Extended SSA with Factored Use-Def Chains to Support Optimization and Parallelism

Eric Stoltz
Michael P. Gerlek
Michael Wolfe

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 93-013

June 1993

Extended SSA with Factored Use-Def Chains to Support Optimization and Parallelism

Eric Stoltz*, Michael P. Gerlek*, Michael Wolfe*

Oregon Graduate Institute of Science and Technology

Department of Computer Science and Engineering

PO Box 91000
Portland, Oregon 97291

(503) 690-1151

FAX: (503) 690-1553

{stoltz,gerlek,mwolfe}@cse.ogi.edu

* supported in part by a grant from Intel
Supercomputer Systems Division and the
Oregon Advanced Computing Institute

Extended SSA with Factored Use-Def Chains to Support Optimization and Parallelism

Abstract

This paper describes our implementation of the Static Single Assignment (SSA) form of intermediate program representation in our prototype parallelizing Fortran 90 compiler, Nascent.

Although the traditional Static Single Assignment (SSA) form algorithm renames variables uniquely at every definition point, it is not practical to add new names to the symbol table at all assignments. Thus, most implementations actually provide def-use chains for each definition. Each use is the head of exactly one link, thus maintaining the semantics of SSA. In contrast, we provide use-def chains, so that in the intermediate representation the link at each use points to its unique reaching definition. We discuss how our approach improves the implementation and efficiency of optimization and analysis techniques such as constant propagation, induction variable detection, and scalar dependence identification.

Nascent accepts explicit parallel section constructs consistent with the Parallel Computing Forum (PCF) standard and will conform to the High Performance Fortran (HPF) standard. We have developed analysis and optimization routines to support these parallel languages which take advantage of our extended SSA form.

1 Introduction

The Static Single Assignment (SSA) form for intermediate program flow representation has become a popular and powerful framework with which to analyze and optimize code [BMO90, CG93, JP93, WZ91, RWZ88b]. SSA form has been used to simplify optimization techniques and data-flow problems including constant propagation [WZ91], global value numbering [RWZ88b], and induction variable detection [Wol92], among others.

```
x = 0                    x0 = 0
y = 0                    y0 = 0
z = 0                    z0 = 0
if ( P ) then            if ( P ) then
  y = y + 1                y1 = y0 + 1
endif                    endif
                         y2 = φ ( y0, y1 )
x = y                    x1 = y2
z = 2 * y - 1            z1 = 2 * y2 - 1

(a)                      (b)
```

Figure 1: program fragment in (a) normal form and (b) SSA form

Figure 1 shows a program fragment translated into SSA form. The distinguishing feature of this form is that each variable use has exactly one reaching definition. Additional definitions are added at each control flow confluence point for each variable defined along at least one of the incoming

