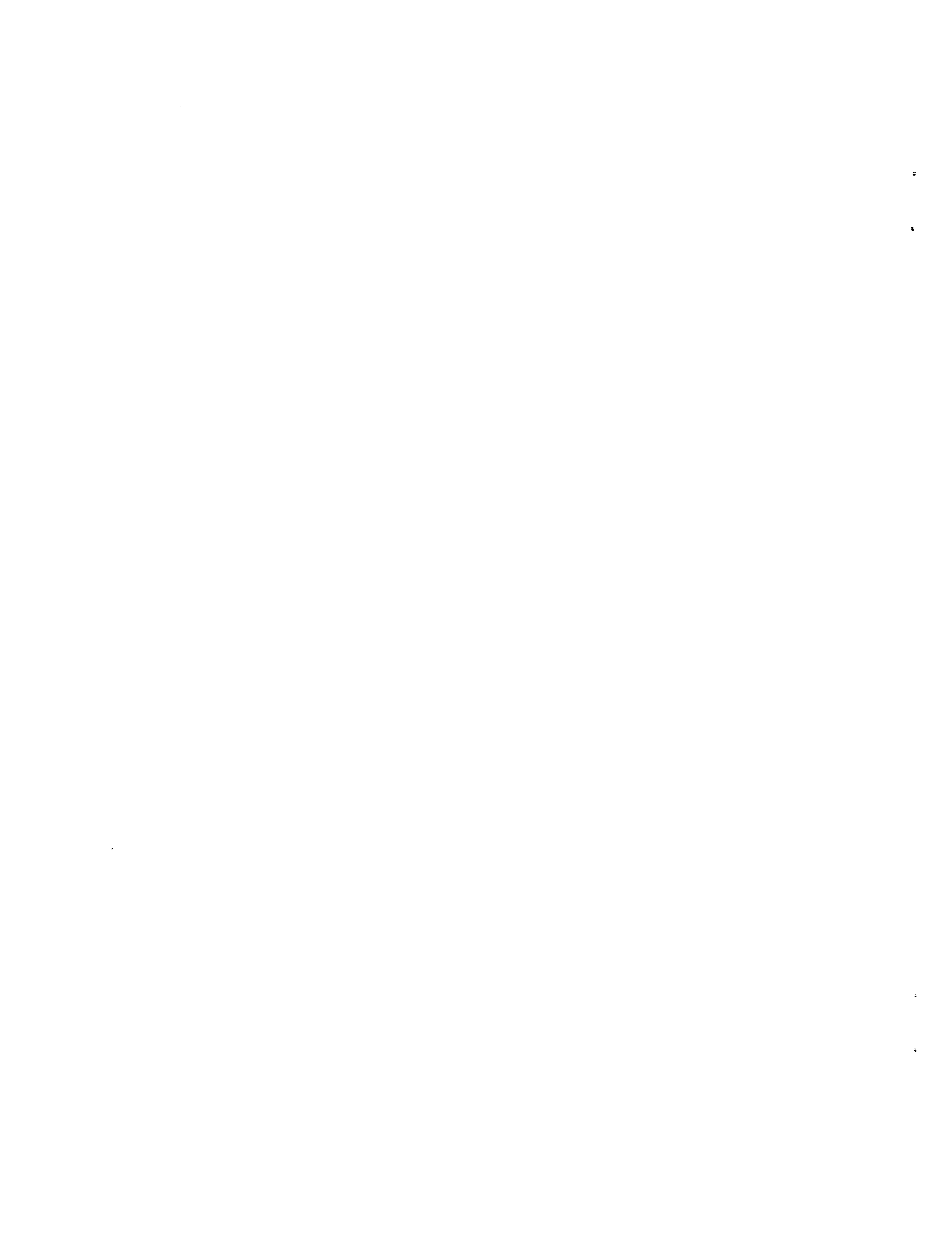
Obviously, memory accesses are an important contributor to the overall running time of the benchmark. The key to avoiding memory accesses is recognizing *reuse*. When values are reused many times, they can be kept in registers, thus avoiding the need for repeatedly fetching them from memory. Techniques for exploiting reuse include *scalar replacement* and *unroll and jam* [CCK87, CCK90].

