Goal-Directed Interprocedural Optimization*

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

Abstract – Previous work on interprocedural optimization has focused on accumulating small improvements by eliminating procedure call overhead and improving global optimization. In our experience, this produces modest improvements while incurring substantial compile-time costs. We propose a new approach to interprocedural optimization – a *goal-directed* strategy. Our approach is conceptually simple: only use interprocedural transformations when they can enable a high-payoff intraprocedural transformation. This paper describes our approach and presents several worked examples.

Index terms – Optimization, interprocedural transformations, inline substitution, procedure cloning, interprocedural analysis, loop transformations

1 Introduction

Trends in microprocessor design and system design are shifting an increasing share of the burden for performance onto the compiler. Microprocessor features like superscalar instruction issue and longer pipelines force compilers to use more aggressive optimization and scheduling [1, 2]. System features like multilevel memory hierarchies and multiprocessor parallelism force compilers to adopt aggressive dependence-based restructuring schemes [3, 4, 5]. The net result is to make application performance increasingly sensitive to compiler effectiveness. This makes code optimization technology a critical contributor to system performance.

As pressure on code optimization increases, so does interest in interprocedural optimization. This is natural. The history of code optimization suggests that increasing the context available to the optimizer leads to better code. Unfortunately, interprocedural optimization can have negative effects.

- The resulting code can run slower than the original [6].
- Compilations can take *much* longer [7].
- Using *any* interprocedural transformation introduces a recompilation problem [8].

These negative effects are particularly pronounced in compilers that perform extensive optimization. The heart of the problem is the compile-time expense of interprocedural optimization compared with payoffs,

*This work has been supported by IBM Corporation and by DARPA through ONR contract N00014-91-J-1989.

which are typically modest and sometimes are negative. A common solution is to avoid interprocedural transformations entirely; we advocate a more aggressive approach – a *goal-directed* approach.

Our strategy is to use interprocedural transformations to enable the application of high-payoff optimizations – optimizations that routinely improve program run-time by a factor larger than the potential degradation. By targeting high-payoff optimizations, we can reasonably expect that the benefits after interprocedural optimization will be significant enough to outweigh any secondary effects in the optimizer. At the same time, we incur the compile-time costs of interprocedural transformations only when the compiler can expect a large payoff.

This paper focuses on two specific interprocedural transformations. *Inline substitution* is a simple form of interprocedural code motion [9]. It replaces a procedure call with a copy of the code for the called procedure. Of course, names must be translated to model the effects of parameter binding and to merge the name spaces. *Procedure cloning* lets the compiler produce multiple implementations of a single procedure [10]. The compiler creates several copies of the procedure and distributes the calls among them. By connecting similar calls to a distinct implementation, the compiler can generate code tailored to the environment at those calls.[1]

The paper is organized into three principal sections. In the next section, we present an overview of our goal-directed approach to managing interprocedural optimization. In Section 3, we provide several examples of target optimizations suitable for use with our strategy. The final section gives a detailed algorithm for one goal-directed strategy and describes two experiments that have used the techniques.

2 Strategy

The key contribution of this paper is a framework for using interprocedural analysis and interprocedural transformations to improve the running time of the program. Previous approaches have focused on accumulating small improvements by eliminating call overhead and improving global optimization. Our strategy selects a few high-payoff *intraprocedural* transformations and uses a combination of *interprocedural* analysis and transformations to create new opportunities for applying them.

This strategy is the result of a long search for an effective approach to managing interprocedural optimization. A turning point in our quest was a study on the efficacy of inline substitution. We used source-to-source transformation and five commercial FORTRAN optimizing compilers [6]. The results were disappointing. With each of the five compilers, we found cases where the side effects of inline substitution overwhelmed improvements, resulting in a net slowdown of the code. This happened even in cases where inline substitution eliminated effectively *all* the call overhead. To make matters worse, there was no

[1] Procedure cloning is similar to the notion of customization in Self [11].

consistency between compilers; the code that provoked one compiler's worst degradation produced another compiler's best improvement.

