International Journal of Parallel Programming, *Vol. 26, No. 1, 1998*

Initial Results for Glacial Variable Analysis

Tito Autrey[1] and Michael Wolfe[1]

Runtime code generation that uses the values of one or more variables to generate specialized code is called value-specific optimization. Typically, value-specific optimization focuses on variables that are modified much less frequently than they are referenced; we call these *glacial variables*. In current systems that use runtime code generation, *glacial variables* are identified by programmer directives. Next, we describe *glacial variable analysis*, the first data-flow analysis for *automatically* identifying glacial variables. We introduce the term *staging analysis* to describe analyses that divide a program into stages or use the stage structure of a program. Glacial variable analysis is an interprocedural staging analysis that identifies the relative modification and reference frequencies for each variable and expression. Later, several experiments are given to characterize a set of benchmark programs with respect to their stage structure, and we show how often value-specific optimization might be applied. Finally, we explain how staging analysis relates to runtime code generation, briefly describe glacial variable analysis and present some initial results.

KEY WORDS: Glacial variable analysis; staging analysis; candidate variables; value-specific optimization.

1. INTRODUCTION

Specialization is tailoring general-purpose procedures to a specific invocation or set of invocations. Specialization takes advantage of specific values of parameters. For the same function, specialized code is frequently an order of magnitude faster than general-purpose code. Functional language compilers traditionally use partial evaluation to produce specialized versions

[1] Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology.

0885-7458/98/0200-0043$15.00/0

of functions.[1] Imperative language compilers traditionally use procedure cloning to perform the same optimization.[2] Value-specific optimization (VSO) uses runtime code generation (RTCG) to produce specialized code at runtime instead of compile-time.[3]

By using the values of slowly changing variables, VSO ensures that the number of invocations of each chunk of specialized code is high and therefore that the cost of the optimization is recovered. We refer to these variables as *glacial variables* because they vary slowly. (Note: Keppel refers to these variables as *candidate variables*).[4] Glacial variables are modified infrequently compared to the frequency with which they are used. Current approaches to compiler-controlled VSO require the programmer to insert directives into the program source code to identify glacial variables.[5–8]

A *staging analysis* is an analysis that partitions a program into chunks larger than a basic block where the partitioning criterion is related to the order in which computations happen in a program. The large, ordered program chunks can be used to focus other analyses and optimizations where they are most effective.

In our model, each loop defines a stage. We use loops to define stages because historically they are considered good indicators of execution frequency. Execution frequency translates directly into reuse of the code of a stage and in many cases the reuse of values referenced within the stage body as well. We label every basic block in the body of a loop with a stage-level equal to the loop depth of the containing loop. A level i stage is indicated by Stage-i. A level $(i+1)$ stage is a *sub-stage* of a level i stage if it is contained within that stage. Correspondingly, the level i stage is the *super-stage* of that level $(i+1)$ stage. Sub-stages execute more frequently than their super-stages.

Glacial variable analysis (GVA) is a staging analysis that automatically identifies glacial variables using two pieces of information. The first is an estimate of the execution frequency of each block of code. The second is an estimate of the modification frequency of each expression and subexpression. The *degree of glacialness* of an expression (variable) is a measure of the difference between its execution (use) and modification (def) frequencies.

Figure 1a shows the labeling of some example code. In our model, RTCG can be applied at the boundary between a level i stage and a level $(i+1)$ stage. Value-specific optimization generates a specialized version of the level $(i+1)$ stage using the values of glacial variables available at that point in the level i stage. By definition these variables aren't modified within the higher level stage; For example, if there is a subexpression in a level $(i+1)$ stage that uses values only from its superstage (Stage-i) then the superstage is extended with a call to a runtime code generator. The code generator evaluates the subexpression and generates code specialized

with the *resulting values* of that evaluation. The code is a specialized version of the sub-stage (Stage-(i+ 1)).

Figure 1b shows an integer matrix–matrix multiply procedure; at exit C is the matrix product of A and B. Inspection (or a simple data-flow analysis) shows that A and B are referenced, but not modified, within the loop nest and therefore have three degrees of glacialness in the k-loop. In addition, portions of the array subscripting expressions are glacial. A[i,1..n] has two degrees of glacialness and B[1..n,j] has one degree of glacialness. Because A is not modified, there are no data-dependence constraints on accesses to it, therefore we can generate code for the k-loop specialized to the subscripting expression A[i,?]. The code is generated at the point marked "*" so a new inner loop is generated each time around the i-loop. The specialized code is the fully unrolled innermost loop, so the loop overhead instructions have been eliminated. In many cases A may have small enough values such that at code-generation time the load can be eliminated in favor of instruction immediates. (Note: RISC ISAs tend to support 16-bit immediates for arithmetic operations). The embedding requires full loop-unrolling. If the arrays are laid out in column-major order then the embedding can have a beneficial effect on cache utilization by removing the large-stride accesses to A. Algebraic identities allow the entire load B-multiply-add-store C sequence to be skipped when A[i,k] is 0. For a sparse matrix this optimization is significant. All this work pays off because the runtime generated code is used n^2 times. (Note: When full loop-unrolling is used, the code is only used n times).

The rest of the paper is organized as follows: in Section 2 we present related work; in Section 3 we discuss glacial variable analysis in more detail; in Section 4 we present experimental results from applying glacial

```
...                     Stage-0
do i1                   Stage-1
  a[i1] = ...           Stage-1
  do i2                 Stage-2
    b[i2] = ...         Stage-2
    do i3               Stage-3
      c[i3] = ...       Stage-3
    end                 Stage-3
  end                   Stage-2
  do i4                 Stage-2
    d[i4] = ...         Stage-2
  end                   Stage-2
end                     Stage-1
...                     Stage-0
```

(a)

```
matmult (A, B, C, n)
  do i = 1 to n
  *
    do j = 1 to n
    C[i,j] = 0
      do k = 1 to n
        C[i,j] = C[i,j] + A[i,k] * B[k,j]
      enddo
    enddo
  enddo
end matmult
```

(b)

Fig. 1. (a) Example of computation stages; and (b) Example of how to use VSO.

variable analysis to a set of benchmark programs; and in Section 5 we present conclusions and future work.