Unfortunately, daxpy offers no opportunities for scalar replacement; that is, there is no reuse of any of the array locations and therefore no profit in allocating array elements to registers. Further, the absence of reuse suggests that cache will be ineffective. Unroll and jam is often suitable for exposing reuse; however, unroll and jam requires nested loops and daxpy contains only a single loop.

Examining dgemv, we notice that the call to daxpy is contained inside a loop. This suggests that inlining daxpy would create a doubly-nested loop, suitable for transforming with unroll and jam.

```
subroutine dgemv(m, n, A, ia, X, ix, Y, iy, job)
dimension A(ia, n), X(ix, n), Y(iy, n)
— code setting ii and ij
do j = 1, n
   k = 1 + (j - 1) * ij
   call daxpy(m, X(1, j), A(k, 1), ii, Y(1, 1), iy)
enddo
```

[2]Note that some variables have been renamed for clarity. Further, the BLAS routines were renamed to emphasize that all floating-point computations are double-precision.



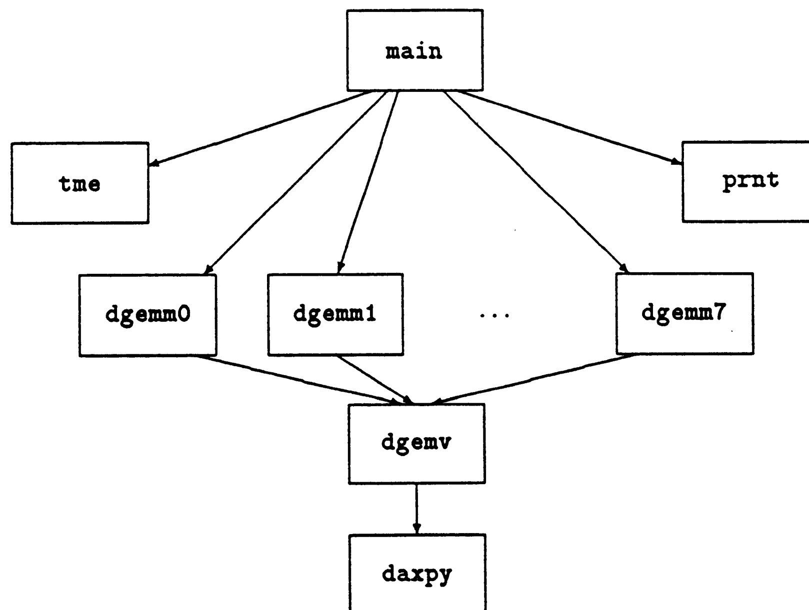
Inlining daxpy involves an ugly array reshape of A. Difficulty arises because of the use of ii in the leading dimension of A's declaration in daxpy. If the compiler were to automatically perform the translation, the resulting reference to A(1, i) in daxpy would be A(1+(j-1)*ij+(i-1)*ii, 1). This subscript expression is complex enough to defy the dependence analysis on which scalar replacement and unroll and jam rely.

Hoping to simplify, we examine the code that sets ii and ij:

```
if ((iabs(job) - 1) / 2) = 0 then
   ii = ia
   ij = 1
else
   ii = 1
   ij = ia
endif
```

Determining the value of job is difficult. Examining dgemm shows that the value of job passed into dgemv depends on the value of a parameter called jtrpos. Further investigation shows that each of the eight call sites in the main program binds a different *constant* value to jtrpos when calling dgemm. This situation is amenable to *procedure cloning.*