This experience, along with other experiments [12], has led us to propose a *goal-directed* strategy. Our approach can be stated simply. The compiler should only apply interprocedural transformations when the potential profit outweighs the potential problems. To implement this notion, the optimizer should:

1. Locate focus points in the program. A focus point must have two properties:
   (a) the target intraprocedural transformation might be profitable if only it could be applied, and
   (b) supplying more context from other procedures would make application possible.
2. Use a combination of interprocedural data-flow information, interprocedural code motion, and procedure cloning to transform the focus point in a way that makes it possible to apply the target optimization.
3. Apply the target optimization.

This scheme only applies interprocedural transformations when they enable application of one of the target optimizations. By targeting high-payoff optimizations, we expect that the improvement from the target transformation will outweigh any degradation introduced by the interprocedural transformations.

3 Target Optimizations

To make this discussion concrete, a few examples of target optimizations are needed. To work well in our strategy, a target optimization must satisfy two requirements:

1. It must produce a relatively large improvement when applied. The study on inline substitution suggests an expected payoff of 20% may be needed to overcome possible side effects [6].
2. It must require some surrounding context for application. If this is not the case, then the interprocedural techniques will not create new sites where it can be used.

We have experimented with two target optimizations: register blocking and loop parallelization. Several others suggest themselves, including specialization of polymorphic procedures based on type information and memory hierarchy management (*e.g.*, blocking for cache).

Simply identifying a target optimization is not enough. We must determine how much context is required to establish both safety and profitability and how interprocedural techniques might create a context in which it can be applied. Finally, we must be able to locate a focus point (at compile-time) where the optimization would be profitable if applied. The following sections explore some specific target optimizations.

3.1 Register Blocking

In scientific computation, the performance of inner loops often determines the overall performance of the program. As memory latencies rise relative to instruction times, the presence of loads and stores in

```
do i ← 1, 2n
      do j ← 1, m
            x[i] ← x[i] + y[j]
      end
end
```

⇓

```
do i ← 1, 2n, 2
      do j ← 1, m
            x[i] ← x[i] + y[j]
      end
      do j ← 1, m
            x[i + 1] ← x[i + 1] + y[j]
      end
end
```

⇓

```
do i ← 1, 2n, 2
      do j ← 1, m
            x[i] ← x[i] + y[j]
            x[i + 1] ← x[i + 1] + y[j]
      end
end
```

⇓

```
do i ← 1, 2n, 2
      x0 ← x[i]
      x1 ← x[i + 1]
      do j ← 1, m
            t ← y[j]
            x0 ← x0 + t
            x1 ← x1 + t
      end
      x[i] ← x0
      x[i + 1] ← x1
end
```

Figure 1 Register blocking transformations

these inner loops becomes an increasingly important factor in determining run-time speed. Of course, any performance problem can be viewed as an opportunity for code optimization; the rise in relative latency of memory has led to effective techniques for eliminating array references in loops [13]. For our purposes, we consider *loop fusion, scalar replacement,* and *unroll-and-jam* as a single class of optimizations that we term *register blocking.*

Loop fusion Adjacent loops are merged if their iteration spaces are identical and the loops reference the same memory locations. By moving definitions and uses into the same loop, we can create new opportunities for scalar replacement to eliminate memory accesses.

Scalar replacement The optimizer uses the results of dependence analysis to find array element values that are reused within a loop and replaces references to them with references to compiler-generated scalar temporaries. This allows the values to be placed in registers, eliminating loads and stores in the loop.

Unroll-and-jam After scalar replacement, many loops use only a small fraction of the floating-point registers. If a loop nest contains at least two loops and loop interchange is safe, then the compiler can apply unroll-and-jam – outer loop unrolling followed by fusing the new inner loop bodies. If the outer loop carries reuse, then unroll-and-jam eliminates additional loads and stores.

Figure 1 shows the effect of each transformation. The first code fragment is a simple doubly-nested loop. The second fragment depicts the first step of unroll-and-jam – the outer loop has been unrolled. The third fragment shows the result of fusing the two inner loop bodies. The final fragment displays the effect of scalar replacement. It encodes the reuse of array elements in scalar temporary variables.

