A Methodology for Procedure Cloning*

Keith D. Cooper† Mary W. Hall‡ Ken Kennedy†

May 24, 1999

Abstract

Procedure cloning is an interprocedural transformation where the compiler creates specialized copies of procedure bodies. The compiler divides incoming calls between the original procedure and its copies. By carefully partitioning the calls, the compiler ensures that each clone inherits an environment that allows for better code optimization.

This paper presents a three-phase algorithm for deciding when to clone a procedure. The algorithm seeks to avoid unnecessary code growth by considering how the information exposed by cloning will be used during optimization. We present a set of assumptions that bound both the algorithm's running time and code expansion.

Keywords: cloning, specialization, interprocedural data-flow analysis, interprocedural optimization

1 Introduction

Compiler developers have long understood that procedure calls pose a barrier to code optimization. The problem shows up in two distinct ways: the overhead of the call itself and its impact on the code around each call site, and a degradation in the quality of information that the compiler derives. It has been widely assumed that call overhead is the more significant effect; a recent study of Fortran suggests that call overhead may play less of a role in run-time performance than believed [9].

Traditionally, two approaches have emerged for breaking down the call-site barrier. The first, *inline substitution,* replaces call sites with distinct copies of the body of the called procedure. The code is then optimized in the context of the calling procedure. The second, *interprocedural data-flow analysis,* estimates the set of compile-time provable facts about the environments passed and returned at procedure calls. This information is used, in turn, to optimize the individual procedures. Each technique has limitations. Inlining can lead to code growth, increased compile time, and degradation in code quality [9]. Using only interprocedural analysis lets the structure of the program constrain the compiler; it assumes that each procedure should be implemented once.

To improve the latter approach, an aggressive compiler can consider *procedure cloning* — creating multiple implementations of a single procedure and partitioning the calls among them [10].

*This research has been supported by the National Science Foundation, IBM Corporation, the Defense Advanced Research Projects Agency, and the State of Texas.

†*Department of Computer Science, Rice University, Houston, TX 77251*

‡*Center for Integrated Systems, Stanford University, Stanford, CA 94305*

- Cloning differs from straightforward application of interprocedural data-flow analysis. It changes the structure of the underlying graph used by the data-flow problem, removing some of the points where paths in the graph merge — it allows the compiler to solve a "nearby" problem that exposes a more useful set of facts for code optimization.

- Cloning differs from inlining. The actual code that implements the call is left intact. The compiler can map multiple calls onto a single copy of the procedure.

In its full generality, cloning can produce exponential growth in program size.

This paper examines how the compiler can apply cloning to expose information useful to optimization while avoiding exponential code growth. Our approach finds potential improvements in forward interprocedural data-flow solutions, considers their impact on optimization, and clones procedures leading to sharper information that will benefit optimization. The remainder of the paper is organized into five sections and a conclusion. The next section discusses similar techniques from intraprocedural optimization, partial evaluation and dynamic compilation. Section 3 presents an example from which we derive key insights for the cloning algorithm. Section 4 presents the three-phase algorithm for deciding when to clone. In Section 5, we provide bounds for both the time complexity of the algorithm and the resulting code expansion. Section 6 addresses how to further restrict cloning in pathological cases that exceed the assumed bounds.

2 Related Work

Techniques and algorithms very closely related to ours appear in intraprocedural optimization [19] and partial evaluation [4, 18]. Similar techniques appear in dynamic compilation of APL [15] and compilation of the dynamically-typed, object-oriented language SELF [8].

Wegman's *node distinction* replicates basic blocks in a procedure's control flow graph based on intraprocedural data-flow solutions and incrementally propagates the more precise solutions [19]. His algorithm uses heuristics to avoid some unnecessary replication. In partial evaluation, *specialization* involves replicating code in order to tailor copies to particular variable values or types [4]. Bulyonkov describes an abstract interpretation approach to locate program points where specialization improves analysis information, both in the interprocedural and intraprocedural settings. Ruf and Weise present an algorithm to reduce the amount of specialization in a partial evaluator [18]. Their algorithm merges specialized copies if they generate the same result at every statement even though the analysis information for the two copies is different (similar to Phase 2 of the algorithm described in Section 4.2).

Dynamic compilation also includes techniques similar to procedure cloning. Johnston presents the earliest such approach in his APL\3000 Dynamic Incremental Compiler [15]. When a statement is invoked for the first time, the compiler creates a compiled version specialized to the invocation environment. Environments for subsequent invocations are compared to the compiled version, and if necessary, the code is generalized to accomodate both environments so that only a single compiled version exists for that statement.

The SELF compiler combines dynamic and static compilation techniques [8]. *Customization* dynamically creates a specialized copy of a method based on the receiver type for a message invoking that method, which may generate copies for every possible receiver type in the program. A static technique, *message splitting*, replicates code at a point where two control flow paths merge; copies of the code are moved up into the

individual control paths if this will allow the copies to be specialized according to the unique environments of the separate paths.

The APL dynamic compiler creates only a single compiled version of a statement, relinquishing optimization benefits in favor of efficient compilation. There are techniques in both node distinction and partial evaluation that reduce the amount of replication, but neither will eliminate the possibility of exponential code growth. The techniques in the SELF compiler clearly have the potential for exponential code growth. Our work improves on these by recognizing and bounding the potential for exponential code growth while only cloning when there is a perceived optimization benefit.

3 Background and Motivation

To motivate our work on cloning, we summarize an experiment aimed at improving the performance of the program matrix300 from release one of the SPEC benchmark suite. We derive some key insights for the algorithm.

3.1 The Experiment

Matrix300 computes eight variants on matrix multiplication, selectively transposing the input and output matrices. The goal of the experiment was to apply a series of transformations to the program to improve its execution time [3]. These transformations reorder the iteration space of a loop to expose reuse of values in registers and decrease cache misses. The most important of these optimizations, *unroll and jam,* has demonstrated dramatic improvements on linear algebra kernels [6].