2. RELATED WORK

The most closely related work to glacial variable analysis is *binding time analysis* (BTA) which is used by offline partial evaluation.[9] Binding time analysis takes as input a partition of the program inputs labeled with elements from a two-level binding-time lattice: *Static* and *Dynamic*. The analysis labels every expression with *Static* or *Dynamic*. The labeled program is then passed to a partial evaluator, along with a set of values for the static inputs, which produces a *residual* program where all static expressions have been evaluated and replaced with their value. Multi-level binding time analysis extends the binding-time lattice in much the same way that we have extended the constant propagation lattice to obtain our stage-level lattice.[10] The differences are that BTA just labels expressions whereas GVA labels code separately from expressions, and BTA works from an initial partition of the inputs which limits the precision of the labeling within a procedure, whereas GVA uses the control flow structure of the procedure to deduce the most precise labeling possible.

In 1982, Ershov[11] distinguished between what was traditionally called partial evaluation and what Jørring and Scherlis call staging transformations.[12] Today "partial evaluation" is understood to encompass both styles of transformation. Staging transformations are syntactic rules used to move computation from the *late* stage of a program (runtime) into the *early* one (compile-time). Compile-time is viewed as having a cost of zero. Thus extensive compile-time transformations may be performed even if the resulting improvement in the *late* stage is uncertain. We wish to extend the partitioning to more than two stages. In our model, most of the multiple stages are specialized at runtime and incur a runtime cost. Thus we use an analysis phase, *staging analysis*, to suggest where the transformations may likely reduce program runtime rather than increase it through unrecoverable RTCG overhead.

There are several compiler-supported RTCG systems: one based on the Multiflow compiler at the University of Washington;[5] one based on partial evaluation of C at INRIA, Tempo,[6] one based on extensions to C at MIT, tick-C;[7] and one based on SML/NJ at Carnegie-Mellon University, Fabius.[8] All of them rely on programmer annotations to identify candidate variables for VSO. For example, in Leone and Lee's Fabius system, the programmer separates the arguments into an early-tuple and a late-tuple. The compiler inserts runtime code generator calls that use the early-tuple to generate a specialized version of the function which takes the late-tuple

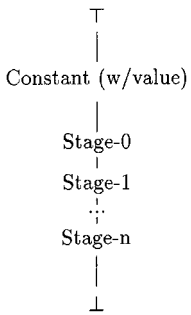
as its argument. With a staging analysis, such as GVA, the programmer would not have to do the argument partitioning by hand. Programmer annotations are prone to errors; both incorrect application and lack of application can lead to suboptimal results.

SELF is a pure object-oriented programming language. The SELF-93 compiler and runtime system monitor method invocations for receiver classes. Method invocations are specialized based on the receiver class in order to improve performance.[13] This specialization is not strictly a VSO (it is a type-based optimization), but it does use RTCG. The SELF-93 compiler does not use interprocedural type analysis as the SELF-91 compiler does;[14] however, it obtains better results by performing the analysis at runtime. We expect that a *glacial type analysis* combined with GRLA could determine at compile-time which method calls to apply receiver class specialization to, instead of checking all method calls at run-time.

```
program main                                  Procedure-CSL=0
  integer factors[n]
  integer matrix[n,n], outmat[n]
  integer vector[n], outvec
  ...
  outvec = 0
  outmat = 0
  read (factors)
  while (P) do                                Block-CSL=1
    read (vector, matrix)
    read (type, P)
    case (type)
      VECTOR:
        sum (vector, factors, outvec)
      MATRIX:
        do i = 1 to n                         Block-CSL=2
          sum (matrix[i], factors, outmat[i])
        enddo
    endcase
  endwhile
end main

procedure sum (in, fact, out)                 Procedure-CSL=2
  out = 0
  do i = 1 to n                               Block-CSL=1
    out = out + in[i] * fact[i]
  enddo
end sum
```

(a)



(b)

Fig. 2. (a) GRLA applied to example code; and (b) Stage-level lattice.

3. GLACIAL VARIABLE ANALYSIS

Glacial variable analysis is a staging analysis composed of two parts. The first, *global recursion-level analysis* (GRLA) [pronounced "gorilla"] identifies the stages inherent in the program and labels each stage with an element from the stage-level lattice shown in Fig. 2b. [The *Constant* element of the lattice is not applied by GRLA, but we want to use the same lattice for both GRLA and GVP]. The second, *glacial variable propagation* (GVP) uses the GRLA results to label variable definitions and expressions, also with elements from the stage-level lattice. The labels generated by GRLA are called *code stage-levels* (CSLs) and those by GVP are called *variant stage-levels* (VSLs).

3.1. Global Recursion-Level Analysis

The purpose of GRLA is to label each block of code with a CSL. The CSL is a measure of the block's estimated frequency of execution. A higher CSL indicates code that is more frequently executed and therefore more important to optimize. A higher frequency of execution indicates a greater level of temporal reuse, of the code. Global recursion-level analysis is a flow-insensitive interprocedural data-flow analysis. [We acknowledge that the use of "global" usually suggests that an analysis is applied to all basic blocks in a procedure.] It uses the program's call graph and the control-flow graph of each procedure.

Global recursion-level analysis partitions the program's computations into stages. A stage is defined as a single or multi-entry loop and its contained body. Note that by definition, any two loops are either nested (one fully contained within the other) or disjoint (having no basic blocks in common). The collection of stages forms a tree where each stage is composed of a sequence of basic blocks interspersed with sub-stages.

The CSL is composed of two parts, a Block-CSL and a Procedure-CSL, which correspond to the two phases of GRLA, the intraprocedural and interprocedural. This separation also supports future work on procedure cloning and inlining. The intraprocedural phase of the labeling is described in Step 1 of the full description GRLA. Here, is a summary of the full interprocedural GRLA:

1. Within each procedure, find all loops and compute loop nest depths. The Block-CSL of a basic block is the loop-depth of the immediately enclosing loop.
2. Compute the complete call graph. We use Hall and Kennedy's algorithm.[15]

3. Assign CSLs to each procedure.

   (a) For the top-level procedure, the Procedure-CSL is zero.

   (b) Otherwise, the Procedure-CSL is the stage-level lattice meet of the CSLs of the call sites that may call that procedure. Each call site CSL is the Procedure-CSL plus the Block-CSL of the block containing the call site. Since the benchmark set is written in Fortran, without recursion, we skip the details of handling recursion in this paper.[2]

Global recursion-level analysis is a forward data-flow problem on a reduced call graph. When there are no cycles in the call graph, all procedures are labeled in a single topological-order pass over the graph. We use MAX as our meet-function. This choice ensures that the deepest possible nesting of stages in the program is labeled with the largest CSL.

