Static Interprocedural Slicing of Shared Memory Parallel Programs*

Dixie M. Hisley
U.S. Army Research Lab
APG, MD 21005
hisley@arl.mil

Matthew J. Bridges
Computer & Information Sciences
University of Delaware
Newark, DE 19716
bridges@cis.udel.edu

Lori L. Pollock
Computer & Information Sciences
University of Delaware
Newark, DE 19716
pollock@cis.udel.edu

Abstract

Software tools for program debugging, software testing, software maintenance, and program understanding have all effectively utilized static program slicing techniques. In this paper, we present an approach to extend this capability to explicitly parallel shared memory programs written using the OpenMP standard. In particular, interprocedural static program slicing of OpenMP programs is enabled by extending standard program representations for control flow and program dependences to support OpenMP parallel, data, and synchronization constructs. The slicing algorithm builds on the algorithms for interprocedural slicing of sequential programs and an algorithm for intraprocedural slicing of parallel programs.

Keywords – shared memory parallel programming, program slicing, dependence graphs, compilers, software tools

1. Introduction

Static interprocedural slicing for *sequential* codes is well understood and used in a variety of applications [10, 2]. A static program slice is defined as follows: let P be a program, p be a point in P, and v be a variable that is defined or used at point p. Then, a static slice relative to the slicing criterion <p,v> is the set of all statements in the program P that might affect the value of v defined or used at point p. Slicing is used for software development and maintenance activities such as program understanding, software testing, and debugging. To the authors' knowledge, this is the first research that addresses static interprocedural slicing for shared memory parallel programs, written using OpenMP [8] explicitly parallel constructs. OpenMP is the standard for developing efficient, portable parallel programs for shared memory multiprocessors.

Few sophisticated tools exist to aid the programmer in debugging and optimizing shared memory parallel codes. Midkiff and Padua [6] demonstrated that straightforward application of sequential optimization techniques within compilers for explicitly parallel shared memory programming fail to maintain correctness. Potential data races and synchronization of shared variables must be taken into account.

Algorithms for slicing shared memory parallel programs must address the possible interactions between shared variables among threads that potentially execute in parallel. In this paper, an intermediate representation for the program, a *threaded system dependence graph* (tSDG) is developed, and the possible interactions between shared variables are analyzed as part of the slicing algorithm. The concepts of a **parallel** region and worksharing that are introduced by OpenMP are handled as well as potential subroutine calls within these constructs. Although Krinke [3] and Cheng [1] have previously investigated slicing parallel codes, Krinke's approach was limited to intraprocedural slicing, and Cheng concentrated on slicing for object-oriented parallel codes.

In the following sections, we describe background and related work, the targeted parallel programming environment, challenges of interprocedural slicing, the threaded system dependence graph (tSDG), an algorithm to slice on this representation, and finally conclusions.

2. Background and related work

For sequential programs, techniques for the static slicing of Fortran and Simple-D programs were first

* This work was supported in part by NSF EIA-9703088 and CCR-0105540.

introduced by Weiser [11]. His techniques were based on control flow graph intermediate representations, but did not take into account the calling context of called subroutines. Intraprocedural slicing based on operations performed on a program dependence graph (PDG) representation was first introduced by Ottenstein and Ottenstein [9].

Horwitz [2] introduced algorithms to compute interprocedural slices by extending the PDG to a system of PDGs called the system dependence graph (SDG). Calling context of the procedures, lacking in Weiser's methods, was captured through the use of an attribute grammar and calculation of sets of variables that might be modified (GMOD), or referenced (GREF) by a procedure. Livadas [5] developed an algorithm to perform interprocedural slicing, also based on the SDG, but using a simple approach to address calling context. Tip surveys slicing techniques for imperative programs [10].

The program dependence graph has been extended for the analysis of parallel programs [1,3]. Two approaches for static slicing of parallel programs are by Krinke [3] and Cheng [1]. Krinke introduced the threaded PDG (tPDG) as the base for his intraprocedural slicing algorithm. Cheng proposed an intermediate representation called the Process Dependence Net (PDN) which was later extended to a System Dependence Net (SDN). We have exploited Krinke's techniques as a basis for developing our techniques for interprocedural slicing of OpenMP programs.