Unroll and jam cannot be applied directly to the key computational kernel of matrix300 because of the program's structure. Unroll and jam transforms a nest of two or more loops; in matrix300, each loop is in a different procedure. The leaf procedure, *daxpy,* only contains a single loop. The code in *daxpy* reveals little or no reuse of values in either registers or cache. This loop is a good candidate for memory optimization, but needs to be inlined into the caller *dgemv* to expose an outer loop.

Unfortunately, the call in *dgemv* performs an *array reshape* — the actual and formal parameters have different dimension sizes. Inlining *daxpy* translates the reference `A(1,i)` to the linearized form `A(k+(i-1)*ii,1)`, where the value of `ii` is unknown at compile time. The multiplication by `ii` makes this subscript expression too complex for *dependence analysis*, upon which these transformations rely [16]. The memory optimizations rely on precise dependence information to locate reuse and to prove safety conditions. Thus, directly inlining the call creates the necessary loop structure, but leaves the code in a form where the transformations cannot be applied. In fact, all the calls in matrix300 must be inlined before enough information is exposed to simplify this subscript expression.

Applying cloning prior to inlining alleviates these problems. To illustrate these points, the call graph for matrix300 is shown in Figure 1, annotated with the relevant pieces of code. The value of `ii` — the dimension size of array `A` — passed at the call to *daxpy* depends only on the evaluation of the *if* condition in *dgemv,* which in turn, depends only on the value of the input parameter `job`. The value of `job` depends solely on *dgemm*'s input parameters `jtrpos` and `job`. `Jtrpos` takes on the integer values from 0 to 7, while `job` always has the value 1.

Taken together, this suggests cloning the eight calls from *main* to *dgemm* to expose unique constant

values for `jtrpos`. This results in *dgemv*'s formal `job` receiving a value of either 1 or 3. By making two copies of *dgemv,* we can finally determine the value of `ii`, the dimension size passed at calls to *daxpy.* For the case where `job` has the value 1, the value of `ii` is 1, so the reference to `A` after inlining *daxpy* becomes `A(k+i,1)`. When `job` has the value 3, no reshape of `A` occurs at the call so the translated reference is `A(k,i)`. Finally, we can inline *daxpy* and perform the memory optimizations. The resulting call graph for matrix300 is shown in Figure 2.

3.2 Key Insights

The algorithm described in this paper was motivated by four key insights, presented in this section. The first three of these were derived from the preceding example.

3.2.1 Propagation

Cloning changes the structure of the call graph in a way that allows interprocedural constant propagation to proceed along distinct paths. More precisely, cloning removes some of the points of confluence — those points where the constant propagation algorithm uses a *meet function* to approximate the facts that are true along two converging paths. By avoiding these approximations, cloning may sharpen the results of the analysis. We can exploit the change in graph structure by cloning a procedure with invocations that contribute significantly different interprocedural information. In our example, we applied cloning when calls contributed different constant values for variables in a called procedure. We propagated the effects of cloning to descendants in the graph since cloning a procedure may in turn expose opportunities for cloning its descendants.

We would like to generalize cloning to improve the results of other interprocedural data-flow analysis problems. Data-flow information provides a good basis for cloning decisions. It is easily manipulated; these problems are formulated as systems of equations on a lattice framework. It has path-specific components, but they can be readily merged to represent aggregations of multiple paths. Finally, it has a direct impact on the quality of the code generated by the compiler. The cloning algorithm should select modifications that result in data-flow information that more precisely models the events that happen at run-time.

In general, our algorithm can be used to clone based on partial solutions to any *forward* interprocedural data-flow analysis problem (*i.e.,* a problem where a node inherits information from its predecessors in the graph, rather than its successors). Examples of forward interprocedural problems are constant propagation, alias analysis and type analysis. This approach directly sharpens the solution to the forward interprocedural problem used as the basis for cloning.*

3.2.2 Goal-directed Cloning

The above discussion suggests that we calculate solutions to the forward interprocedural problems and use these directly as the basis for cloning decisions. Unfortunately, compilers cannot capitalize on every new data-flow fact that is exposed. For example, it would not be profitable to clone based on different constant

*Note that the change in the graph may also indirectly sharpen solutions to backward data-flow problems. For example, changes in the results of alias analysis or constant propagation (both forward problems) can change the results of side-effect analysis (a backward problem). However, it is unclear how to predict the impact a cloning decision will have on the solution to a backward data-flow problem.

values of a string passed to an error printing routine. Thus, a good cloning technique should try to distinguish between facts that have an impact on code quality and those that do not.

We can avoid unnecessary code growth by restricting cloning to those cases where important information is exposed. We describe such a strategy as *goal-directed* [3]. In the matrix300 example, we clone only to expose constants needed to improve the results of dependence analysis. These constants fall into three categories: (1) they specify the dimension size of an array parameter; (2) they determine control flow; or, (3) they appear in a subscript expression. These constants expose sufficient information to perform inlining and unroll and jam.

We detect important constants by examining each dimension statement, control flow test and subscript expression in a procedure. Suppose such an expression could be evaluated assuming all global variables and formal parameters in the expression enter the procedure with constant values. If we can clone to expose constant values for these variables, then it is likely that more precise dependence information will result. A bottom-up pass over the program propagates these variables, translating from formal to actual parameters at calls. This approach derives *ImportantVariables*(p), the variables of procedure p that, if constant, might improve dependence information in this procedure or one of its descendants [3, 12].

For other forward data-flow problems, a goal-directed approach depends both on the problem and the desired optimization effects. Designing a strategy for a specific compiler necessarily involves experimentation to understand how well the compiler takes advantage of the kind of facts that cloning can expose. By understanding what data-flow information results in profitable optimization, the compiler can filter potential cloning opportunities to only expose these data-flow facts.