When it can be applied, register blocking is quite effective at converting a *memory-bound* loop into a *compute-bound* loop. To determine if a loop is memory-bound, we compute the *balance* of both the loop (β_l) and the machine (β_m):

$$\beta_l = \frac{\textit{flops/iteration}}{\textit{memory accesses/iteration}}$$

$$\beta_m = \frac{\textit{flops/cycle}}{\textit{memory accesses/cycle}}$$

If $\beta_l < \beta_m$, the loop is memory-bound. If $\beta_l \geq \beta_m$, the loop is compute-bound (the machine's floating-point units are 100% busy) and blocking would have no major effect [14].

Carr has shown that register blocking often produces speedups by factors of 2 to 3 on the SPARC and MIPS machines [13]. Because of the high payoff, register blocking is an obvious candidate for our goal-directed approach. Further, register blocking meets the second criterion since unroll-and-jam requires a loop that is at least doubly-nested and fusion requires two adjacent loops. Interprocedural transformations can provide opportunities for register blocking by exposing loop nests to these transformations. In Section 4.2, we show an example where a goal-directed interprocedural strategy to register blocking produces speedups

of 2 to 3 on a code that defies the intraprocedural techniques. Carr has reported a factor of 6.4 improvement on an IBM RS/6000 with the same code [15].

3.2 Improving Parallelism

An area that has received much attention in both the literature and in practice is the automatic detection of parallelism. Typical parallelizing compilers search for loops that can be run in parallel. The key to profitable parallel code generation is having enough *granularity*, or useful computation, inside each parallel loop to offset the synchronization costs. Otherwise, parallelizing a loop can lead to significant performance degradation.

Sometimes, transformations can be applied to a loop nest to increase granularity of a parallel loop. For example, *loop interchange* swaps two loop headers in a perfect loop nest, reordering the iterations of the nest. If a parallel inner loop is interchanged with a sequential outer loop, the amount of computation in the parallel loop may be significantly increased. Interchange is just one of many transformations applied to loop nests to increase granularity, expose parallelism, or match the number of parallel processes to available processors.

Procedure calls impede parallel code generation in two different ways. First, calls may interrupt a loop nest. As an example, the standard tests for safety of loop interchange are difficult or impossible to apply when the loops of a nest are split between two or more procedures; it is also difficult to apply the transformations correctly. Second, calls may hide an opportunity. If the routine containing the inner loop is called from multiple places, it may not always execute inside another loop. In each of these cases, our strategy may remedy the problem.

- Reformulating the safety and profitability tests for a transformation to incorporate interprocedural information in place of program text allows these tests to be applied to loop nests spanning multiple procedures.
- Moving the loops into a single procedure creates a context where transformations on the loop nest may be applied.
- Cloning multiple copies of the called procedure can create the necessary context in the call graph. This works by constructing a path through the call graph that ends with the desired environment for the parallel region.

Hall, Kennedy, and McKinley used a goal-directed strategy to transform loops in several programs from the PERFECT benchmark suite [16]. They introduced two new forms of interprocedural code motion – *loop embedding* and *loop extraction*. Loop extraction is a form of partial inline substitution, while loop embedding is the inverse operation. These transformations were combined with interprocedural analysis and procedure cloning to enable interchange and fusion of loops spanning multiple procedures. They followed these with loop parallelization. Interprocedural optimizations were applied only when fusion, interchange,

and parallelization were enabled and performance estimation predicted that the transformed loops would have a better execution time.

4 Examples

To solidify the picture, this section presents a series of examples. The first section explores the use of goal-directed techniques for register blocking. The second section briefly describes two experiments that applied goal-directed techniques to real programs.

4.1 A Register Blocking Strategy

Not all memory-bound loops are amenable to intraprocedural register blocking. Sometimes, the compiler can use interprocedural techniques to transform the code so that these loops can be blocked. To find and transform these sites, the compiler should employ a goal-directed strategy.

Interprocedural techniques can improve the results of register blocking in two ways. First, knowledge about interprocedural constants can sharpen the dependence information used to test for safety and profitability. Second, moving code across procedure calls may create new places where the transformations can be applied. To create a goal-directed strategy, we must tie the application of interprocedural transformations directly to our target optimization, register blocking.

Thus, our overriding goal is to improve the balance of loops, as described in Section 3.1. To achieve this goal, we establish sub-goals. Figure 2 gives an overview of the entire process. The following paragraphs provide more detail on each sub-goal.