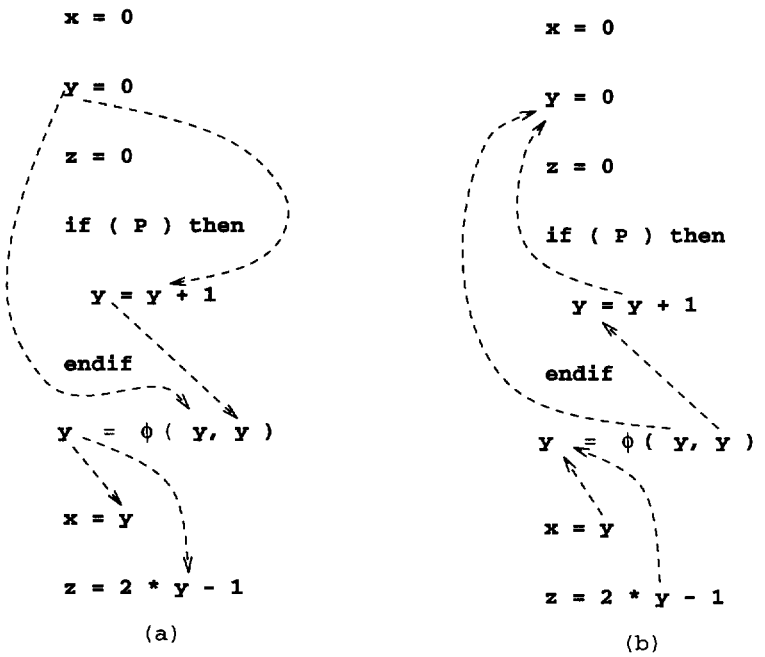

Figure 2: comparison of standard SSA implementation employing (a) def-use links and (b) use-def links

paths by introducing a ϕ-function. Details are provided in [CFRW91].

Although this traditional SSA form renames variables uniquely at every definition point, it is not really practical (and certainly not desirable) to add new names to the symbol table for all assignments. Thus, the common implementation [WZ91, JP93] actually provides def-use links [ASU86] for each new definition (see Figure 2). Since each use is the head of exactly one link, the semantics of SSA are preserved.

The def-use chain style of SSA implementation lends itself well to for-

ward data-flow problems [WZ91, RWZ88a] due to consistency of direction between program flow and def-use links. However, a demand-driven data-flow problem will typically request information at a program point from its data-flow predecessors. As we shall see, use-def chains admirably match the demand-driven style of data-flow analysis.

In this paper we describe our implementation of SSA with use-def chains and discuss how this approach facilitates optimization and analysis techniques such as constant propagation, induction variable recognition, and scalar dependence detection. We also describe an extension to SSA that allows correct analysis of explicit parallel sections. We have implemented SSA form in Nascent, our prototype Fortran 90 compiler. Nascent accepts explicit parallel section constructs consistent with the Parallel Computing Forum extensions and will also conform to the High Performance Fortran (HPF) standard.

The rest of the paper is organized as follows. Section 2 provides the details of our use-def SSA implementation, Section 3 presents the advantages of this approach, while Section 4 will describe its applications to support parallel languages. We will cover related work in Section 5 and conclude with Section 6.

2 Constructing SSA with Factored Use-Def Chains

This section details the methods and techniques employed to transform an intermediate representation into SSA form utilizing factored use-def chains (FUD chains).

We begin our analysis of programs within a framework consisting of a control-flow graph and an SSA data-flow graph. A control flow graph (CFG) is a graph G = <*V,E,Entry,Exit*>, where V is a set of nodes representing basic blocks in the program, E is a set of edges representing sequential control flow in the program, and *Entry* and *Exit* are nodes representing the unique entry point into the program and the unique exit point from the program. For technical reasons, a flow graph edge connects *Entry* to *Exit* [CFRW91]. Each basic block contains a list of intermediate code tuples, described next. We restrict our attention to reducible flow graphs [ASU86].

The data-flow graph consists of tuples of the form <*op,left,right,ssalink*>, as described in [Wol92], where *op* is the operation code and *left* and *right* are the two operands (both are not always required, e.g. a unary minus). The *ssalink* is used for fetches, including arguments of ϕ-functions, as well as indexed stores (which are not discussed further in this paper). The *ssalink*, if applicable, represents the one reaching definition for the variable in question

at that point in the program. The *left*, *right*, and *ssalink* fields are pointers: they are all essentially use-def links. Thus, to perform an operation associated with any tuple, a request (or *demand*) is made for the information at the end of these links.

In converting intermediate code into SSA form, we generally follow the algorithm given in [CFRW91], which relies extensively upon the concepts of *dominators* and *dominance frontiers* for nodes in the CFG. Briefly, node X *dominates* node Y if all paths from *Entry* to Y must pass through X. X *strictly dominates* Y if X dominates Y and $X \neq Y$. X is the *immediate dominator* of Y if X strictly dominates Y and all other nodes which strictly dominate Y also dominate X. Z is in the *dominance frontier* of Y if Y does not strictly dominate Z, but Y does dominate some predecessor of Z.