3.2.3 Merging Equivalent Clones

As described above, we can avoid unnecessary cloning by ignoring information about variables that cannot have an important effect on optimization. In some cases, we can further reduce the amount of unnecessary cloning by merging clones that produce the same effects on optimization. In matrix300, eight copies of *dgemm* were made corresponding to the eight possible constant values of one of its input parameters. However, only two copies were needed to tailor the two versions of *daxpy* in order to apply inlining and unroll and jam. By evaluating important expressions in the program based on the constant values provided by cloning, we can determine if two clones generate the same values for these important program points. If so, then the two clones are "equivalent" from the standpoint of the target optimization and can be merged.

The second phase of the cloning algorithm locates equivalent clones and merges them. We discuss this phase in the context of constant propagation. It turns out that this phase is only necessary for some interprocedural problems, which we characterize in Section 4.2.4.

3.2.4 Exponentiality

The final insight about cloning is perhaps the most important. In its full generality, cloning can result in exponential growth in compile time and object code size. The example shown in Figure 3 demonstrates this point. In the initial program, shown in 3(a), there are n procedures in the program, $p_1, p_2, \ldots, p_n$. Each procedure p_i invokes p_{i+1} at two call sites. At one call, the procedure p_i passes as a parameter $(2i-1)*input$. The other call passes the value $2i*input$. By producing clones for each unique value of the input parameter at p_2, we produce the call graph shown in 3(b). By doing the same for p_3, the call graph shown in 3(c)

results. After cloning all calls in the program, the final call graph has $2^n - 1$ nodes and $2^n - 2$ edges. The original call graph has only n nodes and $2(n-1)$ edges.

Because cloning can exhibit exponential behavior, our algorithm must anticipate this possibility and impose restrictions when necessary. However, based on experience, the amount of useful cloning on a program is likely to be small [12]. For this reason, we expect that the restrictions on cloning will rarely be necessary. Nevertheless, the algorithm will perform well even in the event of pathological behavior.

4 Cloning Algorithm

This section presents a polynomial-time algorithm for procedure cloning. The algorithm has three phases. First, we propagate vectors of interprocedural information describing the possible cloning that can be performed on the program. In the second phase, we merge vectors representing clones with "equivalent" effects. In the third phase, we actually transform the code until program growth exceeds some threshold. This section provides more detail on each of these phases.

4.1 Phase 1: Calculating Cloning Vectors

4.1.1 Motivation

The first phase explores all cloning opportunities that will sharpen the results of a particular forward interprocedural data-flow analysis problem. The algorithm essentially solves the data-flow problem with one important change. Rather than conservatively approximating information when multiple paths converge (*i.e.,* applying a meet function), it retains the unique interprocedural sets contributed by each path.

The propagation of cloning information proceeds in topological order in a single pass over the call graph, with cycles in the graph handled specially. By structuring the analysis in this way, the effects of a cloning decision at a procedure will reach its descendants by the time their cloning opportunities are evaluated.

A key aspect of the Phase 1 algorithm is that it considers potential cloning opportunities but does not actually clone. This feature allows us the freedom to merge potential clones in subsequent phases after we have evaluated their impact on optimization.

4.1.2 Definitions

A few definitions are needed before presentation of the algorithm in Figure 4. Each potential clone can be represented by a *cloning vector,* a unique solution to the data-flow problem being used as the basis for cloning. A cloning vector may represent either information obtained by propagating along a single path through the call graph or information along multiple paths that result in the same solution. As an example, a cloning vector for interprocedural constant propagation would be a set of $\langle variable\ name, constant\ value \rangle$ pairs.

In the algorithm, S identifies the interprocedural problem being used as the basis for cloning. The set *CloningVectors(S, p)* contains, upon termination, the collection of cloning vectors for problem S at the node in the graph representing procedure p. Thus, the goal of the algorithm is to compute *CloningVectors(S, p)* for a particular problem S and for every procedure p in the program.

The function *Translate(c, cv)*, for some call site c with caller p and callee q, maps elements in the vector cv of p to the corresponding variables in q based on parameter passing at c. The result is the creation of a new *CloningVector* set for q. This mapping function is similar to the one used in the underlying interprocedural problem to map variables from the caller to the callee.

Another function *Filter(cv,S,p)* extracts from the cloning vector only the data-flow facts that will be important to optimization. The existence of a filtering function makes the cloning approach goal-directed. Note, however, that the filter can simply be an identity function on cv. For the constant propagation example, the filter would reduce the set of constants to only include values for variables appearing in *ImportantVariables*(p), as described in Section 3.2.2.

4.1.3 Propagation Algorithm

/* Initialization */
Locate cycles and replace with representative nodes and edges
foreach node n in representative graph
  $CloningVectors(S, n) \leftarrow \emptyset$

/* Propagation */
foreach node n in topological order
  foreach call site c invoking n
    let p represent the procedure invoking n at c
    foreach vector cv in $CloningVectors(S, p)$
      $CloningVectors(S, n) \leftarrow CloningVectors(S, n) \cup\ Filter(Translate(c, cv), S, p)$
  if n represents a recursive cycle **then**
    foreach vector cv in $CloningVectors(S, n)$
      Iteratively propagate cv within procedures in cycle until information stabilizes

Figure 4 Phase 1 – Calculating *CloningVectors*.

The algorithm computes and propagates *CloningVectors* sets in topological order. At each propagation step, the *Translate* function renames the variables in each cloning vector according to the parameter passing at its corresponding call. Since *CloningVectors*(`main`) is initialized to $\emptyset$, *Translate* adds facts to the sets for procedures called from the `main` routine. The *Filter* function extracts from the vector returned by *Translate* only the important data-flow facts. The union ($\cup$) only adds the resulting vector to the *CloningVectors* collection if it is unique.