The Procedure-CSL is ambiguous if calls to a procedure are at different stage-levels. In Fig. 2a, the main program has a CSL of zero. Calls to **sum** appear at stages one and two within **main**. We assign procedure **sum** a Procedure-CSL of $MAX(1, 2) = 2$. With this choice, the **i**-loop in **sum** is predicted to execute more frequently than the while-loop in **main**, which matches our intuition.

The time complexity of GRLA, in the absence of cycles in the call graph, is $O(|C|)$ for the interprocedural phase, where C is the set of edges in the call graph. The intraprocedural phase is loop-finding. The cost of the multi-entry loop finding algorithm when all loops are natural (single-entry) loops is $O(|N| + |E|)$ where E is the set of edges and N the set of nodes in the control flow graph. This case is by far the most common. If the worst traversal order is used for a graph where the number of multi-entry loops is $O(N)$, then the cost is $O(N^2 * E^2)$, but it is difficult to imagine what the code would look like in this cause.

3.2. Glacial Variable Propagation

The purpose of GVP is to label each variable definition and expression with a VSL. The VSL is a measure of how frequently the value of the variable or expression changes.

[2] Briefly, we discover cycles in the call graph very similarly to the way that multi-entry loop finding[16, 19] discovers cycles in the CFG, except that we use a top-down rather than bottom-up traversal of the graph. The key difference is that during the algorithm execution we choose which DFST edges become back-edges. We want to compute the highest CSLs possible for each procedure.

Glacial variable propagation is a flow-sensitive interprocedural data-flow analysis. We use a static single-assignment (SSA) graph in our compiler intermediate representation. The VSL of a SSA-name is determined by the assignment or pseudo-assignment to that name. The LHS name of an assignment or pseudo-assignment is labeled with the meet of the stage-level lattice elements of the RHS expression after operator folding. For assignments, the operator folding is important for allowing constants to propagate as far as possible. Nonconstant ϕ-functions are labeled with the VSL corresponding to the CSL of their containing block. A ϕ-function represents the dynamic selection of one of several values of a variable based upon the flow of control followed to reach it. This selection happens as frequently as the code is executed, so the VSL should match the CSL. If, due to conditional expression resolution or constant propagation, all defined inputs to a ϕ-function are the same constant, then we want to propagate the most precise information possible, which is that constant. Our SSA form is extended with μ- and η-functions in the manner of gated single assignment.[18, 19] μ-functions are ϕ-functions at loop headers. Thus the VSL of a μ-function is labeled equivalently to a ϕ-function. The μ-function captures the stage-level for variables with definitions that come from before the loop and around the loop back-edge. η-functions capture the final value of variables modified within a loop. They are placed at the target of each loop exit edge. The η-function is labeled with the VSL corresponding to the CSL of the block it is contained in, except in the case where induction variable analysis determines that it is a constant. Since the η-function is in the super-stage of the loop just exited, it has a lower stage-level than a corresponding μ-function. [GOTO statements in or out of loops may require constructing code blocks at particular stage-levels for η-functions or the analysis may be abandoned for such procedures or programs.]

Glacial variable propagation is a modification of Wegman and Zadeck's intraprocedural sparse conditional constant propagation algorithm (WZ).[20] We use a taller lattice, shown in Fig. 2b. The standard constant propagation lattice has one element for *never defined*, ($\top$), one element for each constant, and one for runtime values, ($\bot$). We extend this lattice with

Table I. Equations for Variable Definitions in Glacial Variable Propagation

SSA-node	Glacial variable propagation equation
$a = f(a_{i \in 1 \ldots n})$	$\bigsqcap VSL(a_i)$, or *Constant*
$\phi(b_{i \in 1 \ldots n})$	Stage-$x \mid x = CSL(\phi(b_i))$, or *Constant*
$\mu(c_{i \in 1 \ldots n})$	Stage-$x \mid x = CSL(\mu(c_i))$, or *Constant*
$\eta(d)$	Stage-$(x-1) \mid x = CSL(\mu(d))$, or *Constant*

Table II. Glacial Variable Propagation Meet-Function

Meet	Result
[=] $Constant(v_1) \sqcap Constant(v_2)$	$Constant(v_3)$ where $v_3 = v_1$ *op* v_2 via operator folding (meet does not happen)
[ϕ/μ] $Constant(v_1) \sqcap Constant(v_2)$	$Constant(v_1)$ if $v_1 = v_2$, CSL(block containing the ϕ/μ) otherwise
$Constant \sqcap X$	X
Stage-$i \sqcap$ Stage-j	Stage-$MAX(i, j)$
$\top \sqcap X$	X
$\bot \sqcap X$	$\bot$

```
program main
  integer factors[n]
  integer matrix[n,n], outmat[n]
  integer vector[n], outvec
  ...
  outvec = 0
  outmat = 0
  read (factors^0)
  while (P) do
    read (vector^1, matrix^1)
    read (type^1, P^1)
    case (type)
      VECTOR:
        sum (vector, factors, outvec^1)
      MATRIX:
        do i = 1 to n
          sum (matrix[i], factors, outmat[i]^2)
        enddo
    endcase
  endwhile
end main

procedure sum (in, fact, out^2)
  do i = 1 to n
    out^3 = out + in[i] * fact[i]
  enddo
end sum
```

Fig. 3. GVP applied to example code.

Stage-i elements to represent computation stages. The equations for variable definitions are in Table I. The meet function is shown in Table II. We label arrays as well as scalars. Arrays are treated as whole units, individual elements are not labeled independently. Otherwise the structure of WZ is preserved.

We extend an interprocedural constant propagation algorithm to use the stage-level lattice.[21] Global recursion-level analysis already uses the call graph, so using an interprocedural propagator increases the precision of GVP without increasing the total space requirements for compilation.

In Fig. 3, we show the same code shown in Fig. 2 with VSL labels as bold superscripts. Only the VSLs of definitions are marked. The VSL of a use comes from its reaching definition. The **out** parameter to **sum** is set to VSL-2 at the end of sum. In the while-loop, **outvec** is set to VSL-1 because it can't be modified more frequently than the function invocation is executed.

Glacial variable propagation retains the time-complexity of WZ, $O(|E|+|S|)$, where E is the set of edges in the control flow graph and S is the set of edges in the SSA graph. Although the maximum size of an SSA graph is quadratic in the size of the input program, empirical evidence shows that is linear in practice.[22] Interprocedural constant propagation has complexity $O(|C| * (|E|+|S|))$ where C is the set of edges in the call graph. In practice, there are so few interprocedural constants that the time required is much less than for intraprocedural constant propagation. Since the two main algorithms of glacial variable analysis are linear-time algorithms in practice, it is efficient.

