Analyzing Programs with Explicit Parallelism

Harini Srinivasan and 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 91-022

July, 1991

To appear in the
Proceedings of the Fourth Workshop on
Languages and Compilers for Parallel Processing
Santa Clara, California, August 7-9, 1991

Analyzing Programs with Explicit Parallelism

Harini Srinivasan Michael Wolfe

July 29, 1991

Abstract

When analyzing programs with parallel imperative constructs (*e.g.*, cobegin/coend), standard computer intermediate representations (Control Flow Graphs) are inadequate. This paper discusses semantics for parallel constructs, and introduces new intermediate forms, called the *Parallel Control Flow Graph* and the *Parallel Precedence Graph*. These data structures have certain advantages for compiler analysis and optimization. As an example of the advantages, the analysis requirements of converting an explicitly parallel program into Static Single Assignment form are given. To do this, the dominance relation and dominance frontiers for explicitly parallel programs must be defined.

1 Introduction

Given the failure of automatic parallelizing compilers, many users want to explore writing explicitly parallel programs. Some language and compiler researchers believe that explicit parallelism should be avoided, and functional or applicative implicitly parallel languages should be used. Nonetheless, a significant user community desires and demands language constructs for expressing explicit parallelism in programs. The Parallel Computing Forum was formed to generate portable syntax and semantics for parallel extensions to Fortran-77 [15]; this consortium of industry and academic parties has now spawned an ANSI standards committee to complete the project. In order to deliver the very best performance, compilers will soon be required to perform aggressive optimization in the presence of explicit parallelism.

The standard intermediate form for compilers is the Control Flow Graph, or CFG. To help with compiler optimizations, other information is generally collected about the program; this information is sometimes represented explicitly in an auxiliary data structure, or replaces the CFG as the primary data structure, *e.g.*, the *data dependence graph*, *program*

dependence graph, *program dependence web*, *dependence flow graph*, and so on. Many optimizations have been designed around the Static Single Assignment (SSA) form of the program [20, 4, 18, 9]. This paper focuses on how to convert an explicitly parallel program to SSA form.

The standard algorithms for converting a program to SSA form use the information in the CFG; focusing on the `Parallel Sections` construct of PCF Fortran, we show here that adding parallelism to a CFG is non-trivial. Instead we propose a new model for control flow in parallel programs, imaginatively named the Parallel Control Flow Graph (PCFG). Explicit ordering between different sections is represented by a Parallel Precedence Graph (PPG).

Two important concepts used in deriving the SSA form of a program are the dominance relation defined between nodes in a Control Flow Graph, and the dominance frontiers of nodes in a CFG. Extending these definitions to nodes in a PCFG is not straightforward; a PCFG has two types of nodes, those representing parallel constructs and those representing basic blocks. This paper defines the dominance relation and dominance frontiers for PCFGs; efficient algorithms to compute these are described in the references [19, 22, 21, 23].

Of critical importance for creating the SSA form of a parallel program is the definition of what are the reaching definitions for a variable in an explicitly parallel program. The concept of reaching definitions comes from the semantics of the language; we explore *copy-in/copy-out* semantics for parallel constructs, both for clarity in writing parallel programs, and to simplify and improve compiler analysis.

2 Parallel Section Semantics

The `Parallel Sections` construct [15] is similar to a `cobegin/coend` [7] or the `Parallel Cases` statement introduced by Allen et al [2]. It is a block structured construct used to specify parallel execution of identified sections of code. The parallel sections may also be nested. The sections of code must be data independent, except where an appropriate synchronization mechanism is used. Here we consider only structured synchronization expressed as `Wait` clauses, *i.e.*, DAG parallelism [10]. Transfer of control into or out of a parallel section is not supported.