For call graphs containing cycles, usually representing recursion, we locate strongly-connected regions and replace the cycle with a representative node. When the algorithm reaches a representative node, it propagates each incoming cloning vector within the nodes in the cycle until the information stabilizes [17]. This approach to cycles has two important benefits. Most importantly, it prevents the algorithm from analyzing the clones generated by unrolling the recursion, which can generate an infinite number of possible clones. Secondly, it enables the algorithm to complete in only a single pass over the nodes in the reduced graph. For a node representing a cycle with multiple cloning vectors, the final phase of the cloning algorithm will replicate all nodes in the cycle for each incoming cloning vector. Thus, our simple approach to recursion will generate clones when the cycle contains unique initial values that remain constant within the body of the cycle.

This phase of the cloning algorithm can generate an exponential number of cloning vectors. In practice we have not seen this behavior and do not expect to encounter it. Section 5 presents an argument that the number of cloning vectors is polynomial under a plausible set of assumptions. Section 6 describes a strategy for restricting the number of cloning vectors that it actually generates.

4.2 Phase 2: Merging Equivalent Cloning Vectors

4.2.1 Motivation

The previous phase produces the *CloningVectors* collections that represent all the interesting opportunities for cloning in the program. If we have filtered the information in the cloning vectors to consider only important data-flow facts, these sets may fairly precisely indicate the clones that must be produced to perform the targeted optimizations. However, for certain interprocedural problems including constant propagation, it is still possible for two unique cloning vectors to produce the same effect on optimization.

For example, we may be interested in the result of a control flow test that compares whether a variable's value is greater than 1. There may be many different constant values for that variable represented among the cloning vectors that are greater than 1, all effecting the same simplification of the control flow test. Thus, rather than the value of the variable, we would really like to capture during cloning the effects distinct values have on optimization.

The second phase of the cloning algorithm locates cloning vectors producing equivalent effects on optimization and merges them. Identifying that two cloning vectors are equivalent requires a goal-directed strategy. It is necessary to locate specific targets of optimization, so that the effects of a particular cloning vector on the targets of optimization can be ascertained. Then, the effects of a pair of cloning vectors can be compared to determine if they are equivalent.

The following discussion presents an algorithm for merging equivalent cloning vectors of interprocedural constants. A similar approach could be taken for related problems, like type analysis. In Section 4.2.4, we characterize the interprocedural data-flow problems for which this phase is necessary.

4.2.2 Definitions

Suppose we want to evaluate the effects a particular cloning vector of constants has on the important expressions appearing in a procedure, *i.e.,* the control flow tests, subscript expressions and array dimensions. We use *jump functions* to describe the value of each such expression as a function of the external variables for which constant propagation and cloning may potentially uncover a constant value [7]. A jump function, $J_{E1}(f)$, describes the value of expression $E1$ as a function of external variable f. These jump functions can be constructed by examination of the procedure prior to any interprocedural analysis.

By applying the values described by a cloning vector to the jump functions for a procedure's important expressions, we arrive at a *state vector*. $StateVector(p, cv)$ maps a cloning vector cv for procedure p to values of important expressions in p. It is the state vector that captures how the values in a cloning vector will affect optimization within the procedure.

The example in Figure 5 illustrates these points. The Phase 1 algorithm calculates the *CloningVectors* collections for procedures p and q. Procedure p does not contain any important expressions; procedure q contains three of them, $E1$, $E2$ and $E3$. By applying the values represented by three incoming cloning

program *main*
  call $p(10, 1)$
  call $p(10, 2)$
  call $p(10, 3)$

subroutine $p(f_1, f_2)$
  call $q(f_1, f_2 + 4)$

subroutine $q(f_1, f_2)$
$E1$: **dimension** $A(f_1, 1)$
$E2$: **if** $(f_2 \ mod \ 2 = 1)$ **then** ...
$E3$: $A(f_1 + 2, 1) = \ldots$

Jump functions for q:
$J_{E1}(f_1) = f_1$
$J_{E2}(f_2) = (f_2 \ mod \ 2 = 1)$
$J_{E3}(f_1) = f_1 + 2$

$$\mathit{CloningVectors}(\text{CONSTANTS}, \ p) = \{ \ \{\langle f_1, 10\rangle, \langle f_2, 1\rangle\}, \quad \{\langle f_1, 10\rangle, \langle f_2, 2\rangle\}, \quad \{\langle f_1, 10\rangle, \langle f_2, 3\rangle \ \} \ \}$$
$$\mathit{CloningVectors}(\text{CONSTANTS}, \ q) = \{ \ \{\langle f_1, 10\rangle, \langle f_2, 5\rangle\}, \quad \{\langle f_1, 10\rangle, \langle f_2, 6\rangle\}, \quad \{\langle f_1, 10\rangle, \langle f_2, 7\rangle \ \} \ \}$$

$$\mathit{StateVector}(p, \{\langle f1, 10\rangle, \langle f2, 1\rangle\}) = \mathit{StateVector}(p, \{\langle f1, 10\rangle, \langle f2, 2\rangle\}) = \mathit{StateVector}(p, \{\langle f1, 10\rangle, \langle f2, 3\rangle\}) = \emptyset$$

$$\mathit{StateVector}(q, \{\langle f1, 10\rangle, \langle f2, 5\rangle\}) = \langle 10, true, 12\rangle$$
$$\mathit{StateVector}(q, \{\langle f1, 10\rangle, \langle f2, 6\rangle\}) = \langle 10, false, \bot\rangle$$
$$\mathit{StateVector}(q, \{\langle f1, 10\rangle, \langle f2, 7\rangle\}) = \langle 10, true, 12\rangle$$

Figure 5 Example illustrating *StateVector* calculation.

vectors for procedure q to the jump functions for q's important expressions, we obtain only two distinct state vectors.

4.2.3 Partitioning Algorithm

The algorithm for merging equivalent cloning vectors appears in Figure 6. It is related to the algorithm for minimizing the number of states in a Deterministic Finite Automaton (DFA) [14]. It is also similar to an algorithm used to minimize the number of implementations of a procedure required when multiple definitions of the same procedure occur in a program [11].