When performing the transformation into SSA form, we follow four main steps:

1. **Variable Modification List**

   We make one pass through the CFG to create a linked list of variables modified and a list of modification sites for each variable. This is accomplished by simply examining all data-flow tuples which belong to each node of the CFG using a depth-first search, although any

pass which visits all nodes will suffice. We note that all variables are assumed to be initialized (hence defined) at *Entry*.

2. Dominator Computation

We compute the immediate dominator tree of the CFG using the algorithm by Lengauer and Tarjan [LT79]. The dominance frontier of all nodes is then quickly assembled using a compact technique described by [CFS90], which is just as fast as previous techniques and an improvement in terms of space. In addition, it is much simpler to code; our original 150 lines of code for this phase was reduced to just 10!

3. ϕ Placement

The placement of ϕ-functions is done exactly according to the algorithm in [CFRW91]. For each variable, we place a ϕ-function at the *iterated* dominance frontier of nodes in the CFG associated with all modification sites for that variable. The iterated dominance frontier of a node consists of the original dominance frontier set of the node plus the dominance frontier of each node in the set, augmented by nodes in which ϕ-functions are placed. The number of arguments of each ϕ-function is equal to the number of predecessors of the CFG node in which it resides.

4. **Renaming**

A depth-first pass is made of the dominator tree, pushing definition sites onto a stack when encountered. All variable uses have their *ssalink* field filled in with a pointer to the current definition at the top of the stack. Flow graph successors are then checked for ϕ-functions, filling in the corresponding ϕ-argument with a pointer to the reaching definition at the top of the stack. This routine is called recursively on the dominator tree children, popping definitions off the stack when returning.

A traditional use-def chain would list all definitions of a variable which reach the current use. The result of the preceding procedure is the *factored* form – each use has exactly one reaching definition (see Figure 2b), thus preserving SSA semantics.

3 Advantages of FUD Chains

3.1 Space Requirements

Although relatively minor, the first advantage of our approach is that it requires only constant space per node to implement. Since each variable use has exactly one reaching definition, it has only one *ssalink* field. A

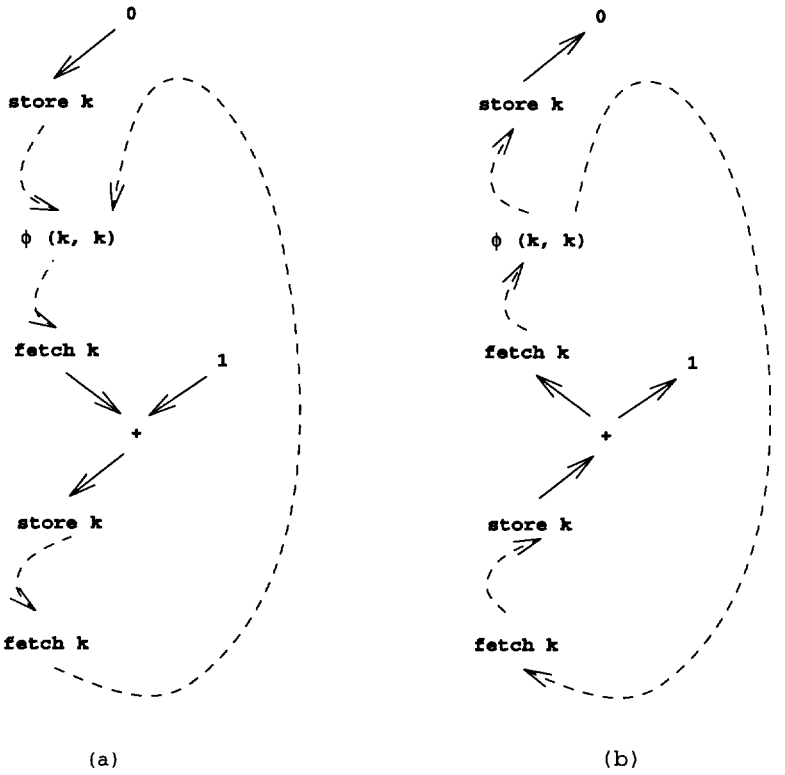

Figure 3: induction variable detection with (a) def-use links and (b) use-def links