4. INITIAL RESULTS

We describe a series of experiments that use glacial variable analysis to discover the opportunities for applying RTCG. The experiments described here are applied to four programs that have been used as scientific computing benchmarks, BARO (shallow water atmospheric model using finite difference approximation), MHD2D (2-D magnetohydrodynamics using FFTs), SHEAR (hydrodynamics model using FFTs) and VORTEX (vortex sheet model). They have all been written to vectorize easily on a Cray-1. These are some whole programs which our compiler, Nascent, can handle in their entirety. We do not claim that the programs are particularly representative, but they are complete applications that are not known to be favorable for RTCG.

A few details about the Nascent compiler: it builds a complete call graph of the program and it assumes that code for all routines is present. Calls to builtin functions are allowed within a leaf procedure. Typically the

compiler won't have access to the source of a builtin function, but it knows what side-effects, if any, the builtin function has. We distinguish between leaf and nonleaf procedures because we want to show bow widely interprocedural optimization is applicable.

The key result of this paper is that there are many glacial variables in programs such as our benchmarks. The statistics for the benchmarks appear in Section 4.4.

4.1. Call Graph Characterization

Procedures are used to introduce modularity and abstraction into a program. In order to understand how GRLA is affected by the modularity introduced by the programmer, we perform the following experiments:

- Count the number of calls to procedures and correlate with leaf vs. nonleaf procedures. The number of calls indicates whether procedures are used for abstraction or modularity. If a procedure is called from only one place, then it is a modular use (at least in this program). Leaf procedures, by their nature, seem more likely to be abstractions. Analysis of leaf procedures is more precise due to the fact that information is not lost within them due to call sites.
- Count the different CSLs from which a procedure is called and correlate with leaf vs. nonleaf procedures. The number of distinct CSLs indicates how much information is lost by GRLA for computing CSLs in this procedure and its descendants in the call graph.

In Table III, for columns two, four and five, the number on the left is the percentage of all procedures that are leaves with that property and the one on the right is for nonleaves. For column three, the number on the left is one percentage of all calls that are to leaf procedures and the one on the right is for calls to nonleaf procedures. For column six, the numbers are averages and the left and right correspond to leaf and nonleaf procedures

Table III. Percentage of Procedures or Call Sites with a Given Property

Program	Procs	Calls to	With 1 call to	With 1 CLS	Avg No. CSLs
BARO	57/43	50/50	50/0	57/14	1.00/1.67
MHD2D	50/50	60/40	10/7	36/36	1.43/1.29
SHEAR	29/71	33/67	9/3	21/71	1.25/1.00
VORTEX	84/16	89/11	79/11	79/16	1.06/1.00

as do the other columns. For example, the fourth column shows that 79% of all procedures in VORTEX are leaf procedures that are called from only one place. The second column shows that, with the exception of SHEAR, a bit more than half of procedures are leaves. The third column shows that there are more static calls to leaf than nonleaf procedures and that the distribution of call sites matches the distribution of procedures. The fourth column shows that only VORTEX uses procedures mostly for modularity (1 call to a procedure) and that leaf procedures are used rather than nonleaves. We expected more nonleaf procedures to be used for modularity. The fifth column shows that about three-quarters of the procedures are called from one CSL, indicating only a little loss of precision in our Procedure-CSLs. The sixth column shows that on average a given procedure is called from only one or two CSLs, so the loss of precision is not too great.

4.2. Definition Variant Stage-Level Characterization

Glacial variable propagation labels all expressions with their VSL. We would like to understand how many values are produced at each stage-level. This gives us some insight into which program stage-levels are large in terms of the static values generated. The programmer may have broken down large expressions into assignments to temporaries, which makes a stage appear to generate more values than have upward exposed uses beyond the stage. We distinguish between scalars and arrays because scalars are simpler to embed in specialized code than arrays, but arrays contain more values so the potential payoff is larger when they can be used effectively. We perform the following experiment:

- Count the number of definitions and pseudo-definitions (ϕ-, μ- and η-functions) at each stage-level. Distinguish between scalars and arrays.

These experiments show the distribution of modification frequencies. Figure 4 shows the distribution by type of scalar modification and stage level at which they occur of true assignment vs. the several pseudo-assignments. Figure 5 shows the same information, but for arrays. The shaded area in each graph is 100%, except where some of the taller bars are clipped at 50%. As one would expect from the definition of μ- and η-functions, the shapes of their graphs correlate well.

It is important to remember that the stage-level is on a log-scale and that this graph contains static percentages. The tiny percentages in the higher stage-level columns become huge dynamic counts when they are