The algorithm partitions the cloning vectors for a procedure according to the values for their state vectors. It begins by assuming all cloning vectors for a procedure are equivalent. It proceeds to distinguish between cloned versions of a procedure based on their state vector and the partitioning of procedures they invoke. Two clones can be merged if they have the same *StateVector* mapping, and for corresponding call sites in the cloned versions, the invoked procedures are in the same partition of cloning vectors. Upon termination of the algorithm, clones remaining in the same partition can be merged and represented by a single implementation. Nodes are visited in a single reverse topological pass so that the clones of a procedure have been partitioned

1. Initially, all *cloning vectors* for a particular procedure are placed in the same partition.
2. In reverse topological order, visit the partition π corresponding to each node n:
   (a) Partition cloning vectors v_i of π based on the value of *StateVector*(n, cv_i).
   (b) For each partition π_i of π consisting of multiple cloning vectors:

   Form partitions of elements of π_i such that if two *cloning vectors* a and b in π_i result in invocations at some call site c with *cloning vectors* x and y of the called procedure, then a and b are in different partitions if x and y are in different partitions.

Figure 6 Phase 2 – Merging *CloningVectors*.

before any of its callers are considered. In this algorithm as in the previous one, recursion is handled by considering a cycle in the call graph as a single procedure unit. The jump functions and state vector calculations must summarize all the procedures in the cycle.

To clarify the algorithm presentation, consider the example in Figure 5. Procedure q has three unique cloning vectors. Partitioning these according to state vector values results in two partitions, one partition for the cloning vector $\{\langle f_1, 10\rangle, \langle f_2, 6\rangle\}$ and another partition containing the remaining two cloning vectors. Proceeding to partition the cloning vectors for p, there are three distinct cloning vectors. Each one generates the same state vector. However, $\{\langle f_1, 10\rangle, \langle f_2, 2\rangle\}$ is placed in its own partition since it invokes a partition of q that is separate from that invoked by the other two partitions of p.

This approach is very similar to work by Ruf and Weise, but differs in two important ways. First, while both algorithms consider the state resulting from a cloning decision, their algorithm does not perform the state minimization over the program. It would presumably maintain separate specializations when two copies have function calls passing different parameters, even if the net effect results in identical specializations in all descendant procedures. Second, by targeting specific points of interest, our approach can use significantly less space than maintaining information about each statement.

4.2.4 When is Phase 2 Necessary?

Consider why this phase is necessary for interprocedural constant propagation. The unbounded number of potential constant values for a variable appearing in an important expression makes it impossible to enumerate all possible values and determine which ones are important. Instead we locate the variables involved in important expressions prior to cloning. We evaluate only the constant values that appear in the cloning vectors. If two cloning vectors have unique constant values that produce the same effect on optimization, they are merged.

Other interprocedural problems exist for which this merging phase is unnecessary. As an example, we briefly consider cloning based on alias analysis[2]. Two variable names are aliases in a procedure if they can refer to the same memory location. A compiler uses alias information to verify the safety of certain optimizations. The two data-flow analysis problems differ in that the lattice for constant propagation is infinite (*i.e.,* has an unbounded number of possible set values), while the lattice for the alias problem is finite. Thus, the most precise approach to filtering aliases — enumerating all the possible aliases and evaluating

their effects on optimization — is tractable because the lattice is finite. The algorithm can perform this enumeration prior to analysis for cloning and use the resulting important aliases to filter the cloning vectors in Phase 1, eliminating the need for the Phase 2 algorithm.

For lattices with a reasonably small number of values, it may be practical to filter the cloning vectors as they are produced. However, when the underlying data-flow problem has an infinite lattice, the merging phase is both necessary and practical. In fact, it may also be desirable in cases where the lattice is finite but large enough to make enumeration expensive.

4.3 Phase 3: Perform Cloning

4.3.1 Motivation

As suggested in Section 4.1, we expect the number of cloning vectors to be polynomial. If all of this cloning were performed, the final program size could be a polynomial of its original size. The polynomial bound on the number of cloning vectors is acceptable during analysis, but a polynomial growth in program size may be intolerable due to its effects on compile time. Thus, as an additional safeguard to the costs of cloning, we only clone until program growth exceeds some threshold. However, we expect the desired amount of cloning will be achieved for most programs without exceeding the threshold.

4.3.2 Final Cloning Algorithm

```
originalSize ← programSize
foreach node n in topological order
    CloneProcedure(n)
    if (programSize > originalSize * threshold) then
        exit
endfor

CloneProcedure (n)
    foreach partition π_n of n
        let p be the procedure (or procedures) represented by n
        create a copy newp of the procedure p
        annotate representation of newp in the call graph with the set of CloningVectors in π_n
        identify the call sites in the call graph invoking newp
        programSize ← programSize + newp.size
    endfor
end /* CloneProcedure */
```

Figure 7 Phase 3 – Transforming program.

The final phase, given in Figure 7, performs the cloning indicated by the partitions of cloning vectors produced in the previous step. The algorithm clones until the program size reaches some threshold factor of its original size. Since decisions at a procedure are affected by cloning of its ancestors in the call graph, it is critical that the cloning be performed in topological order. An ideal ordering of cloning decisions would also take into account how a decision would affect performance. A simple approach is to estimate the execution frequency of procedures and perform cloning along paths leading to the most frequently executed

procedures [12]. We could also use a strategy similar to the merging of vectors for an individual procedure (see Section 6).

We need to do two things for the newly created clone. First, we annotate *newp* with the cloning vectors from its partition; these annotations direct the optimizer to apply the desired optimizations. Second, we locate in the call graph the call sites formerly invoking p that should now be invoking *newp*. We identify these call sites by examining the cloning vectors associated with each caller. This step allows the compiler to rename the procedure invoked at the call to now invoke *newp*. As a final point, when the algorithm reaches a node representing a cycle in the graph, it will clone all the procedures in the cycle.