traditional def-use implementation, however, must have the capability to dynamically expand its use list at each definition site since an unbounded number of uses can be dependent upon that site. Contrast Figure 2(a) with 2(b): since the pointers in (b) are stored with the node at the tail of each link, at most one link need be stored with each tuple.

3.2 Demand-Driven Analysis

FUD chains readily lend themselves to demand-driven analysis. Traditional iterative solvers for optimizations such as constant propagation require many passes, recomputing information until a fixed point is reached [FL91, ASU86, Ken81]. In our approach we classify each node in the SSA data-flow graph in terms of its predecessors using a depth-first search.

Constant propagation is performed in Nascent using a standard constant propagation lattice, with each node being assigned a lattice value as a function of the lattice value of its data-flow predecessors. If the predecessors have not yet been evaluated, the algorithm is recursively called on those tuples corresponding to the targets of the links. This approach requires only one pass.

We apply techniques outlined in [BMO90] in order to correctly propagate values down conditional branches. This entails associating a *predicate* with each ϕ-function so that, if possible, we can determine which branch will be taken to reach the current confluence point. When dealing with loops, our demand-driven approach may find a cycle in the data-flow graph. We then use a standard iterative method, but only for those operations in the cycle, resulting in less recalculation and faster convergence. These ideas will be

expanded in a forthcoming paper.

Induction variable detection and classification is an important first step in many program transformations, particularly loop parallelization. We can generalize from the detection of constant values to the detection of many forms of recurrence relations, including linear and nonlinear induction variables, as well as variables that describe monotonic or periodic sequences [Wol92]. Based on the observation that statements describing induction variables correspond to strongly connected regions (SCRs) in the data-flow graph, we use Tarjan's depth-first algorithm [Tar72] to detect these regions.

Nodes in trivial SCRs have values determined as functions of their data-flow predecessors, as in constant propagation; however for induction variable classification we assign not just constant values but symbolic expressions. Nodes contained within nontrivial SCRs are assigned expressions corresponding to the type of recurrence relation or other sequence described by the operations in the region. The same SCRs would be discovered in the traditional def-use SSA graph as in our use-def SSA graph if a data-driven approach were used (i.e. operator to operand links), but clearly the ability to define a node in terms of its dataflow predecessors in one pass would be lost. As shown in Figure 3 (a) and (b), there is a cycle containing `k = k + 1` in both SSA graphs, but in the def-use graph the initial value

of the induction variable cannot be determined from within the context of the cycle unless all the predecessors of the cycle have first been visited. In Figure 3(a) both the *store* `k = 0` and the `+ 1` nodes must have been classified prior to classifying the cycle. However, this implies that classification of the cycle, although initially discovered as a descendant of one of these two nodes, must be delayed until *all* predecessors have been visited. In 3(b), employing use-def links, we recursively visit all predecessors using Tarjan's algorithm, thus insuring that the cycle can be correctly classified when first detected.

4 Support for Parallelism

In this section we describe how our implementation of SSA can be used to detect and support parallelism.

4.1 Data Dependence

In order to execute code in parallel, most notably loops, dependences must be preserved [WB87]. Induction variable detection, as explained in section 3, allows analysis of subscripts for array references. Given two statements S_i and S_j, if a dependence exists between these two statements we denote

that relationship: $S_i \; \delta \; S_j$. Specific dependences annotate this notation.