Find the opportunities To implement a goal-directed strategy successfully, we must have an effective way of identifying points in the code where the strategy can pay off. For register blocking, the key is to recognize memory-bound inner loops that cannot be blocked in their intraprocedural contexts. Thus, for each inner loop, we compute β_l. If $\beta_l < \beta_m$, the loop is memory bound. When we find a memory-bound loop contained in an obvious loop nest, we can directly block the loop with scalar replacement and unroll-and-jam.

If, in its current procedure, the loop is not nested inside another loop, we mark it as a candidate for interprocedural help. (The word "candidate" is chosen carefully; procedure cloning and inline substitution will only help if a call to the current procedure originates inside another loop. Subsequent steps in the algorithm discover such nesting.) For each candidate loop, we make note of all critical parameters – that is, parameters to the procedure whose values may be important to the blocking transformations.

Compute interprocedural information For our strategy to improve the object code, two conditions must be met. First, a combination of procedure cloning and inline substitution must create a multiply-nested loop; at least two loops are required for unroll-and-jam. Second, the compiler must have enough information about

1. *find the opportunities*
   (a) identify memory-bound loops
   (b) if > 2 loops in nest $\Rightarrow$ block it
   (c) if single loop, identify critical parameters
2. *compute interprocedural information*
   (a) build an initial call graph
   (b) propagate loop nesting summaries
   (c) propagate interprocedural constants
3. *refine the information*
   for each procedure p, in topological order,
   (a) partition calls to p by critical constants
   (b) clone a copy of p for each partition
   (c) propagate exposed constants through p
4. *inline to deepen loop nests*
   for each shallow, memory-bound loop
   (a) if call is in a loop $\Rightarrow$ inline the call
   (b) block the newly created loop nest

Figure 2 Goal-directed register blocking

the loop nest to let it determine questions of safety and profitability. Thus, the second step of the algorithm performs interprocedural analysis to discover the necessary facts. The compiler uses a three-step process. It builds a call graph for the program as written [17]. It creates a summary for each loop that represents the iteration space – loop index, bounds, and stride – and propagates these around the call graph. It performs interprocedural constant propagation [18].

Refine the information Our goal is to change the interprocedural structure of the program in a way that enables additional register blocking. Thus, the third sub-goal is to create an equivalent program that exposes more information. When multiple call sites invoke the same procedure, the compiler must rely on only those facts that hold across all of the call sites. Thus, when paths through the call graph merge, information may be lost. For example, if one site passes the value 3 to parameter x and another passes the value 4, the compiler cannot rely on either value. By cloning a separate implementation for each call, the compiler can create an environment where it can use both facts. This may lead to better optimization.

To capitalize on this observation, the compiler examines the procedures in topological order.[2] It partitions calls to a procedure p by constants passed to the critical parameters identified in step 1. Next, it clones a copy of p for each partition. To refine later procedures, the compiler then propagates any newly exposed

[2]This discussion ignores recursion. Cooper, Hall, and Kennedy present a more detailed discussion of procedure cloning [19]. They discuss procedure cloning to refine arbitrary forward interprocedural data-flow sets and how to handle recursive cycles.

constants and loop summary information through the body of p so that they can serve as a basis for the cloning of procedures that p calls.

Note that this new program is equivalent to the original program. Along each execution path, the same sequence of statements occurs. It differs from the original because the paths share less code. This eliminates some of the information loss described earlier.

Inline to deepen loop nests The final step is simple. For each candidate loop, the compiler examines the loop summaries and interprocedural constants. If the interprocedural information reveals a call site where inline substitution would produce a loop nest amenable to blocking, the compiler inlines that call site and blocks the loop.

4.2 Experiments

This section describes two experiments that led to the development of our goal-directed strategy. Both target register blocking. A third experiment targeting parallelism is described by Hall, Kennedy and McKinley [16].

Matrix300 Our approach was initially inspired by an attempt to optimize the program `matrix300`, from Release 1 of the SPEC benchmark suite. From this experience, we learned the value of procedure cloning and inline substitution to enable unroll-and-jam.