Some definition must be made when two sections of code that can execute in parallel both modify the same variable, or when one section modifies a variable that is used by the other. Consider the program in Figure 1(a) as an example. What values should be printed for `t` and `u`? What values for `w` can reach statement 6? What assignments to `v` can

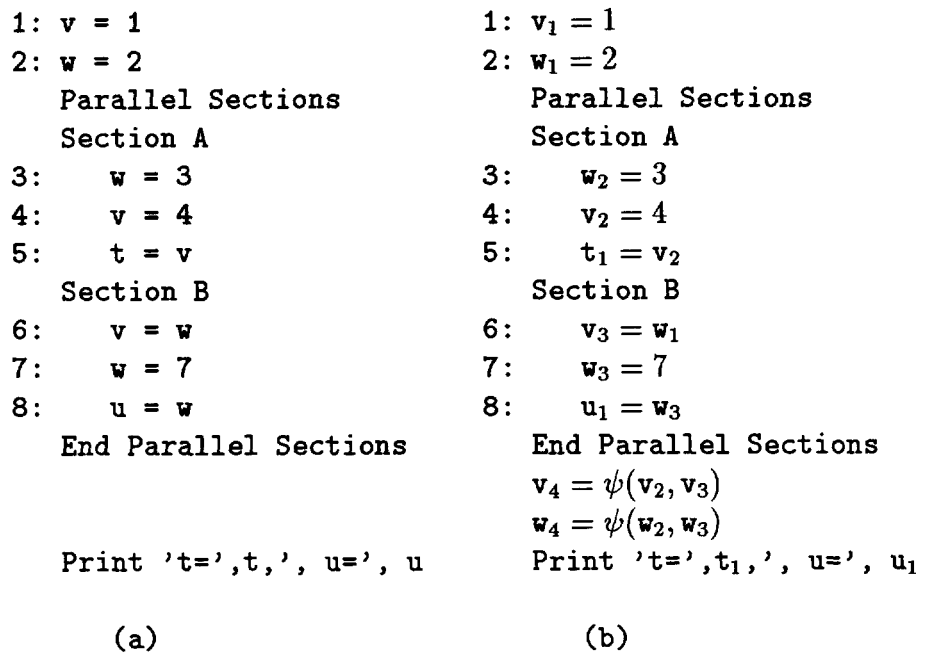

Figure 1: Parallel Sections Construct and its SSA Form

reach statement 5? Can statements 3 and 4 be interchanged? Can the compiler forward substitute statement 4 into statement 5? These are all questions that the compiler should be able to answer via analysis of the program.

Under some models of parallelism, such as a model allowing any *sequentially consistent* execution [13], there is more than one legal output for this program; we might consider the following possibilities (to save space, only a few orderings are shown):

statement ordering	output
3,4,5,6,7,8	t=4, u=7
3,4,6,7,5,8	t=3, u=7
3,6,4,5,7,8	t=4, u=7
3,6,4,7,5,8	t=4, u=7
6,3,4,5,7,8	t=4, u=7
6,7,3,4,5,8	t=4, u=3

Note that optimization within a parallel section is restricted; even though statements 3 and 4 are completely data independent, interchanging these two statement would allow the statement order: 4,6,7,3,5,8, which would give the unexpected output t=2,u=3. The legal

outputs for this program depend on how the anomalous parallel updates to the variables are resolved, and this depends on the rules for the language. Several different rules can be (and have been) proposed for such syntax:

Error: The language might define anomalous updates to variables as a programmer error. In this case no output is a legal output, since this is an illegal program. With the Ada view of a language, such errors should be detected and reported, either at compile time (if possible) or at run time. While detection of potential anomalous updates at compile time is possible, precise compile time analysis would in general be intractible. The user may want to know about potential anomalies in explicitly parallel code, since it may indicate a programming error, even when it is legal. Thus, compiler analysis for potential anomalies may be a very useful option [8, 5, 11, 3].

Undefined: This is the Fortran view of languages: anything not required is optional, and there are no illegal options. Thus, compiler implementers are free to do whatever they want; any output is a legal output (from the compiler point of view). Often the implementers define some meaning, either actively or passively ("the definition is what the compiler does"), and then are forced into compatibility for the rest of eternity.

Sequential: Here the only legal output is one that could have arisen by a sequential execution of the statements in lexical order. This view is sometimes expressed for anomalous parallel loops. In any case, with this definition there is only one unambiguous legal output, `t=4,u=7`.