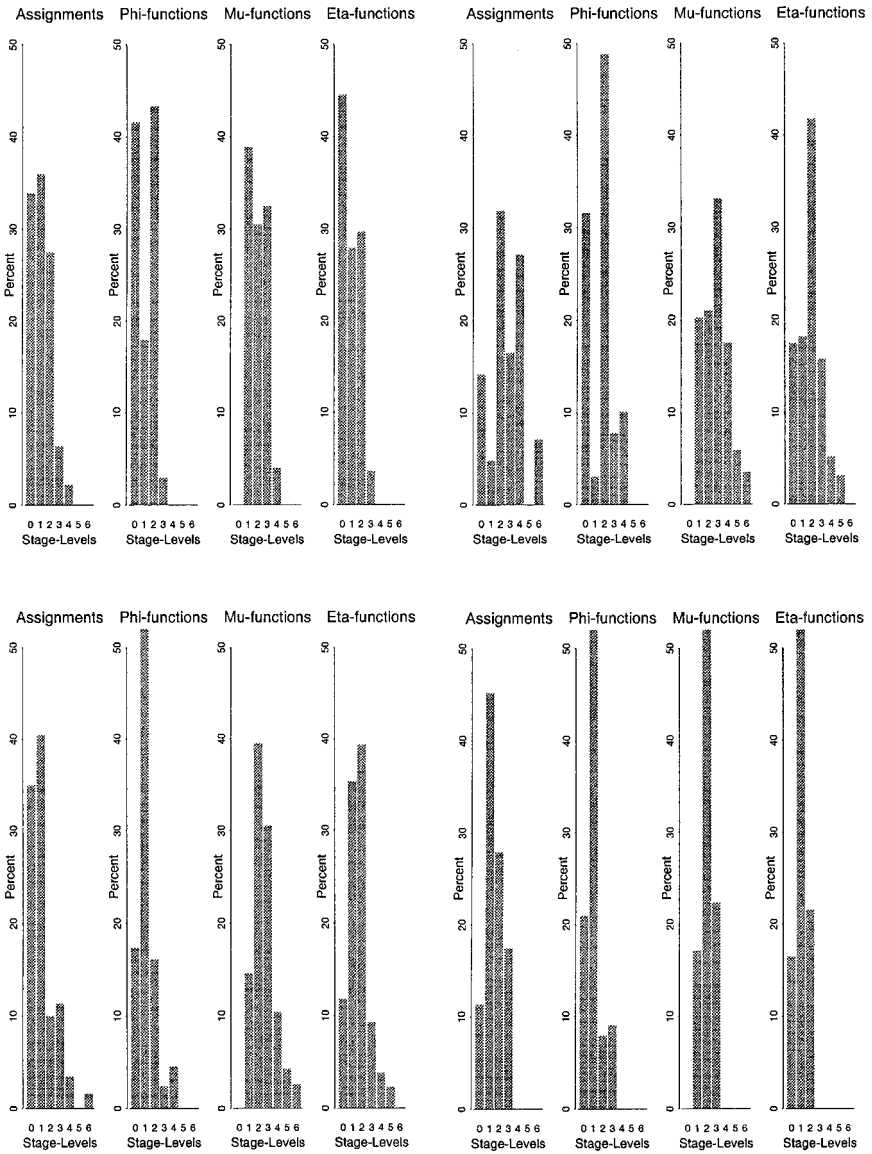


Fig. 4. Histogram of percent of *scalar* definitions of each type at each level (values $W > 50\%$ are clipped).

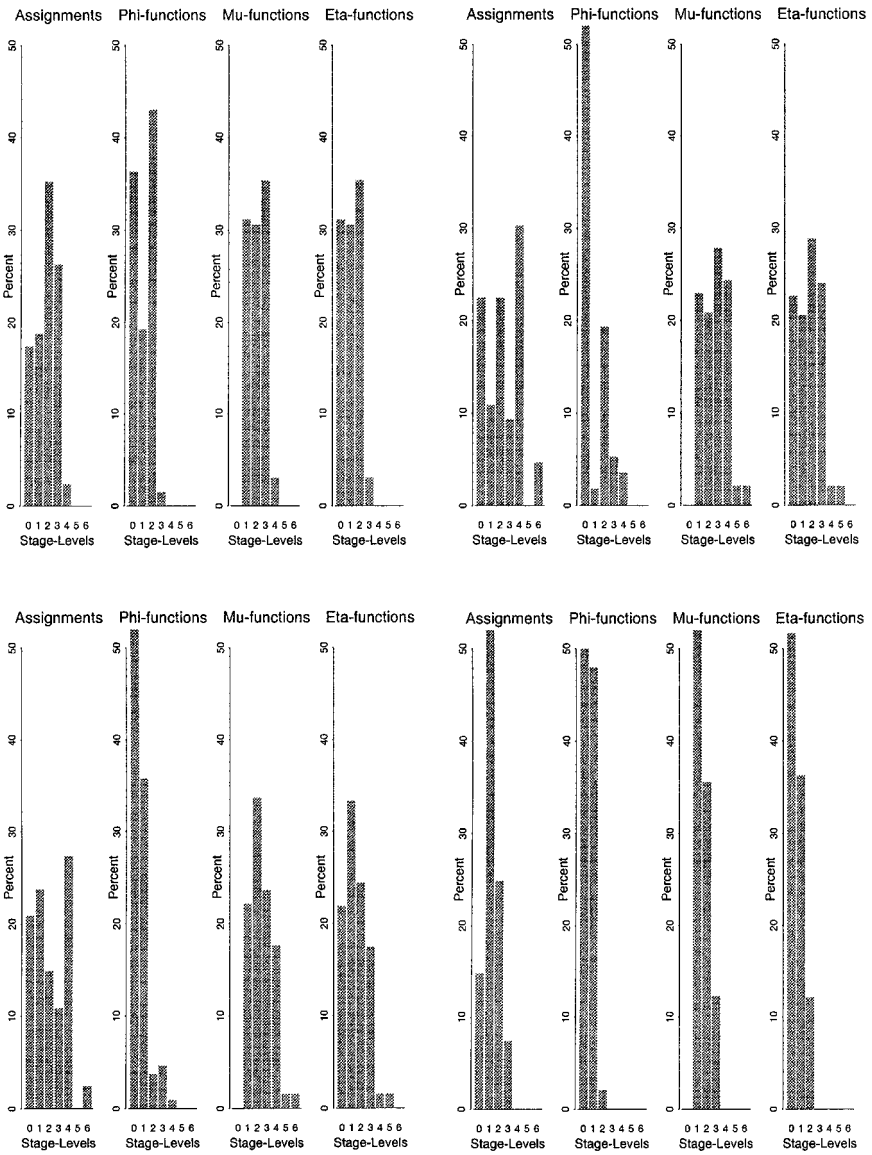


Fig. 5. Histogram of percent of *array* definitions of each type at each level (values >50 % are clipped).

converted to their corresponding count and multiplied by $10^{stage\text{-}level}$ (from the old rule-of-thumb that loops execute ten times).

Various optimizations change the stage-level distributions. Copy propagation replaces variable (pseudo-)definitions with additional uses of the variables of the RHS expression. Common subexpression elimination creates definitions for new temporaries. Dead store elimination removes unused definitions. Loop-invariant code motion pulls expressions up to lower stage-levels.

The VSLs of uses are shown in Section 4.4.

4.3. Validation of GRLA Estimates

The stage-level labels of GRLA are static estimates of dynamic execution frequency. We wish to know how good the estimates are, so we perform the following experiment:

- Measure the dynamic execution frequency of procedures, call sites and loops, and compared the results with the assigned stage-levels.

Figures 6 and 7 show the dynamic execution frequency for procedures, call sites and loops compared to their CSLs. Points are for individual items and the solid (dotted for call sites) line connects the geometric means of the samples for each stage-level. For the loop graphs, several stages have samples which vary by many orders of magnitude. We chose the geometric mean because it captures the mean of order of magnitude. The geometric mean is the nth root of the product of n samples. The dashed line is a reference line showing $(stage\text{-}level, 10^{stage\text{-}level})$.

The graphs show that the CSL is a good predictor both for absolute and relative execution frequency. There are a few anomalies, such as the estimates for Stage-4 loops in BARO and Stage-5 loops in MHD2D. We will look into heuristics to try to compensate for these minor deviances.