Most of the computation in `matrix300` is located in the leaf procedure `daxpy`. Unfortunately, `daxpy` contains a single loop in which every value is used only once, so there is no potential for reuse. To expose reuse, a compiler must inline `daxpy` into its caller `dgemv` and perform unroll-and-jam. The left side of Figure 3

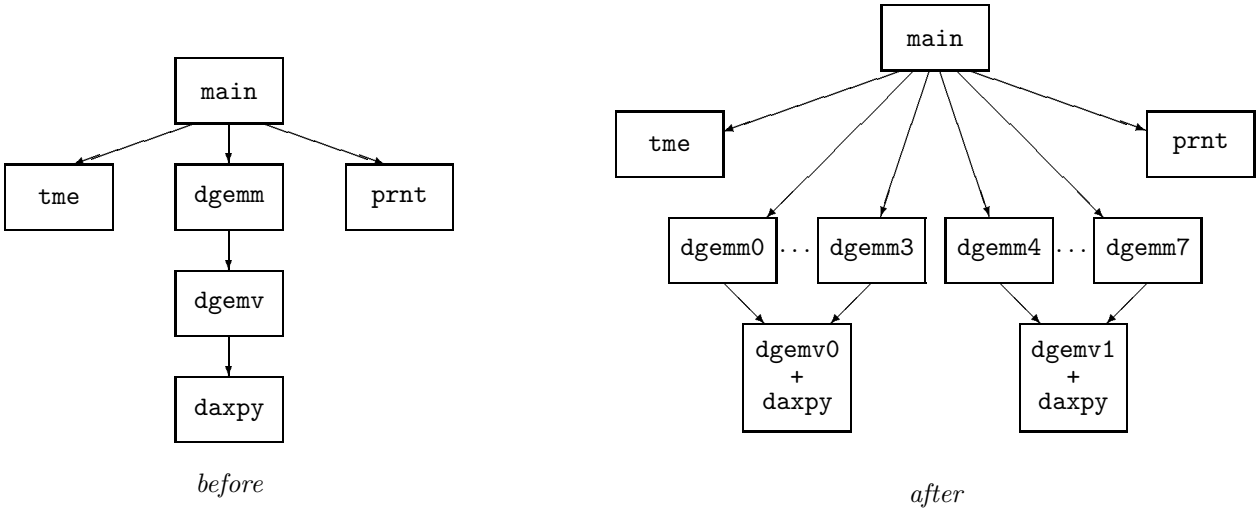

Figure 3 Call graph for `matrix300`

shows the call graph for the program as written.

Unfortunately, directly inlining `daxpy` results in subscripts that are too complex for current dependence analyzers. This effectively rules out unroll-and-jam, since it relies on dependence information to prove safety. Applying our strategy, however, creates a version of the code that can be optimized.

Procedure cloning based on critical parameters exposes enough additional information to allow simplification of the subscript expressions. Specifically, it exposes the size of the leading dimension of an array that is passed to `daxpy` as a formal parameter. This parameter receives it value indirectly from a formal parameter of `dgemm`. Main invokes `dgemm` from eight distinct call sites, each passing a different constant value in that parameter position. Cloning `dgemm` and subsequently `dgemv` enabled `daxpy` to be inlined into the two copies of `dgemv` with simple subscript expressions for the reshaped array. At this point, it was possible to apply unroll-and-jam. The right side of Figure 3 shows the call graph that resulted from this set of transformations.

To see the effects of the interprocedural transformations, we applied them by hand to `matrix300` to create the new call graph. Next, we applied register blocking to the new loop nests in the two inlined versions of `dgemv`. We compiled the two versions of the program on the Sparc-1, the MIPS M/120 and the IBM RS/6000-540. The results were outstanding.

`matrix300`	SUN Sparc-1	MIPS M/120	RS/6000 Model 540
speedup	2.0	3.3	6.4

Most of the improvement is due to eliminating nearly 500,000,000 memory accesses – almost half the loads and nearly all the stores in the program. Of course, this program is essentially a kernel; improvements on real programs will typically be much smaller.

Other experimenters have since achieved similar results on `matrix300` with user-specified inline substitution. The goal of our work is to produce a decision procedure that *automatically* selects focus points and derives a strategy to improve them. The strategy presented in Section 4.1 would lead the compiler to produce the code depicted in Figure 3.