5 Time Complexity

Phase 1. In the algorithm from Figure 4, the outer loop iterates over procedures, and the inner loop iterates over cloning vectors at a call site. Assume the maximum number of elements in a cloning vector is L, and the maximum number of values for each element is V. N is the number of procedures in the program, and E is the number of call sites. Then, the algorithm is bounded by $O((N+E)V^L)$ time.[†]

The actual sizes of V and L depend on the interprocedural set being used and the possible values of the set elements. Since we are dealing with interprocedural information, the size of L is related to the number of externally accessible variables in the scope of the procedure. This is the number of formal parameters of a procedure and global variables in the program. Based on experience, the number of formals is a small constant and the number of globals increases more slowly than program growth. For the sake of this presentation, let us assume that this number is bounded by $c \log N$.

Let v_i be the number of distinct values that the i^{th} element in a *CloningVector* can have. For each v_i, there is a k_i such that $2^{k_i-1} < v_i \leq 2^{k_i}$. For a given procedure, an upper bound on the number of unique *CloningVectors* is defined by the following equation:

$$\prod_{i=1}^{L} 2^{k_i} = 2^{\sum_{i=1}^{L} k_i}.$$

Taking the average of k_i over its L possible values, we arrive at some value k_p. 2^{k_p} gives an average number of values for each element, so 2^{Lk_p} is an upper bound on the number of cloning vectors for procedure p. Assuming $L \leq c \log N$, we know that the number of cloning vectors for a procedure $p \leq 2^{k_p(c \log N)}$. The total number is bounded by the following:

$$\sum_{p=1}^{N} 2^{k_p(c \log N)} \leq N * 2^{k_{max}(c \log N)} = N^{ck_{max}}.$$

Here, k_{max} is the maximum value of k_p over all procedures p. Thus, given reasonable values for L and k_{max}, the complexity is $O((N+E)N^{ck_{max}})$, a polynomial.

[†]When a program contains recursive cycles, propagating cloning vectors within the nodes in the cycle contributes a factor I, where I is the number of times the iterative algorithm visits a node in the cycle. This factor is ignored in subsequent discussion because it is completely dependent on the interprocedural problem being solved. However, the amount of iteration required for cloning vector propagation will not be worse than that required by the underlying data-flow problem.

Phase 2. Cloning vectors are partitioned in a single reverse pass over the call graph. Assume that the *StateVector* representation is a string with some canonical order imposed on its elements. If we test for equality by hashing the strings, the partitioning step for each procedure has an expected time linear in the number of its cloning vectors. (An approach based on state minimization would yield $O(n \log n)$ time, even for worst-case performance [14].)

Phase 3. The final phase of the cloning algorithm is accomplished by a single top-down pass over the call graph. The number of clones created is less than the total number of cloning vectors. Thus, Phase 3 is also bounded by the number of cloning vectors.

Given that the time required by each of the phases is bounded by the total number of cloning vectors, the entire algorithm has an expected time complexity of $O((N + E)N^{ck_{max}})$.

6 Rationing Cloning Vectors

We have argued that real programs will produce a polynomial number of cloning vectors; in practice, we expect the number to be manageable. Nonetheless, it is still possible for programs to produce impractically large numbers of cloning vectors. When the compiler encounters such a program, the cloning algorithm must be prepared to limit the number of vectors stored and propagated. A practical approach to this problem is to adopt a rationing scheme for cloning vectors. When the quota of vectors is exceeded, the algorithm should begin merging vectors as they are produced.

We can define the *opportunity cost* of merging two cloning vectors cv_i and cv_j as a measure of the effect that the merge will have on optimization. The opportunity cost must account for improvements enabled by information exposed by cv_i and not by cv_j, and vice versa. Having a metric to compare vectors is crucial to the rationing scheme.

Several strategies are possible to determine the opportunity cost of merging two cloning vectors. As a possibility, albeit an unrealistic one, we can compile and run two versions of the program. One program maintains the separate versions of the procedure, while the other merges them. The opportunity cost is the difference in execution time of the two program versions. We would like to approximate this approach using static analysis to estimate the opportunity cost. For example, it can be the number of positions that differ between a pair of cloning vectors. This stategy can be improved by taking into account execution frequency estimates and weighting the effects of each piece of information [1].

Given a method to compute the opportunity cost of merging two cloning vectors, the compiler can adopt a relatively simple rationing scheme. Assume that we set a quota for the total number of cloning vectors allowed during compilation and a quota for each procedure. The overall quota should be proportional to the number of procedures, the individual quotas should be set somewhat higher than the overall quota divided by the number of procedures. When propagation attempts to create a vector for procedure p that would exceed either the local or the global quota, the algorithm either

1. merges v into an existing partition, or

2. merges two lower profit classes and keeps v as a new partition.

In implementing this scheme, an efficient means of incrementally comparing and merging vectors based on opportunity costs is needed. A number of schemes suggest themselves, including clever application of string matching algorithms, representing the set of retained cloning vectors as a prefix tree, and simply keeping the k partitions with largest estimated improvement. Since our experience suggests that merging clones will not be necessary beyond what occurs in Phase 2 of the algorithm (Section 4.2), we believe merging two arbitrary vectors will be sufficient when pathological cases do arise.

7 Summary and Conclusions

This paper has described an algorithm for deciding how to clone a program for improved optimization. This general approach bases cloning on any forward interprocedural data-flow analysis problem. The three-phase algorithm explores potential cloning opportunities, merges potential clones that produce equivalent effects on optimization and, finally, performs cloning of the program. Full cloning of a program may potentially lead to exponential execution time and program growth; however, we have argued, based on the characteristics of interprocedural data-flow sets, that it is actually bounded by a polynomial of program size. Nevertheless, in the event of pathological behavior, we suggest mechanisms to reduce the amount of cloning to a manageable size.