Immediate Update: This view is often taken due to hardware support for coherent multiprocessor caches. Here, any update to a variable must be visible to all other processors (or processes). This is closely related to the problem of multiprocessor memory coherence, where a sequentially consistent implementation is considered indistinguishable from a strict conflict-free shared memory multiprocessor [13]; sometimes a *weak consistency* model is used to overcome long latency operations for shared variable updates [12]. Under this model, any of the statement orderings in the table above would be legal, with three different possible outputs. The problems for the compiler are to detect what statement orderings are required, and what optimizations are illegal in the presence of other code that might execute in parallel with this code [14]. The statement reordering question mentioned above is one such example. This problem is even more insidious than it seems; if the language allows subroutines to be called in parallel (as does PCF Fortran) with potentially anomalous updates to global variables, the compiler can't even know the scope of the parallelism nor the variables that might be volatile, much less the interaction between multiple variables. Such a definition would essentially invalidate all compiler optimizations involving global

variables.

Copy-in/Copy-out: This is similar to the value/result style of parameter passing. The values of shared variables in a parallel section are defined to be initialized to the values they had when the parallel block was entered; any updates are made (conceptually) to *local copies* of the variable. When the parallel block is complete, the global state is updated with any modifications made within any section. This completely defines the values to be used for shared variables that are defined and used in different sections. If any shared variable is updated by more than one parallel section, some definition must still be made, corresponding to one of the previous choices. With this definition, the compiler can know that statement 4 can always be forward substituted into statement 5, since no anomalous updates are allowed to any variable while the section is executing. The only legal output is the same (for this program) as for sequential semantics, `t=4, u=7`.

Of these possible definitions, sequential semantics is the most well-defined, and also the most restrictive. Many current coherent memory parallel computers support immediate update semantics; however preserving immediate update semantics may be too restrictive in terms of the optimizations allowed. We advocate using copy-in/copy-out semantics. This gives a well-defined program without volatile variables, and allows optimization within a parallel section independent of code in other sections. The model has several potential problems, such as the overhead of making local copies of variables, and atomic merging of updated variables. There is more opportunity for compiler optimization here; the compiler can try to distinguish variables which are read-only in the parallel block (so no local copies need be made), those that are read and written in different parallel sections (so local copies must be made), and those that are read and written, but for which updates can be made in place. The analysis for update-in-place will be similar to that for functional languages, such as SISAL. For read-write variables which must be merged, the compiler must generate code to merge the updated values efficiently, without causing a bottleneck in the executing program. The Myrias SPS-1 control mechanism supported copy-in/copy-out semantics by clever use of the virtual paging translation hardware and operating system primitives [6]; in that system, multiple updates to a shared variable by parallel tasks gave an undefined result after the parallel block.

3 Static Single Assignment Form

After converting a program into SSA form, it has the following two properties [9]:

- Each use of a variable is reached by exactly one assignment to that variable.

```
X = 1                X = 1
Y = 1                Y = 1
if P then            Parallel sections
                     Section
   X = 2                X = 2
else                 Section
   Y = 5                Y = 5
endif                end parallel sections
Z = X+Y              Z = X+Y
```

Figure 2: Conditional and Parallel programs

- The program contains merge functions, called ϕ-functions to distinguish values of variables transmitted from *distinct* incoming control flow edges.

Cytron et al [9] present a fast algorithm to convert a program into SSA form in which the number of ϕ-functions inserted is minimal. The algorithm uses the dominance relation and dominance frontiers as reviewed briefly here.

A CFG is a directed graph with a distinguished unique *Entry* vertex. We say a vertex v *dominates* another vertex w, written $v \gg w$, if v appears on every path from *Entry* to w. By this definition, every vertex dominates itself, and *Entry* dominates every other vertex. The *dominance frontier* of a node v, DF(v), is the set of all CFG nodes z such that v dominates a predecessor of z but does not strictly dominate z. Note that v may itself be a member of DF(v).

The SSA algorithm uses dominance frontiers to determine where to place ϕ-functions. ϕ-functions for a variable X are required at all the nodes in the iterated dominance frontier of $\mathcal{S}$, where the set $\mathcal{S}$ is the union of all the nodes where X is assigned. The dominance frontier sets are constructed in a single bottom-up traversal of the dominator tree. Thus, both the dominance relation and the dominance frontiers are crucial to the conversion of a program to SSA form. We need to be able to extend these concepts to explicitly parallel code, and we want to have a meaningful SSA form of a parallel program.