Cloning a procedure involves creating copies of the procedure and assigning different calls to different copies. In this case, we clone dgemm to improve our information about the value of jtrpos. To clone, we create eight copies of dgemm. In each copy, jtrpos is removed from the parameter list and reintroduced as a variable initialized to the appropriate constant value. Additionally, each of the call sites in the main program must be updated to call the correct clone and pass the correct number of parameters. The resulting call graph is illustrated below:



Performing constant propagation within each clone determines the value of job at each call to dgemv. Unfortunately, in four of the clones, dgemv is called with a value of 1 for job; in the other four clones, job is

bound to 3. Again, we can use procedure cloning – this time two copies of `dgemv` are created (`dgemv1` and `dgemv3`), with `job` initialized appropriately.

Performing constant propagation on the clones of `dgemv` determines the value of `ii` and `ij` and allows `daxpy` to be inlined cleanly. With some simplification, we arrive at the following loop nest for `dgemv1`:

```
do j = 1, n
   do i = 1, m
      Y(1, i) = Y(1, i) + X(1, j) * A(i, j)
   enddo
enddo
```

For `dgemv3`, the results are similar:

```
do j = 1, n
   do i = 1, m
      Y(1, i) = Y(1, i) + X(1, j) * A(j, i)
   enddo
enddo
```

At this point we are finally in a position to make use of the transformations described in [CCK90]. We perform *loop interchange, loop fusion,*[3] *unroll and jam,* and *scalar promotion.* The results of the transformations on `dgemv3` are outlined below:

```
do i = 1, m, 10
   y0 = zero
   y1 = zero
   ...
   y9 = zero
   do j = 1, n
      x0 = X(1, j)
      y0 = y0 + x0 * A(j, i+0)
      y1 = y1 + x0 * A(j, i+1)
      ...
      y9 = y9 + x0 * A(j, i+9)
   enddo
   Y(1, i+0) = y0
   Y(1, i+1) = y1
   ...
   Y(1, i+9) = y9
enddo
```

2.3 Results

The table below summarizes the results of our experiment. In addition to the relevant timings, we have included the object code sizes for both versions of the program.

	compiler	*original*		*modified*		*execution*
machine	*options*	(bytes)	(seconds)	(bytes)	(seconds)	*improvement*
Sparcstation 1	-O4	122,880	466	122,880	229	2.0×
MIPS M-120	-O3	114,464	755	112,976	229	3.3×

[3] Fusion is performed with an earlier loop that initializes the first row of `Y` to zero.

The improvements on each machine are quite significant. There are several sources of improvement:

Memory In the original code, there were mn fetches from `A`, n fetches from `X`, mn fetches from `Y`, and $mn+m$ stores into `Y`. The improved code avoids much memory traffic by holding values in registers for reuse (the variables `y0`, ..., `y9`, and `x0`). Overall, **dgemv3** requires mn fetches from `A`, $mn/10$ fetches from `X`, and m stores into `Y`; there are *no* fetches from `Y`. Briefly, the improvements avoid nearly 100% of the stores and 50% of the loads. In `matrix300`, we save about 216 million stores and nearly as many loads.[4]

Scheduling Close examination of the code in **dgemv3** shows that each of the multiply-accumulate statements can potentially be executed in parallel. Both compilers take advantage of this freedom when scheduling. The original code in **daxpy** is less amenable to scheduling improvements, even after unrolling, because of the store to `Y`. Without dependence analysis, most optimizers will not recognize the *lack* of dependence between successive stores and loads to `Y`.

Inlining and unrolling The improved code has fewer procedure calls and less loop overhead. However, it should be noted that these are only small sources of improvement. Even if call overhead is 100 cycles, the total time required for 720 thousand calls is no more than a few seconds.

Further improvements seem possible. In particular, further inlining would provide additional opportunities for loop restructuring to exploit reuse, possibly leading to better cache behavior.