The algorithm was designed in the context of the program compiler for the ParaScope programming environment – the tool that manages interprocedural issues in compilation [5]. This general algorithm supports a number of emerging applications for cloning. These applications come from diverse areas: compiling for scalar architectures, compiling for both shared-memory and distributed-memory parallel architectures, and instrumenting code for run-time detection of race conditions in shared-memory parallel programs. To date, we have effectively employed cloning in experiments with interprocedural constant propagation [3, 12] and interprocedural transformations for parallel code generation [13] through hand optimization and a partial implementation of the algorithm.

ParaScope is devoted to high-performance Fortran programming, but the need for cloning arises in many other contexts such as those discussed in Section 2. Experimentation is needed to verify that the assumptions used in our algorithm generalize to these other contexts.

Acknowledgements. The authors wish to thank several people who contributed to this work. Research collaborations with Preston Briggs and Linda Torczon helped motivate this work. Conversations with Ben Chase, Urs Hoezle, Kathryn McKinley, Doug Moore, Bill Noyce, Bob Rao and Alejandro Schaffer provided useful insights that were incorporated into this work. Paul Havlak suggested several improvements after reading a draft of this paper. The ParaScope group at Rice University has provided a particularly useful environment for examining problems of this kind.

References

[1] Ball, J.E. Predicting the effects of optimization on a procedure body. *ACM SIGPLAN Notices*, 14(8):214–220, 1979.

[2] Banning, J.P. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the Sixth Annual Symposium on Principles of Programming Languages*, pages 29–41. ACM, January 1979.

[3] Briggs, P., Cooper, K.D., Hall, M.W., and Torczon, L. Goal-directed interprocedural optimization. Technical Report TR90-148, Dept. of Computer Science, Rice University, November 1990.

[4] Bulyonkov, M.A. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[5] Callahan, C.D., Cooper, K.D., Hood, R.T., Kennedy, K., and Torczon, L. ParaScope: a parallel programming environment. *International Journal of Supercomputer Applications*, 2(4):84–89, 1988.

[6] Callahan, D., Carr, S., and Kennedy, K. Improving register allocation for subscripted variables. *ACM SIGPLAN Notices*, 25(6):53–65, 1990.

[7] Callahan, D., Cooper, K.D., Kennedy, K., and Torczon, L. Interprocedural constant propagation. *ACM SIGPLAN Notices*, 21(7):152–161, 1986.

[8] Chambers, C. and Ungar, D. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. *ACM SIGPLAN Notices*, 24(7):146–160, 1989.

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

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

[11] Cooper, K.D., Kennedy, K., Torczon, L., Weingarten, A., and Wolcott, M. Editing and compiling whole programs. *ACM SIGPLAN Notices*, 22(1):92–101, 1987.

[12] Hall, M.W. *Managing Interprocedural Optimization*. PhD thesis, Rice University, Department of Computer Science, Houston, TX, April 1991.

[13] Hall, M.W., Kennedy, K., and McKinley, K.S. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*. IEEE Computer Society, November 1991.

[14] Hopcroft, J. An *nlogn* algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196. New York: Academic Press, 1971.

[15] Johnston, R.L. The dynamic incremental compiler of APL\3000. In *Proceedings of the APL '79 Conference*, pages 82–87. ACM, June 1979.

[16] Kuck, D.J. *The Structure of Computers and Computations*, volume 1. New York: John Wiley and Sons, 1978.

[17] Marlowe, T.J. and Ryder, B.G. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the Seventeenth Annual Symposium on Principles of Programming Languages*, pages 184–196. ACM, January 1990.

[18] Ruf, E. and Weise, D. Using types to avoid redundant specialization. *ACM SIGPLAN Notices*, 26(9):321–333, 1991.

[19] Wegman, M. *General and Efficient Methods for Global Code Improvement*. PhD thesis, University of California, Berkeley, CA, December 1981.

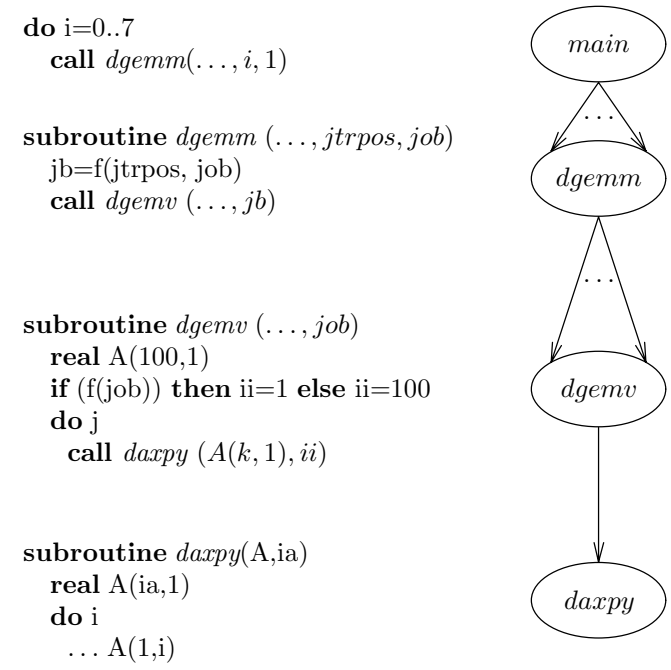


Figure 1 Call graph for matrix300.