In general, one must preserve the following data dependences [PW86]:

- Flow-dependence: read after write, $S_i \; \delta \; S_j$
- Anti-dependence: write after read, $S_i \; \overline{\delta} \; S_j$
- Output-dependence: write after write, $S_i \; \delta^o \; S_j$

When loops are considered, we are often interested in distance and/or direction of the dependence. Dependence distance reflects the number of iterations around the loop that the dependence is *carried*, while dependence direction just identifies whether the direction is positive, zero, or negative A dependence within the same iteration is considered *loop independent.* The distance or direction annotates the dependence relation by using subscripts. For example, if a flow dependence exists between S_i and S_j with a distance of d, it is denoted $S_i \; \delta_{(d)} \; S_j$, while if its direction is $(<)$, it is denoted $S_i \; \delta_{(<)} \; S_j$.

To understand how variable values are carried around loops and merged at confluence points, we note that ϕ-functions come in two "flavors": merge nodes as the result of conditional branches, and those at the top or *header* of a loop. The loop-header ϕ-function merges the definition coming into the loop with the last definition within the loop body.

We have developed a method of detecting flow and output dependence for scalars within a loop based upon FUD chains in SSA. We shall describe the method in detail for identifying flow dependence within a single loop. The following algorithm takes as input a loop within our SSA framework and outputs scalar flow-dependences for variable uses within the loop body.

Detecting Scalar Flow Dependence Within a Loop

def_list ⇒ list of definitions reached by a loop-header ϕ-function
self ⇒ flag to indicate whether ϕ-function transitively reaches itself
variable_of ⇒ returns the variable associated with a ϕ-function
ssalink ⇒ returns the reaching definition corresponding to a use

```
find_scalar_dependences( loop L )
  for each loop-header φ-function f in L do

    def_list(f) = ∅
    self(f) = false
    v = variable_of(f)

    for each argument arg of f do
      if arg points within loop body of L then
        pre_process( ssalink(arg),f )
      endif
    endfor

    for all uses u of v within L do
      follow_use( ssalink(u),f,u )
    endfor

  endfor
end find_scalar_dependences
```

```
pre_process( def arg, φ-function f )
  if arg == scalar definition then
    def_list(f) = def_list(f) ⋃ {arg}
  else if arg == f /* self-reference */
    self(f) = true
  else  /* non-loop-header φ-function */
    for each φ-argument param of arg do
      pre_process( ssalink(param),f )
    endfor
  endif
end pre_process

follow_use( def d, φ-function f, use u )
  if d ==  scalar definition then
    set flow-dependence:  d δ(0) u
  else if d == f /* loop-header φ-function */
    for each i on def_list of d do
      if self(d) then
        set flow-dependence:  i  δ(<) u
      else
        set flow-dependence:  i  δ(1) u
      endif
    endfor
  else  /* non-loop-header φ-function */
    for each argument a of d do
      follow_use( ssalink(a),f,u )
    endfor
  endif
end follow_use
```

Let us follow this procedure through an example. In Figure 4 we first pre-process the loop-header ϕ-function at S_1 for T. The first argument, T_0, comes from outside the loop, so is considered no further. The second argument, T_3, points to its reaching definition, which is a ϕ-function in S_6. Thus we follow

its arguments, $\mathtt{T}_2$ and $\mathtt{T}_1$. $\mathtt{T}_2$ is a scalar killing definition, so it is added to the *def_list* for the loop-header ϕ-function. $\mathtt{T}_1$ is a self-reference to the loop-header ϕ-function, so the *self* flag is set; at this point the pre-processing phase concludes.

Next we look for any uses of T. The first use we find, lexically, is a use of T in S_4. Its *ssalink* points to the scalar definition of T in S_3, thus discovering the flow dependence, $S_3\ \delta_{(0)}\ S_4$, which is loop-independent. The next use of T is in S_7. Its *ssalink* points to the ϕ-function in S_6. Following the first argument takes us to the scalar definition of T in S_3, which means we have a loop-independent flow dependence: $S_3\ \delta_{(0)}\ S_7$. The second argument from S_6 takes us to the the loop-header ϕ-function at S_1. Thus, a flow dependence exists from each element in the *def_list* to the original use in S_7. In this case, there is only one item in the *def_list*, $\mathtt{T}_2$, so we have established the flow dependence: $S_3\ \delta_{(<)}\ S_7$, with a $(<)$ dependence direction since the *self* flag is set. A $(<)$ direction lets us know the direction of the dependence, but says that we can not calculate the exact distance, in this case due to the uncertainty of which iteration will execute the `if` condition at S_2.