3. Challenges in interprocedural slicing of shared memory parallel programs

A unique challenge in slicing of shared memory parallel programs is how to handle shared variables between threads that might potentially execute in parallel. The analysis of programs where some statements may execute in parallel can be handled by modeling every possible execution order of statements, but this will be too memory intensive, and is not a practical solution. Therefore, representations of parallel programs must be developed to capture the new intraprocedural and interprocedural data dependences introduced by shared memory parallelism.

There has been a lack of convention for representing the control flow of parallel programs. Current representations of parallel programs for program analysis and optimization are defined for generic parallel constructs, as opposed to an implementation of a complete parallel library/language (like OpenMP). OpenMP introduces new parallel constructs, in particular, **parallel** regions and worksharing constructs, that have not been previously represented in intermediate representations for parallel programs. In section 4, we present descriptions and examples of our representations to capture both the new intraprocedural and interprocedural data dependences introduced by OpenMP. In addition to declarations of local, global, and static data, we also need to support private, threadprivate and shared variables in a representation for OpenMP programs.

4. Modeling OpenMP parallel programs

In this section, the intermediate representations we use for intraprocedural parallel slicing and interprocedural parallel slicing, the threaded procedure dependence graph (tPDG) and the threaded system dependence graph (tSDG), respectively are described. We discuss how they are related to their sequential counterparts and the modifications required in order to generate threaded versions for representing OpenMP programs.

A commonly used intermediate representation for sequential programs is the control flow graph (CFG). The *concurrent control flow graph*, CCFG, as defined by Lee and Novillo [4, 7], was developed to represent the control structure of a generic shared memory parallel program (no particular language) that uses the parallel constructs **cobegin/coend** and **parloop/parend,** and the synchronization constructs **set, wait, barrier, lock and unlock.**

A CCFG is similar to its sequential counterpart, the CFG. A separate CCFG is generated for each procedure. The CCFG is a directed graph $G = \langle N, E, Entry_{G,} Exit_G \rangle$ such that N is the set of nodes, E is the set of control flow edges, conflict edges, and synchronization edges, and $Entry_G$, $Exit_G$ are the unique entry and exit points of the program. A *concurrent basic block* (node) is essentially a sequential basic block, with the following additional constraints: a) contains at most one shared variable access, b) contains at most one **wait** statement at the start of the block and at most one **set** statement at the end of the block c) synchronization operations **lock, unlock**, and **barrier** and parallel control instructions **cobegin, coend, parloop, parend** are placed in their own block.

A *conflict* edge is a bidirectional edge in the CCFG that joins any two concurrent basic blocks that can be executed in parallel and reference the same shared variable (with one of the references being a write). In addition, a CCFG contains synchronization edges between concurrent basic blocks that contain explicit synchronization operations.

In order to limit the modifications to the CCFG representation, we provide a mapping from OpenMP constructs to these constructs whenever possible. The combined parallel worksharing OpenMP constructs **parallel sections** and **parallel for** have direct one-to-one correspondence with the semantics of the CCFG's representation of **cobegin/coend** and **parloop/parend** constructs, respectively. The OpenMP worksharing constructs **for**, **sections**, and **single** can all be embedded within a **parallel** region. We chose to represent the OpenMP parallel regions construct as a cobegin/coend with two or more threads, where each thread gets a copy of the statement block associated with the parallel region as shown in Figure 1a. The possibility of more than two threads occurs when a **sections** construct with more than two section bodies is embedded within the **parallel** construct. There must be one thread for each unique section. For the cases where no **sections** are embedded in the parallel region, we simply replicate the body of the original **parallel** region as shown in Figure 1b. In the case of a **for**, we represent the parallel loop by replicating the original body of the loop and considering it to be like a **cobegin/coend** structure with two threads as shown in Figure 1c. The **for** and **parallel** representations have the drawback of potentially increasing memory requirements, but it is easier to analyze and support than self-referencing conflict edges through this representation. Each statement in the copy of a **for** or **parallel** body corresponds to a dual statement in the original body. For a compiler optimization to be applied, it must be applied to both dual statements. If this is not possible, we do not perform the optimization. The **single** and **master** constructs are represented as a **cobegin/coend** with one thread. With the available synchronization constructs in the CCFG, we can simulate the semantics of OpenMP's **barrier, flush**, and **critical** synchronization constructs.