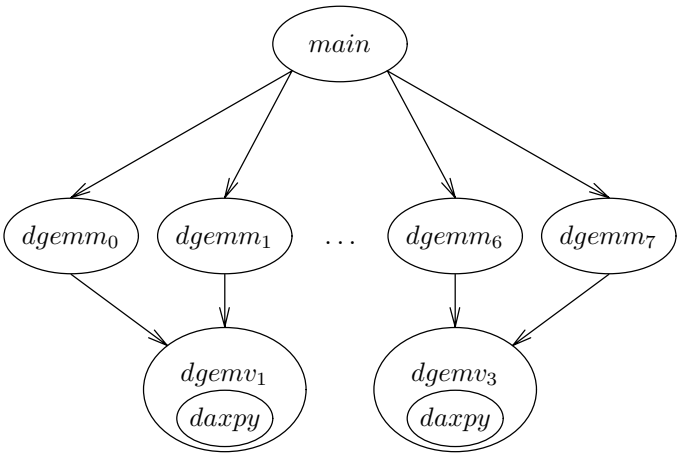
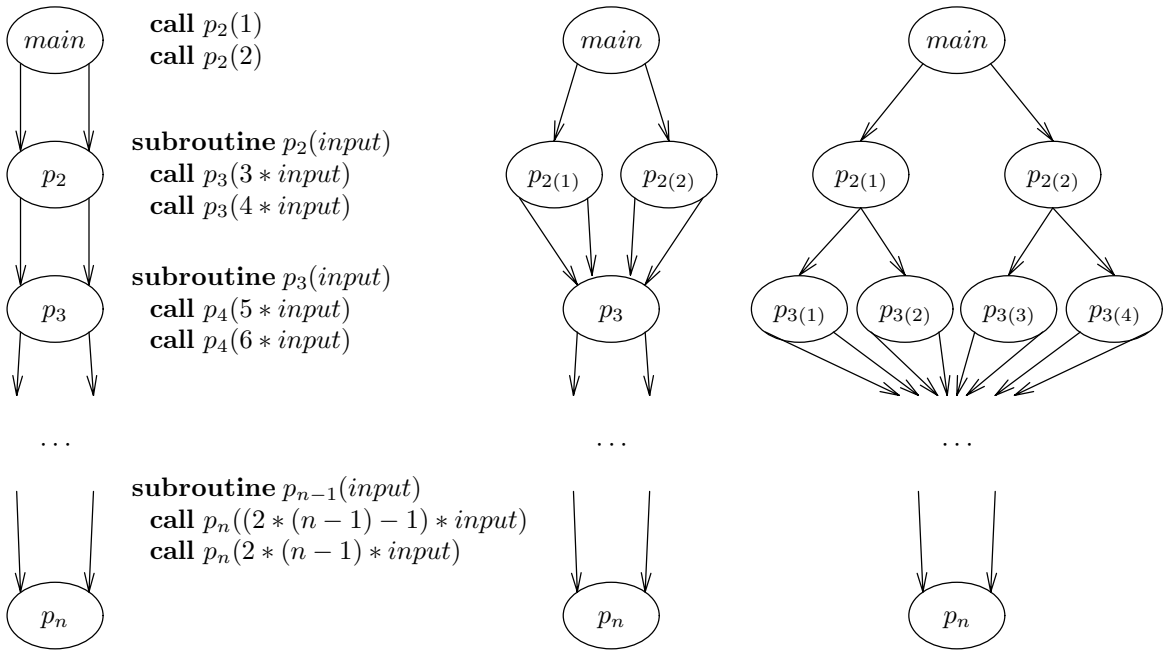


Figure 2 Call graph for matrix300 after cloning and inlining.



(a) Initial program (b) After cloning p_2 (c) After cloning p_3

Figure 3 Exponential code growth due to cloning.

About the Author — KEITH D. COOPER is an associate professor in the Department of Computer Science at Rice University. His research interests include interprocedural analysis and optimization, code optimization for advanced microprocessors, and advanced programming environments. He was graduated from Rice University in 1978 with a B.S. in electrical engineering. He received an M.A. from the Department of Mathematical Sciences at Rice in 1982 and a Ph.D. from the same department in 1983.

About the Author — MARY W. HALL is currently a research associate in the Electrical Engineering Department at Stanford University. Her research interests are in interprocedural compiler optimizations, parallelizing compilers and programming environments. She was graduated magna cum laude from Rice University in 1985, with a B.A. in Computer Science and Mathematical Sciences. She received an M.S. in Computer Science in 1989, and a Ph.D. in Computer Science in 1991, both from Rice. From October, 1990, to May, 1992, she was a research scientist in the Center for Research on Parallel Computation at Rice.

About the Author — KEN KENNEDY is a Noah Harding Professor of Computer Science at Rice University, Director of the Computer and Information Technology Institute at Rice and heads the Center for Research on Parallel Computation (CRPC), an NSF Science and Technology Center at Rice, Caltech, Los Alamos National Lab, Argonne National Lab, the University of Tennessee and Syracuse University. Professor Kennedy's current research focuses on extending techniques developed for automatic vectorization to programming tools for parallel computer systems and high-performance microprocessors. Through the CRPC, he is seeking to develop new strategies for supporting architecture-independent parallel programming, especially in science and engineering. Kennedy received the B.A. degree in mathematics from Rice University in 1967, the M.S. degree in mathematics and the Ph.D. degree in computer science from New York University in 1969 and 1971, respectively. He has been on the faculty at Rice since 1971.

Abbreviated Summary

This paper has described an algorithm for deciding how to clone a program for improved optimization. This general approach bases cloning on any forward interprocedural data-flow analysis problem. The three-phase algorithm explores potential cloning opportunities, merges potential clones that produce equivalent effects on optimization and, finally, performs cloning of the program. Full cloning of a program may potentially lead to exponential execution time and program growth; however, we have argued, based on the characteristics of interprocedural data-flow sets, that it is actually bounded by a polynomial of program size. Nevertheless, in the event of pathological behavior, we suggest mechanisms to reduce the amount of cloning to a manageable size.

The algorithm for procedure cloning was designed in the context of compiling high-performance scientific Fortran. While the paper focuses on cloning to expose better constants, we have found many other important applications of the algorithm in our system. It is clear that cloning is also useful in other contexts, including intraprocedural optimization, partial evaluation and dynamic compilation discussed in the Related Work section of the paper. Experimentation is needed to verify that our assumptions bounding the algorithm are valid in these other contexts.