3 A New Strategy for Interprocedural Optimization

Our previous experience has shown that, in many cases, secondary effects in the compiler and optimizer can obscure any improvement derived from cloning or inlining [CHT, Hal90].[5] The experiment with `matrix300` showed that these transformations can be used to enable high-payoff transformations, like cache blocking. This suggests a new strategy for deciding when to clone and when to inline: apply these interprocedural transformations only when they can enable application of a high-payoff transformation.

In this section, we develop this insight into an algorithm for deciding where to clone and where to inline. Section 3.1 discusses our approach to cloning. Section 3.2 describes our approach to inlining. Finally, Section 3.3 formalizes these ideas in algorithmic form.

3.1 Cloning Strategy

In our goal-directed approach, cloning is used to transform the program into a "nearby" program that exposes more interprocedural facts. The difference between the original program and the nearby version is simple. Some procedures in the original are implemented with multiple distinct copies in the nearby program. The calls to the original procedure are distributed among the clones in the transformed version.

This transformation does not change the "meaning" of the program; individual execution paths in the source code are unchanged, except for the names of procedures. Why, then, should the compiler clone?

[4] `dgemm` is invoked 8 times, on 300 × 300 matrices.

[5] In her thesis, Hall reports on an experiment that compared the results from interprocedural constant propagation against cloning to based on the values of interprocedural constants. Her experiment showed little direct benefit from this style of cloning [Hal90]. In our experiment with `matrix300`, we saw benefit from this strategy because it enabled the high-payoff transformations.

If the compiler partitions the calls on the basis of the interprocedural environment inherited by the called procedure, then each of the clones inherits a larger set of facts. This, in turn, allows the compiler to tailor the clones more closely to the run-time context in which they will be invoked. The key to successful cloning in `matrix300` was first determining which constants were important for the high-payoff optimization and then partitioning call sites around those values.

Reconsider `matrix300` from this perspective. The calls to `dgemm` used eight distinct values for `jtrpos`. Because constant propagation computed the *meet* of the incoming information, it concluded that the value of `jtrpos` was not known. After cloning, each call site invoked its own copy of `dgemm`. Thus, each copy received a consistent and known value for `jtrpos`. This, in turn, showed that `dgemv` was invoked with two distinct values for `job`. We partitioned the eight calls to `dgemv` by the value of `job`, necessitating two clones. This exposed the fact that `daxpy` was invoked with two distinct values for `ia`. Our algorithm would clone `daxpy` by `job` and then inline each of the clones. For `matrix300`, we applied these techniques by hand and skipped this last cloning.[6]

Formalizing these insights leads to a simple procedure for deciding where to clone and how many clones to create.

1. *discover "important" constants*
2. *in topological order, visit each procedure* p
   (a) *partition calls to* p *by values of the important constants*
   (b) *propagate newly exposed constants through procedure bodies*

Conceptually, the difficult part of the problem is identifying "important" constants. Not all constants are useful in optimization. An excellent example is the set of string constants passed to an error message routine. Cloning based on these constants would probably not improve normal execution time.

Part of the difficulty arises from the fact that the compiler cannot know, in advance of cloning, what constants will be available in the final program. Before cloning, it can look at the values found by interprocedural constant propagation. But, cloning increases the set of known constants – that is the whole point of our cloning strategy. Thus, the cloning phase cannot know, before it starts, what results it will obtain.

Therefore, we seek to identify *variables* whose values would be useful in establishing the safety of our high-payoff transformations. For the memory-management transformations, these are values that (1) determine control flow, (2) specify array dimensions, (3) dictate loop bounds and strides, or (4) appear in subscript expressions. For a procedure p, we call the set of important variables `CloningVars(`p`)`. Because we are looking at an interprocedural effect, `CloningVars(`p`)` can contain only formal parameters of p and variables that are global to p.

To compute `CloningVars(`p`)`, we formulate it as a backward interprocedural data-flow analysis problem. This problem divides naturally into two distinct phases.

[6]Of course, it is simple to adjust the algorithm so that it does not clone leaf procedures. The decision as to whether or not this clone should be made is directly related to another question: how should the compiler represent clones? If clones have large representations, the algorithm probably should not clone the leaf procedures.