4 Flow Graphs for Parallel Constructs

In the case of sequential programs, CFGs accurately model potential control flow. We might be tempted to model `Parallel Sections` in a CFG by treating the fork point as a branch node and the join as a merge node. The CFGs for the two programs in Figure 2

```
X1 = 1                    X1 = 1
Y1 = 1                    Y1 = 1
if P then                 Parallel sections
                          Section
   X2 = 2                    X2 = 2
else                      Section
   Y2 = 5                    Y2 = 5
endif                     end parallel sections
X3 = φ(X2,X1)
Y3 = φ(Y1,Y2)
Z1 = X3+Y3                Z1 = X2+Y2
```

Figure 3: SSA Forms of Conditional and Parallel programs

will then look the same. However, the execution semantics of the two programs are very different. In the sequential program, only one of the two assignments `X = 2` or `Y = 5` will be executed; the value of `Z` will be 3 or 6, depending on the branch taken. In the parallel program, however, both assignments will be executed, and the value of `Z` will always be 7; in fact, the two initial assignments to `X` and `Y` are dead code. The proper SSA forms of these two programs are shown in Figure 3. No ϕ-functions are needed in the parallel program, since only the X_2 and Y_2 assignments reach the Z_1 assignment. Clearly, trying to model this parallel program with a simple CFG is incorrect; the simple CFG would model the parallel construct just like it models conditionals, and would then add unnecessary ϕ-functions at the join point.

What we would like to have is a representation where the X_2 assignment *dominates* the Z_1 assignment, Y_2 assignment *dominates* the Z_1 assignment, but there is no dominance relation between the X_2 and Y_2 assignments at all. No Control Flow Graph will give us this kind of relationship. To handle this, we introduce Parallel Control Flow Graphs.

5 Parallel Control Flow Graphs

This section presents the *Parallel Control Flow Graph* (PCFG) that models control flow in parallel programs accurately and the *Parallel Precedence Graph* (PPG) that models concurrent execution within a parallel construct. A set of PCFGs and PPGs make up the Extended Flow Graph set (EFG) that model an entire program unit.

A *Parallel Control Flow Graph* (PCFG) is a CFG which may have a special type of

node called a *supernode*. A supernode essentially represents an entire `Parallel Sections` construct. Parallel execution of the sections within a parallel block is represented by a *Parallel Precedence Graph* (PPG). `Wait` clauses in a parallel block impose *wait-dependence* between the waiting section and the sections specified in the `Wait` clause. Nodes in the PPG represent the sections in the parallel block with two additional nodes, cobegin and coend. The edges in the PPG (also called *wait-dependence arcs*) represent the wait dependences. To conserve space, we do not discuss wait dependence arcs in detail here.

Formally a PCFG is defined as the graph $G = \langle V_G, E_G, Entry_G, Exit_G \rangle$ where

- V_G is a set of vertices, each representing a basic block (basic block node) or an entire parallel block (supernode).
- E_G is a set of edges $\{a \rightarrow b \mid a, b \in V_G\}$, representing potential flow of control in the program.
- $Entry_G \in V_G$ is the unique start node (or entry node) of the PCFG, with all vertices reachable from $Entry_G$.
- $Exit_G \in V_G$ is the exit node of the PCFG, where $Exit_G$ is reachable from all vertices in V_G.

Parallel execution within a parallel block is represented by a PPG which is formally defined as a graph $P = \langle V_P, E_P, Entry_P, Exit_P \rangle$ where

- V_P is a set of vertices, each representing a section in a parallel block (section node).
- E_P is a set of edges or wait-dependence arcs in the PPG.
- $Entry_P \in V_P$ is the cobegin node.
- $Exit_P \in V_P$ is the coend node.

By definition of the language, the PPG graph must be a DAG.

Each section S is again represented by a PCFG $S = \langle V_S, E_S, Entry_S, Exit_S \rangle$ where $Entry_S$ marks the entry into that section and $Exit_S$ marks the exit from that section.

The Extended Flow Graph set (EFG) is the set of PCFG's and PPG's representing control flow and parallelism for a single program unit. The distinguished PCFG corresponding to the program unit is called G_{main}. When we talk about the nodes in an EFG, we mean the union of all the nodes in all the PCFG's and PPG's in the EFG. The nodes

may be *Entry* or *Exit* nodes, basic block nodes, supernodes, cobegin or coend nodes, or section nodes.