Intuitively, this procedure works since a loop independent flow dependence will always be discovered by following use-def links in the current loop body, while loop-carried dependences must always flow through the

```
S0:  loop:
S1:   T1 = φ ( T0, T3 )
S2:   if ( R ) then
S3:     T2 = ...
S4:     ... = T2
S5:   endif
S6:   T3 = φ ( T2, T1 )
S7:   ... = T3
S8:  endloop
```

Figure 4: sample loop to detect scalar dependences

loop-header ϕ-function. If the *self* flag is set, a non-loop-header ϕ-function must have been encountered during the pre-processing phase; this indicates a conditional branch which may or may not be taken during any particular iteration. Thus, while the dependence is loop-carried and its direction is $(<)$, we do not know its distance. If the *self* flag is not set, then the flow dependence must be to the subsequent iteration, hence its distance is 1.

The purpose of the algorithm is to identify scalar flow dependences within a given loop, providing a distance or direction around loop iterations. Dependences which originate from outside the loop are not recognized here, such as the flow dependence when $\mathtt{T}_0$ was defined outside the loop to the use of T in S_7. We also do not consider here the detection of scalar dependences which cross loop boundaries. Nested scalar dependence can be analyzed us-

ing recursive routines by providing trip count information concerning each loop. We are currently developing methods to extend analysis into these areas.

Arrays can also be analyzed with this approach, but would be treated as a scalar without regard to any subscripts. A similar algorithm to the one presented above is used to identify output dependence. Anti-dependence, on the other hand, is a more complex task and the subject of future work.

4.2 Explicit Parallel Sections

We have designed Nascent to accept explicit parallel section constructs consistent with the Parallel Computing Forum extensions to Fortran [PCF91]. An example of parallel section constructs, similar to ideas advocated by Hansen [Han73] and Allen et al. [ABC+88], is given in Figure 5. We needed to design our intermediate representation to translate these programs into SSA form, preserving the unique use-def link when applicable, but dealing with multiple (perhaps anomalous) updates among parallel sections. We solved this problem by adding a new operator to the SSA representation, the ψ-function, to merge multiple parallel updates. This extension of SSA form is necessary since with explicit parallel sections a parallel merge is distinguishable from a sequential merge. Parallel section *coend* nodes are

```
S1:   k = 0
S2:   if ( Q ) then
S3:     k = 3
S4:   else
S5:     k = k + 2
S6:   endif
```

(a)

```
S1:   k = j = 0
S2:   Parallel Sections
S3:   Section A
S4:      k = 3
S5:   Section B
S6:      k = k + 2
S7:   Section C
S8:      j = j - 1
S9:   End Parallel Sections
```

(b)

Figure 5: contrasting sequential and parallel control flow

true merges of multiple control flow paths from explicit parallel section, *all* of which must execute, while a sequential merge is actually a choice or selection of the possible incoming edges.

Figure 5 demonstrates this distinction between sequential and parallel confluence points on two small fragments of code. In (a) only one branch will, in fact, reach S_6. However, in (b) all branches may execute in parallel, thus it is possible for *more than one* definition of a variable (**k**, in this case) to actually reach S_9.