Local analysis Initially, the compiler examines each procedure to locate values that fit into the "important" category. Next, it finds the globals and formals that directly affect these values. The "direct" effect may be partially masked by intervening assignments and other statements. The compiler also will need the standard information required for interprocedural analysis, like mappings of formals to actuals, descriptions of call graph components, and jump functions for constant propagation [CCKT86].[7]

Propagation The `CloningVar` sets must be propagated backwards through the call graph, applying the appropriate mapping functions to model parameter binding. For each procedure, the final `CloningVar` set will contain the formals and globals that are "important", both in the procedure and in any of its descendants.

The full paper will provide more detail on the computation of `CloningVars(`p`)`.

3.2 Inlining Strategy

Once cloning is complete, the compiler has exposed all the constants that it will discover. At this point, it can proceed to select call sites for inlining. Our goal for inlining is simple – we want to inline call sites in a way that enables application of the high-payoff optimizations. This divides the process into two parts: identifying shallow loop nests where the optimization would be profitable and determining if inlining can create a deep enough loop nest to allow its use.

For the memory-management optimizations, identifying the loops that can be improved is easy. The key idea behind the memory-management work is adjusting *balance* [CCK87]. We say that a loop nest has a balance – the ratio of memory traffic to computation, measured in cycles. The machine also has a balance – the ratio of memory access to computation sustainable by the processor. If the loop nest and the machine have radically different balances, then the computation is either compute-bound or memory-bound. We can often transform memory-bound loop nests to make them more closely match the machine's balance.

To identify sites where the memory-management transformations may be profitable, the compiler can simply examine each inner loop and compute its balance. When it discovers a memory-bound, singly-nested loop, it should look back up the call chain to determine whether inlining could create an appropriately deep loop nest.

This leads to the following algorithm for selecting call sites to inline. For each inner loop, perform the following steps:

1. *compute the loop's "balance"*
2. *if the loop is memory bound and the nest is shallow*
   (a) *search back along the call chain for an enclosing loop*
   (b) *inline to move loops into same procedure*

This algorithm will inline call sites where it appears that the memory management techniques can produce a reasonably high payoff. We expect that this strategy will lead to consistently profitable application of inlining.

[7] Depending on implementation, it may also need a mapping from actual parameters into the globals and formals that determine the actual's value. This mapping is found by inverting the *support* set of the actual parameter's jump function.

From an implementation perspective, there are a number of additional interesting points.

- A compiler that applies these techniques will undoubtedly perform other kinds of interprocedural analysis. It can easily precompute information about loop nests that contain call sites. This could eliminate the backward search along the call chains for enclosing loops.
- At the same time, the compiler can compute a simple estimate of execution frequency for each call site. Using actual loop bounds, when known, and the classical estimate of eight or ten trips per loop, the analyzer can accumulate, for each call site contained in a loop, an estimate for the number of calls. This simple method ignores recursion; it is not hard to factor in cycles in the call graph. With such estimates, the compiler could limit its attention to loops with estimated frequencies above some threshold value.
- As with any interprocedural technique, recompilation is a concern. Whenever we inline a call site, we increase the amount of compilation needed after a change to either procedure. To limit the number of new recompilation dependences, the compiler might place an absolute limit on the length of a call chain that it will inline. This makes sense from another perspective; if we must inline a large number of calls to get a doubly-nested loop, the resulting loop nest may be complex enough to make analysis difficult. For example, it is unlikely that the loop nest will be perfectly nested!
- Step *2a* must be somewhat more complex. The memory-management techniques can only be applied if the final loop nest has no remaining calls. Thus, the compiler must inline all calls in the nest if it inlines any call in the nest. Therefore, the actual test will be more complex than we have shown. It must account for linguistic and implementation limitations that would prevent inlining on *any* call in the nest.