Ocean Experimentation with programs in the PERFECT benchmark suite suggested using inline substitution to enable loop fusion. In particular, five of the PERFECT programs contain points where this kind of fusion exposes reuse. These programs are `adm`, `dyfesm`, `ocean`, `spec77` and `track`.

To understand the potential, we hand-simulated the actions of a goal-directed optimizer on `ocean` to produce the fused loop nests. Then, we applied scalar replacement to take advantage of the reuse exposed by loop fusion. In all, sixteen fused loops appeared in the transformed program.

We executed the original and transformed version of the program on an IBM RS/6000-540. The execution times and associated speedups for the whole program are summarized in the table below:

`ocean`	optimized portion	program
Original	18.5 s	202.1 s
Fused	15.2 s	199.0 s
Speedup	1.22	1.02

The column labeled "optimized portion" reflects the improvements achieved in the fused loops. The "program" column reflects the improvement over the entire program. Note that the low overall improvement is partially due to an "apples-and-oranges" comparison. We have applied a higher level of optimization to the sixteen fused loops than to the rest of the program. A similar level of optimization on the rest of the program might decrease total execution time, increasing the importance of optimizing these sixteen loops.

5 Conclusions

This paper presents a strategy for goal-directed interprocedural optimization of programs. Our approach is conceptually simple: only use interprocedural transformations when they enable a high-payoff intraprocedural transformation. We have used this technique in several experiments where we hand-simulated its application. The results are promising. Other researchers intend to employ this strategy in a source-to-source parallelizer for shared-memory multiprocessors [20].

References

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

[2] A. Aiken and A. Nicolau, "Perfect pipelining: A new loop parallelization technique," in *Proceedings of the 1988 European Symposium on Programming*, Springer Verlag Lecture Notes in Computer Science, Mar. 1988.

[3] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," *SIGPLAN Notices*, vol. 26, pp. 30–44, June 1991. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*.

[4] S. Carr and K. Kennedy, "Blocking linear algebra codes for memory hierarchies," in *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing* (J. Dongarra, P. Messina, D. C. Sorensen, and R. G. Voight, eds.), pp. 400–405, 1990.

[5] M. Wolfe, *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois, Urbana-Champaign, 1982.

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

[7] S. Richardson and M. Ganapathi, "Interprocedural analysis versus procedure integration," *Information Processing Letters*, vol. 32, Aug. 1989.

[8] M. Burke and L. Torczon, "Interprocedural optimization: Eliminating unnecessary recompilation," *ACM Transactions on Programming Languages and Systems*, 1992. *to appear*.

[9] F. E. Allen and J. Cocke, "A catalogue of optimizing transformations," in *Design and Optimization of Compilers* (J. Rustin, ed.), Prentice-Hall, 1972.

[10] 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*, vol. 8, pp. 491–523, Oct. 1986.

[11] C. Chambers and D. Ungar, "Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language," *SIGPLAN Notices*, vol. 24, pp. 146–160, July 1989. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*.

[12] M. Hall, *Managing Interprocedural Optimization*. PhD thesis, Rice University, Houston, Texas, Apr. 1991.

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

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

[15] S. Carr, 1991. Personal communication about RS/6000 register blocking results for `matrix300`.

[16] M. W. Hall, K. Kennedy, and K. McKinley, "Interprocedural transformation for parallel code generation," in *Proceedings of Supercomputing '91*, pp. 424–434, Nov. 1991.

[17] D. Callahan, A. Carle, M. W. Hall, and K. Kennedy, "Constructing the procedure call multigraph," *IEEE Transactions on Software Engineering*, vol. 16, Apr. 1990.

[18] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation," *SIGPLAN Notices*, vol. 21, pp. 152–161, July 1986. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*.

[19] K. D. Cooper, M. W. Hall, and K. Kennedy, "Procedure cloning," in *Proceedings of the IEEE International Conference on Computer Languages*, pp. 96–105, Apr. 1992.

[20] K. S. McKinley, *Automatic and Interactive Parallelization*. PhD thesis, Rice University, May 1992.