As with ϕ-functions, arguments to ψ-functions are filled in with a use-def *ssalink*. However, as we see in Figure 6 (Figure 5(b) translated into SSA form using ψ-functions), there is not necessarily a one-to-one correspondence

```
k0 = j0 = 0
Parallel Sections
Section A
  k1 = 3
Section B
  k2 = k0 + 2
Section C
  j1 = j0 − 1
End Parallel Sections
k3 = ψ ( k1, k2 )
```

Figure 6: previous parallel program translated into SSA form with ψ-functions

between predecessors of the merge and the number of arguments of the ψ-function, as there was with ϕ-functions. An additional complication arises due to the possible nested nature of `Parallel Section` statements. Thus, the algorithm to place ψ-functions, due to its parallel semantics, is somewhat different than that used for ϕ-functions. When performing the renaming phase, however, each definition point is treated equally and pushed onto the same stack, be it a scalar killing definition, array assignment, ϕ- or ψ-function.

Once ψ-functions are placed, explicit parallel code is smoothly translated into SSA form. ψ-functions, just like ϕ-functions, generate new variable definitions, and their arguments are considered uses of that variable. We gain a consistent framework with which to sensibly reason about such parallel

programs. More detail on this project can be found in a preliminary paper [SHW93].

5 Related Work

Related work in this area also addresses program representation with which to perform intermediate optimization and analysis. The work by Cytron et al. [CFRW91] established the foundation for an intermediate form which significantly reduces the complexity of def-use links. Their method describes efficient techniques to compute the location of, and minimally construct, ϕ-function augmentation to data-flow graphs.

The Program Dependence Web (PDW) [BMO90] attempts to provide a framework for integrating control dependence as well as SSA within a consistent model. The SSA form is extended to include *gating* functions which provide predicate information for non-loop-header ϕ-functions as well as exit values for variables within loops. The PDW is considered an augmented Program Dependence Graph, a graph which captures both control and data-dependence information.

Another method employed to generalize SSA is the Dependence Flow Graph [JP93] which combines control information with def-use SSA chains

by defining *control regions* which consist of single-entry, single-exit areas of the control flow graph. The aim of this work is to allow both forward and backward data-flow analysis, as well as to provide a representation that can be transformed into SSA form or used to construct control dependence graphs.

6 Conclusion

In this paper we have described how the intermediate SSA form is actually implemented in our Fortran compiler, Nascent, utilizing factored use-def chains as an alternative to the more popular def-use chains. This approach has profound implications on the general nature and style of data-flow analysis. FUD chains readily accommodate demand-driven data-flow analysis, as opposed to using more traditional iterative solvers.

We have also detailed extensions to SSA to accommodate work with parallel languages. We facilitate data dependence analysis with new techniques for detecting both linear and advanced induction variables, and have outlined a new method for identifying scalar flow dependence within loops. We have also provided a framework with which to reason more intelligently about explicit parallel sections.

[Han73] Per Brinch Hansen. *Operating System Principles.* Prentice-Hall, Englewood Cliffs, NJ, 1973.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1993.

[Ken81] Ken Kennedy. Global data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Practice*, page 18. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, July 1979.

[PCF91] Parallel Computing Forum. PCF Parallel Fortran Extensions. *Fortran Forum*, 10(3), September 1991. (special issue).

[PW86] David A. Padua and Michael Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, December 1986.

[RWZ88a] Barry K. Rosen, Mark N. Wegman, and Kenneth F. Zadeck. Detecting equality of variables in programs. In *Fifteenth Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, January 1988.

[RWZ88b] Barry K. Rosen, Mark N. Wegman, and Kenneth F. Zadeck. Global value numbers and redundant computation. In *Fifteenth Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, January 1988.

[SHW93] Harini Srinivasan, James Hook, and Michael Wolfe. Static single assignment for explicitly parallel programs. In *Twentieth Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, January 1993.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, June 1972.

[WB87] Michael Wolfe and Utpal Banerjee. Data dependence and its applications to parallel processing. *International Journal of Parallel Programming*, April 1987.

[Wol92] Michael Wolfe. Beyond induction variables. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[WZ91] Mark N. Wegman and Kenneth F. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, July 1991.