If we applied the method described in Section 3.1 to `matrix300`, and this method to the resulting program, we would derive the same version of `matrix300` that our hand-simulation produced. As shown in Section 2, this program is amenable to cache and register blocking.

3.3 Algorithm

Combining the techniques discussed in the previous two sections gives us the following algorithm.

Phase 1 Cloning

1. *Compute* `CloningVars(p)` *for each procedure*
2. *In topological order, visit each procedure* `p`*:*
   (a) *partition calls to* `p` *by* `CloningVars(p)` ∩ `Constants(`*caller*`)`
   (b) *create a clone for each partition*
   (c) *for each clone* `c`*, propagate constants through* `c`

Phase 2 Inlining

for each inner loop that contains no calls

1. *compute the loop's "balance"*
2. *if the loop is memory-bound and singly-nested*
   (a) *check if inlining is possible and will create an appropriate loop*
   (b) *if indicated, inline the call chain*

If applied to `matrix300`, it would yield the same program that resulted from our hand transformation.

4 Conclusion

Our early experience with inlining and cloning showed mixed results [CHT, Hal90]. However, the experiment with `matrix300` showed that inlining and cloning can be used to expose new opportunities for high-payoff transformations like cache and register blocking. These high-payoff transformations differ in two fundamental ways from classical optimizations:

1. they require dependence analysis and adequate program context, and
2. the potential for improvement is substantial – we expect 100 percent or more improvement

Stated simply, our goal-directed approach to interprocedural optimization suggests that we inline and clone to create contexts in which high-payoff optimizations can be applied. Thus, while widespread use of inlining and cloning may not be effective in accumulating small performance improvements , limited and directed use can open the door for advanced optimizations.

5 Future Work

This paper suggests a new approach to deciding where to apply inlining and cloning. Our intent is to use these two transformations to enable important, high-payoff optimizations. We are currently building a compiler that will automatically apply all of the transformations discussed in this paper. Among our projects for the future are:

- Exploring the effectiveness of our strategy with other high-payoff transformations such as vectorization and parallelization.
- Validating the techniques on other scientific FORTRAN programs.
- Reducing the amount of cloning.

  We can reduce the amount of cloning by observing when clones of a procedure are "equivalent." For example, in `matrix300`, four of the clones of `dgemm` bound `job` to `1` when calling `dgemv` and the other four clones bound `job` to `3` – the clones in each group can be considered equivalent. The process of locating equivalent clones is similar to minimizing the number of states in a finite state machine [AHU74].

6 Acknowledgements

Steve Carr and Ken Kennedy shaped our thoughts on transformations to manage memory hierarchies. Through questions and encouragement, Kaivalya Dixit provided the impetus that began and sustained this work. Our colleagues at Rice have produced a software-rich environment in which we are able to conduct experiments like this. All of these people deserve our deep thanks.

References

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[CCK87] David Callahan, John Cocke, and Ken Kennedy. Estimating interlock and improving balance for pipelined architectures. In *Proceedings of the 1987 International Conference on Parallel Processing*, August 1987.

[CCK90] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 53–65, June 1990.

[CCKT86] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 152–161, July 1986.

[CHT90] Keith D. Cooper, Mary Hall, and Linda Torczon. An experiment with inline substitution. *Software – Practice and Experience*, 1990. To appear.

[DH88] Jack W. Davidson and Ann M. Holler. A study of a C function inliner. *Software – Practice and Experience*, 18(8), August 1988.

[Hal90] Mary Hall. *Managing Interprocedural Optimization.* PhD thesis, Rice University, October 1990.

[RG89] Steve Richardson and Mahadevan Ganapathi. Interprocedural analysis versus procedure integration. *Information Processing Letters*, 32(3), August 1989.

[Sch77] Robert Scheiffler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9), September 1977.

[Yer66] Andrei P. Yershov. Alpha – an automatic programming system of high efficiency. *Journal of the ACM*, 13(1), January 1966.