The EFG of the parallel program in Figure 1(a) is shown in Figure 4. G_{main} has 5 vertices: *Entry*, m (an assignment basic block), $P1$ (a parallel block), n (a print basic block), and *Exit*. The parallel block is represented by PPG_{P1} with four vertices: the cobegin, A and B (one for each of the two sections), and coend. The two parallel sections are then again represented by PCFGs, each (in this case) with three vertices, one of which represents the basic block of assignments.

6 Dominance Relation Between Vertices of an EFG

Given a parallel program and its EFG, we want to compute the SSA form of the program. As mentioned before, the SSA algorithm depends on the dominance relation in the program. However, the vertices of the EFG are now spread over several graphs. While we have an intuitive feel for how the graphs are "nested," there is little formal basis for this. For instance, in Figure 4, by the semantics of the language, we want to have the relationships $m \gg q$ and $q \gg n$, but not to have $q \gg r$. Since m and q do not appear in the same directed graph, the canonical dominance relation between them is not defined. Here we address this problem.

One way to define the execution relationships of vertices from different PCFGs is to consider all possible combinations of paths through all the parallel blocks in the program. We use a method to derive a set of sequential CFGs from an EFG, referred to as **factoring** [19, 21]. Since there are only two paths through the only parallel block in Figure 4, there are only two factors, shown in Figure 5.

Computing the dominance relation between nodes in the sequential CFGs obtained by *factoring* the EFG can be done easily using the standard definition of dominance. We define the *parallel dominance relation* to be the dominance relation for a basic block node in an EFG, computed as the *union* of the sequential dominators of in each of the sequential CFGs obtained by factoring the EFG.

In the sequential factors in Figure 5, the dominance relations $m \gg q$ and $q \gg n$ hold in the first factor, and $m \gg r$ and $r \gg n$ in the second factor. We thus define this to be the set of parallel dominance relations for the program. Note that the parallel dominance relation can not be represented by a tree.

Similarly, we define the *parallel dominance frontier* of a basic block node to be the dominance frontier in the EFG, computed as the union of the sequential dominance frontiers