A good static predictor for execution frequency is an aid to selecting code to be generated at runtime. It is important to ensure that the code reuse is high enough so that the expense of code generation is a small enough fraction of the runtime that the overall computation time is reduced.

4.4. RTCG Opportunity Characterization

Glacial variable analysis labels variables with useful properties to aid selection of cost-effective value-specific optimization. It estimates the modification frequency of variables and the execution frequency of code.

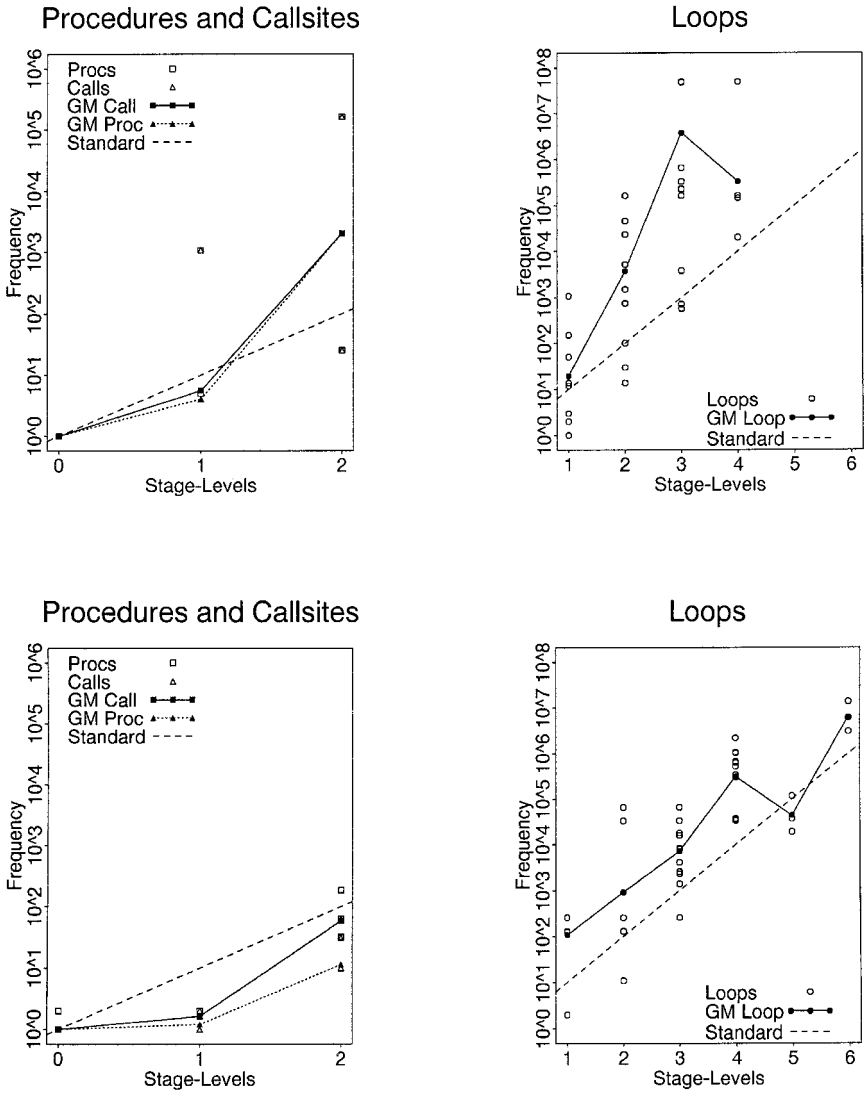


Fig. 6. Dynamic frequency vs. stage-level with geometric mean and $10^{stage\text{-}level}$ reference line.

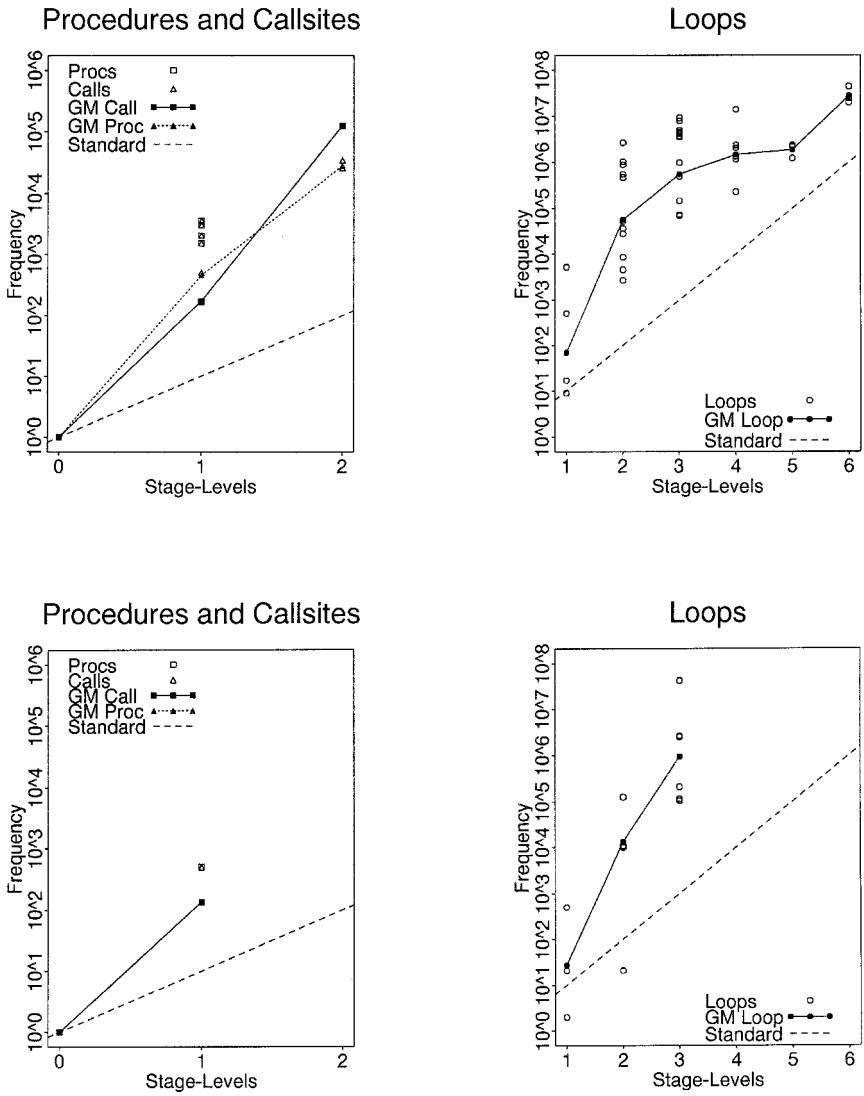


Fig. 7. Dynamic frequency vs. stage-level with geometric mean and $10^{stage\text{-}level}$ reference line.