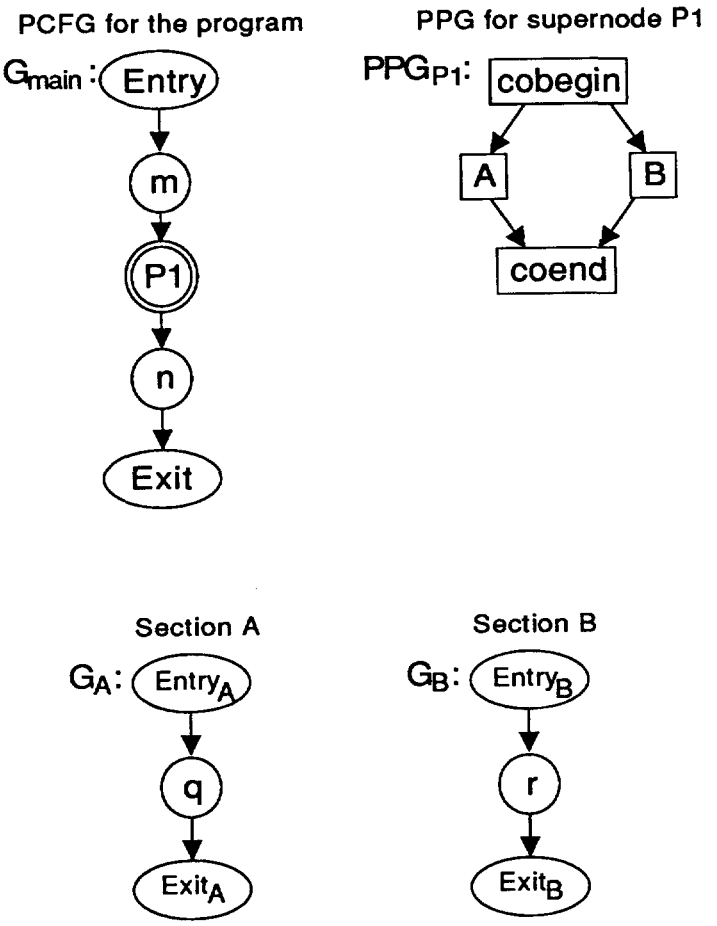

Figure 4: Extended Flow Graph set

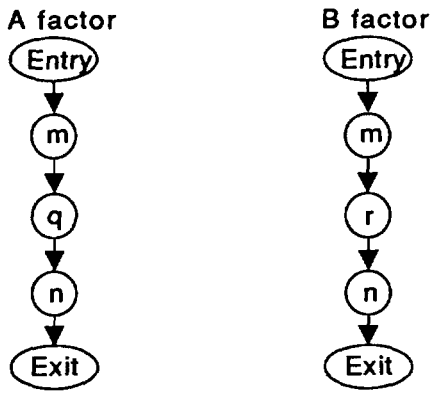

Figure 5: Factors of an EFG

in each of the sequential CFGs derived by factoring the EFG.

7 SSA with Copy-in/Copy-out Semantics

The advantage of using the copy-in/copy-out semantics is that the SSA form of the program can be easily constructed from the parallel dominance frontier. In fact, the same algorithms used to construct the SSA form in [9] can be easily modified to construct the SSA form of a parallel program. The key property of copy-in/copy-out semantics that makes this possible is that the only definitions for a variable that can reach a use within a parallel block must reach that use by some path from the cobegin vertex. We took advantage of this when defining the parallel dominance relation. Some examples will show the advantages; our first example program can be simply converted to SSA form as shown in Figure 1(b). Note for instance that the only definition of `w` that reaches statement 6 is from statement 2. No ϕ-functions are needed here since there is no conditional code. The ψ-functions will be explained shortly.

A second example shows conditional code within and surrounding the parallel block, in Figure 7. In every case, the only ϕ-functions needed in the program are those that are needed in some factor of the graph.

```
a = 1
b = 2
if a = c then
   b = b + 1
else
   Parallel Sections
   Section A
      a = a + 1
      if e > 0 then
         e = 99
      endif

   Section B
      c = a
      if b = c then
         e = 1
      endif

      d = e + 1
   End Parallel Sections

endif

print a,b,c,d,e
```

```
a1 = 1
b1 = 2
if a1 = c0 then
   b2 = b1 + 1
else
   Parallel Sections
   Section A
      a2 = a1 + 1
      if e0 > 0 then
         e1 = 99
      endif
      e2 = φ(e1, e0)
   Section B
      c1 = a1
      if b1 = c1 then
         e3 = 1
      endif
      e4 = φ(e3, e0)
      d1 = e4 + 1
   End Parallel Sections
   e5 = ψ(e2, e4)
endif
a3 = φ(a1, a2)
b3 = φ(b2, b1)
c2 = φ(c0, c1)
d2 = φ(d0, d1)
e6 = φ(e0, e5)
print a2, b3, c2, d2, e6
```

Figure 6: Original and SSA Form of Extended Example

8 Anomalous Updates

When only one section updates a shared variable, the value of that variable after the parallel block is complete is well defined. Even if other sections use that variable, they will always get the "old" value of the variable (in the absence of any synchronization or `wait` clauses); for example, see variable `a` in Figure 7. Section A increments `a`, but Section B still sees the old value of `a`, namely a_1, which is equal to 1. In fact, constant propagation could be used to forward substitute the constant value 1 into all uses of a_1; this would then allow a_2 to be computed as the value 2, and so on.

However, when two parallel sections update the same variable, the value of the variable after the parallel block is indeterminate. In Figure 1(b), variables `v` and `w` were updated in both sections; within the sections, the values to be used for each variable is well defined. A use of `v` outside the parallel block, however, might be reached by either v_2 or v_3; to keep the Static Single Assignment rules, where each use is reached by only a single assignment, we must somehow merge these two updates at the `End Parallel Sections` statement. This is the reason for the ψ-functions in the parallel SSA programs.

The presence of a ψ-function may indicate to the compiler that the program contains an actual or potential anomaly. A useful compiler option would be to flag all potential anomalies such as this. Note that such anomalies may only be potential; in Figure 7, the variable `e` is only potentially updated in the two sections. If the two conditions are mutually exclusive, then in fact only one update will be done, and the semantics of the language should preserve whichever update is performed. Nonetheless, the ψ-function is needed to preserve the SSA properties.

9 Conclusion

Previous work has shown that the Static Single Assignment intermediate representation forms a practical basis for optimizing sequential programs. We have shown how to extend Control Flow Graphs, dominance relation and dominance frontiers used in computing SSA to include parallel constructs. SSA is discussed in detail in [9]. By extending SSA to include parallel constructs, the resulting intermediate language will form a powerful platform for many classical code optimization algorithms that can be run on parallel programs.

Our current work has found an efficient and simple algorithm for finding the parallel dominators and parallel dominance frontiers [22]. We are implementing this algorithm as well as the algorithm to find anomalous updates in parallel programs in our prototype

compiler, Nascent.

References

[1] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the PTRAN analysis system for multiprocessing. In Elias N. Houstis, Theodore S. Papatheodorou, and Constantine D. Polychronopoulos, editors, *Supercomputing: 1st International Conf.*, volume 297 of *Lecture Notes in Computer Science*, pages 194–211. Springer-Verlag, Berlin, 1987.

[2] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the PTRAN analysis system for multiprocessing. *J. Parallel and Distributed Computing*, 5(5):617–640, October 1988. (update of [1]).

[3] Todd R. Allen and David A. Padua. Debugging Fortran on a shared memory machine. In Sartaj K. Sahni, editor, *Proc. 1987 International Conf. on Parallel Processing*, pages 721–727, St. Charles, IL, August 1987.

[4] B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *Conf. Record 15th Annual ACM Symp. Principles of Programming Languages* [16], pages 1–11.

[5] Vasanth Balasundaram and Ken Kennedy. Compile-time detection of race conditions in a parallel program. In *Proc. 3rd International Conference on Supercomputing*, pages 175–185, June 1989.

[6] Monica Beltrametti, Kenneth Bobey, and John R. Zorbas. The control mechanism for the myrias parallel computer system. *Computer Architecture News*, 16(4):21–30, September 1988.

[7] Per Brinch Hansen. *Operating Systems Principles.* Automatic Computation. Prentice-Hall, 1973.

[8] David Callahan, Ken Kennedy, and Jaspal Subhlok. Analysis of event synchronization in a parallel programming tool. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* [17], pages 21–30.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conf. Record 16th Annual ACM Symp. on Principles of Programming Languages*, pages 25–35, Austin, TX, January 1989.

[10] Ron Cytron, Michael Hind, and Wilson Hsieh. Automatic generation of DAG parallelism. In *Proc. ACM SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 54–68, Portland, OR, June 1989.

[11] Anne Dinning and Edith Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* [17], pages 1–10.

[12] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Conf. Proc. 13th Annual International Symp. on Computer Architecture*, pages 434–442, Tokyo, June 1986.

[13] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

[14] Samuel P. Midkiff, David A. Padua, and Ron Cytron. Compiling programs with user parallelism. In David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, pages 402–422. MIT Press, Boston, 1990.

[15] Parallel Computing Forum. *PCF Fortran*, April 1990.

[16] *Conf. Record 15th Annual ACM Symp. Principles of Programming Languages*, San Diego, CA, January 1988.

[17] *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, Washington, March 1990. ACM Press.

[18] B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Global value numbers and redundant computations. In *Conf. Record 15th Annual ACM Symp. Principles of Programming Languages* [16], pages 12–27.

[19] Harini Srinivasan. Analyzing programs with explicit parallelism. M.S. thesis 91-TH-006, Oregon Graduate Institute, Dept. of Computer Science and Engineering, July 1991.

[20] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. on Programming Languages and Systems*, 13(2):181–210, April 1991.

[21] Michael Wolfe, James Hook, and Harini Srinivasan. Flow graph relations for explicitly parallel programs, 1991. submitted for publication.

[22] Michael Wolfe and Harini Srinivasan. Data structures for optimizing programs with explicit parallelism. In *Proc. First International Conference of the Austrian Center for Parallel Computation*, Salzburg, September 1991. to appear.

[23] Michael Wolfe and Harini Srinivasan. Static single assignment form for explicitly parallel programs, 1991. submitted for publication.