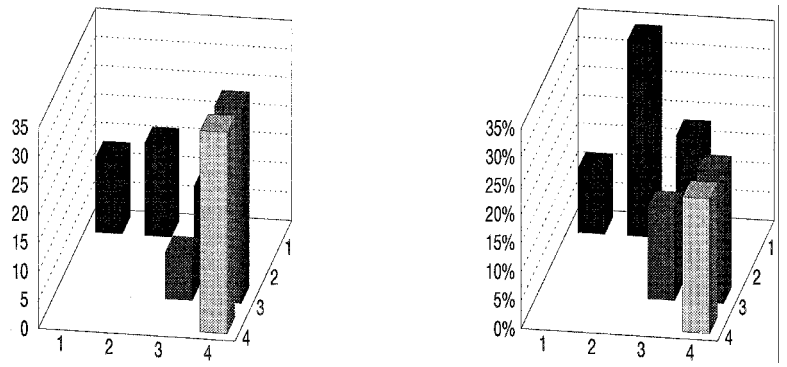


Fig. 8. Degrees of glacialness (*Z*-axis) by stage level (*X*-axis) for SHEAR (scalars).

We wish to know the number of glacial variables that VSO might be able to use. We perform the following experiment:

- Compare the difference between the CSLs and VSLs of scalar expressions. A larger difference indicates more potential invocations to specialized code.
- Do the same for arrays. As noted earlier, scalars are easier to optimize than arrays, but arrays *may* produce bigger payoffs.

This experiment shows the *potential* for VSO based on glacial variables. Figures 8 and 9 show the glacial scalars and arrays, respectively, for SHEAR. The CSL of the use goes from low (left) to high (right) along

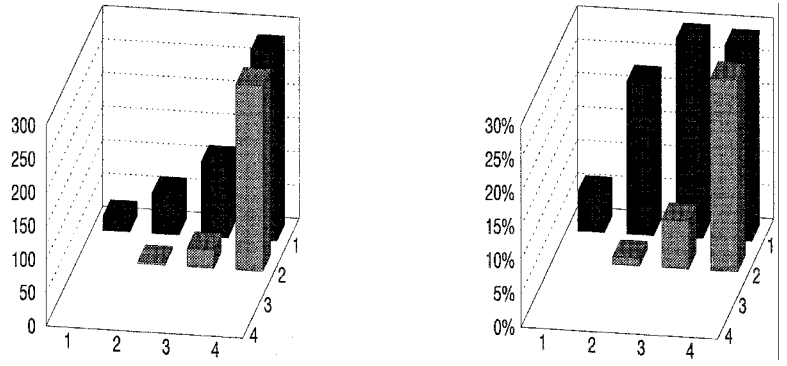


Fig. 9. Degrees of glacialness (*Z*-axis) by stage level (*X*-axis) for SHEAR (arrays).

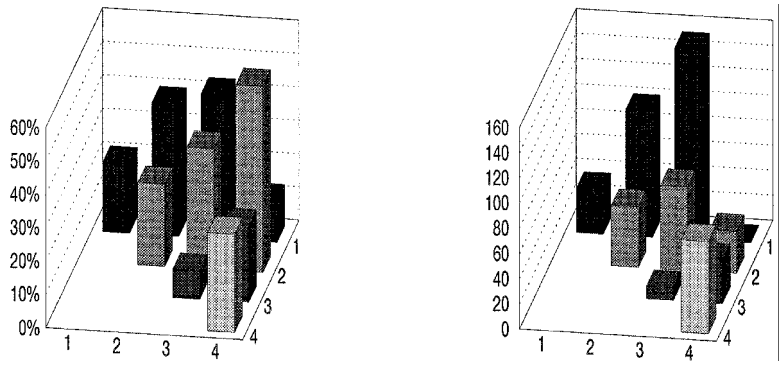


Fig. 10. Degrees of glacialness (*Z*-axis) by stage level (*X*-axis) for benchmark set (scalars).

the *X*-axis. A higher number means the use executes more frequently. The *Z*-axis is the difference between the VSL of the reaching definition and the CSL. The difference goes from low (back) to high (front). A higher number means the definition is more glacial with respect to the use. The row for values with a difference of zero has been left off to make the graph more distinct, but the number of such values is accounted for in the percentages. The left graphs show the absolute count and the right graphs show the percentages of uses with the indicated degree of glacialness. The percentages sum to 100 along the *Z*-axis if uses with zero degrees of glacialness are included. In Fig. 8 notice that about 23% of all scalar uses at CSL-4 have four degrees of glacialness and another 23% have three degrees. This implies that some variable definitions with VSL-0 (1), are reaching the

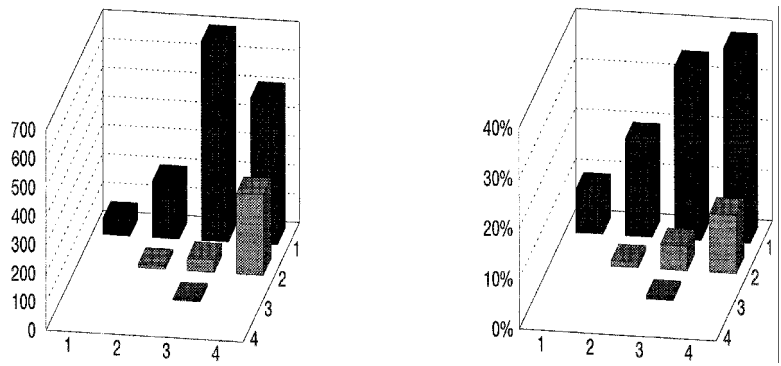


Fig. 11. Degrees of glacialness (*Z*-axis) by stage level (*X*-axis) for benchmark set (scalars).

loops unmodified. And in Fig. 9, notice that about 15% of all array uses at CSL-4 have two degrees of glacialness.

Figure 10 shows the results of the same analysis for scalars averaged over the four programs from the benchmark set. To prevent larger applications from dominating the results, the percentages were normalized for each application and then averaged together. The percentage for variable uses with 2 degrees of glacialness are particularly promising. Figure 11 shows the results for arrays. Figures 4 and 5 can be examined to see the total percentages of all static definitions that occur at each stage-level.

All uses with nonzero degrees of glacialness are glacial variables. Those with two or more degrees of glacialness are likely to be worth using with VSO. More work is needed to analyze how much code and how many values are control-dependent on a given fork in the control-flow graph.

5. CONCLUSIONS

Our glacial variable analysis is the first fully automatic analysis to discover glacial variables for value-specific optimization. The analysis is fully interprocedural and no programmer assistance is required.

We presented the results of our analysis on four complete benchmark programs. The initial results are quite promising: many glacial variables were found, often with several degrees of glacialness. Value-specific optimization is well-understood for integer values and scalars, but not so well for floating-point values and arrays. We are investigating optimizations to make full use of the glacialness of these more difficult types.

One optimization of arrays is to use VSO to generate specialized inspector/executor pairs.[23] In current techniques, the compiler specializes a generic inspector for each loop nest where an index array is used. With GVA, when an index array becomes glacial VSO can be applied to further specialize the inspectors at runt time. When the last index array becomes glacial the runtime code generator can execute the final inspector to generate the executor. The executor is an interpreter of a little program output by the inspector that indicates in what order to perform the loop iterations. The executor, the little program and the loop body can be "compiled," using VSO/RTCG to produce a machine-code version of the loop that honors the data-dependence requirements and is as parallel as possible.

Our future work will include experimenting with modifications to GRLA for procedures called at several levels and for recursive procedures. We will also integrate our glacial variable analysis with a RTCG system to experiment with cost/benefit heuristics, as well as to study opportunities for new optimizations based on VSO and data specialization.[24]

ACKNOWLEDGMENTS

We would like to acknowledge Robert Prouty and Sally McKee for their thoughtful reviews that greatly aided the comprehension of the paper and the anonymous reviewers for their suggested improvements.

REFERENCES

1. C. Consel and O. Danvy, Tutorial notes on partial evaluation, *Conf. Record of the 20th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Progr. Lang.*, Charleston, South Carolina, pp. 493–501 (January 1993).
2. M. W. Hall, Managing Interprocedural Optimization, Ph.D.Thesis, Department of Computer Science, Rice University (1991).
3. D. Keppel, S. J. Eggers, and R. R. Henry, Evaluating runtime-compiled value-specific optimizations, Technical Report UWCSE 93-11-02, Department of Computer Science and Engineering, University of Washington (November 1993).
4. D. Keppel, Runtime Code Generation, Ph.D. Thesis, Department of Computer Science and Engineering, University of Washington (1996).
5. J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad, Fast, effective dynamic compilation, *Proc. ACM SIGPLAN '96 Conf. Progr. Lang. Design and Implementation*, pp. 149–159 (see Ref. 25.)
6. C. Consel and F. Noël, A general approach to runtime specialization and its application to C, *Conf. Record of POPL '96: 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Progr. Lang.*, St. Petersburg, Florida, pp. 145–156 (January 1996).
7. D. R. Engler, Vcode: A retargetable, extensible, very fast dynamic code generation system, *Proc. ACM SIGPLAN '96 Conf. on Progr. Lang. Design and Implementation*, pp. 160–170 (see Ref. 25).
8. P. Lee and M. Leone, Optimizing ML with runtime code generation, *Proc. ACM SIGPLAN '96 Conf. Progr. Lang. Design and Implementation*, pp. 137–148 (see Ref. 25).
9. N. D. Jones, C. K. Comard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Englewood Cliffs, New Jersey, Prentice-Hall (1993).
10. R. Glück and J. Jørgensen, Efficcient multi-level generating extensions for program specialization, Technical Report D-229, DIKU, Department of Computer Science, University of Copenhagen (1995).
11. A. P. Ershov, Mixed computation: The potential applications and problems for study, *Theoret. Comput. Sci.* **18**:41–67 (1982).
12. U. Jørring and W. L. Scherlis, Compilers and staging transformations, *Conf. Record of the Thirteenth Ann. ACM Symp. on Principles of Progr. Lang.*, St. Petersburg Beach, Florida, pp. 86–96 (January 1986).
13. U. Hölzle and D. Ungar, Optimizing dynamically-dispatched calls with runtime type feedback, *Proc. ACM SIGPLAN '94 Conf. Progr. Lang. Design and Implementation*, pp. 326–336 (June 1994).
14. C. Chambers, The design and implementation of the SELF compiler, an optimizing compiler for object-oriented programming languages, Ph. D. Thesis, Department of Computer Science, Stanford University, 1992.
15. M. W. Hall and K. Kennedy, Efficient call graph analysis, *ACM Letters on Progr. Lang. Syst.* **1**(3):227–242 (September 1992).
16. P. Havlak, Nesting of reducible and irreducible loops, *ACM Trans. Progr. Lang. Syst.* **19**(4):557–567 (July 1997).

17. M. Wolfe, *High-Performance Compilers for Parallel Computing*, Reading, Massachusetts: Addison-Wesley (1996).
18. R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein, The program dependence web: A representation supporting control, data, and demand-driven interpretation of imperative languages, *Proc. ACM SIGPLAN '90 Conf. Progr. Lang. Design and Implementation*, pp. 257–271 (June 1990).
19. P. Havlak, Construction of thinned gated single-assignment form. In U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua, (eds.), *Languages and Compilers for Parallel Computing Proc. sixth Int'l. Workshop*, Berlin, Germany: Springer-Verlag, Portmand, Oregon, pp. 477–499 (August 1993).
20. M. N. Wegman and F. Z. Zadeck, Constant propagation with conditional branches, *ACM Trans. Progr. Lang. Syst.* **13**(2):181–210 (April 1991).
21. T. Autrey, Demand-driven interprocedural constant propagation: Implementation and evaluation. Technical Report OGI-CSE-96-008, Department of Computer Science and Engineering, Oregon Graduate Institute (1996).
22. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. F. Zadeck, Efficiently computing static single assignment form and the control dependence graph, *ACM Trans. Progr. Lang. Syst.* **13**(4):451–490 (October 1991).
23. L. Rauchweger, N. M. Amato, and D. A. Padua, Runtime methods for parallelizing partially parallel loops, *Proc. ACM Conf. Supercomputing*, pp. 137–146, Barcelona, Spain (July 1995). ACM SIGARCH.
24. T. B. Knoblock and E. Ruf, Data specialization, *Proc. ACM SIGPLAN '96 Conf. Progr. Lang. Design and Implementation*, pp. 215–225.
25. ACM Sigplan, *Proc. ACM SIGPLAN '96 Conf. Progr. Lang. Design and Implementation*, Philadelphia, Pennsylvania (May 1996).