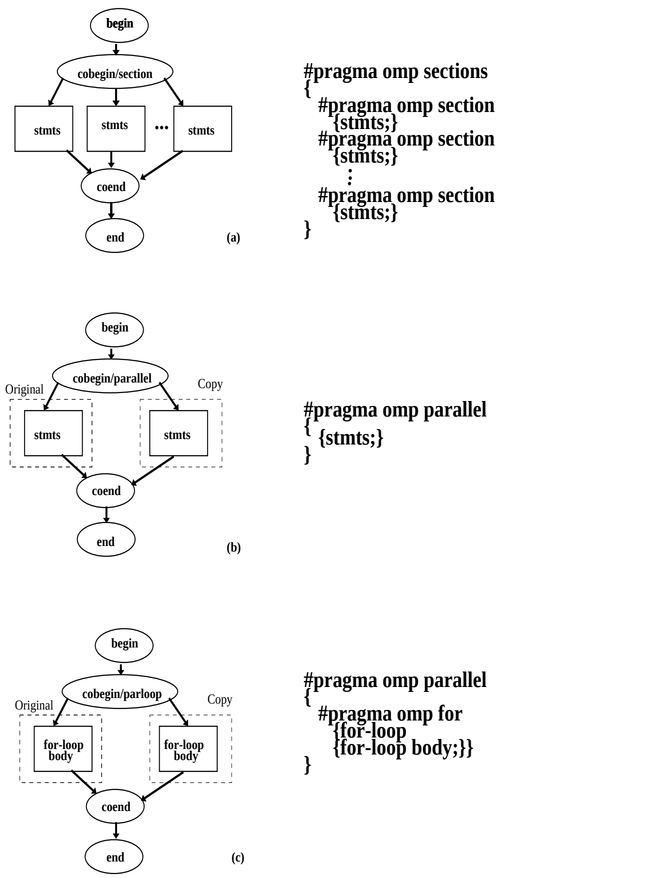

Figure 1. OpenMP (a) Sections (b) Parallel (c) For Modeled as Cobegin/Coend

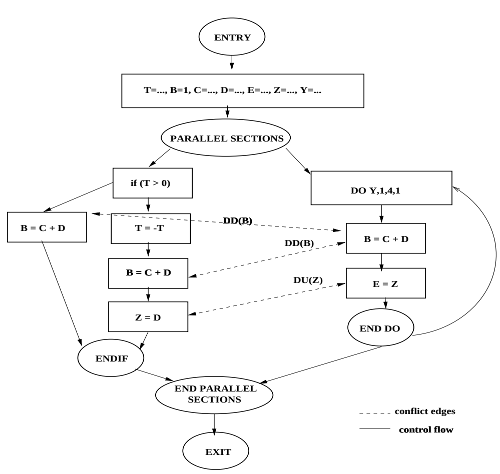

Figure 2. Example CCFG with Conflict Edges

Figure 2 presents an example intraprocedural CCFG with conflict edges for the sample code shown in Figure 3. The solid edges in the CCFG are control flow edges, while conflict edges are represented by dashed edges. For example, there is a def-def edge linking the two assignments to variable *B* in the two different threads. A def-use edge for variable *Z* indicates the relationship between the two threads created by the statements involving variable *Z*.

The PDG [9] consists of nodes for each statement and each predicate and edges to indicate control and data dependences. The tPDG was developed by Krinke [3] as a threaded counterpart of the PDG for sequential programs. It serves as the basis for his algorithm that

```
S1:  T=…, B=…,C=…,D=…,E=…,Z=, Y=…
S2:  !$OMP PARALLEL SECTIONS
S3:  !$OMP SHARED   (T,B,C,D,E, Z,Y)
S4: IF (T > 0) THEN
S5:    T = -T
S6:    B = C+ D
S7:    Z = D
S8: ELSE
S9:    B = C + D
S10: ENDIF
S11: !$OMP SECTION
S12: DO Y = 1,4
S13:    B = C + D
S14:    C = Z
S15: ENDO
S16: !$OMP END PARALLEL SECTIONS
```

Figure 3. Example OpenMP

performs static intraprocedural slicing of threaded programs.

To construct the tPDG, Krinke introduces the concept of a threaded CFG (tCFG). His tCFG is similar to our CCFG for OpenMP programs except he uses **costart/coexit** nodes to represent **cobegin**/**coend** statements for parallel sections. He does not describe how to handle **for** loops, **parallel** regions or synchronization. In his tPDG representation, control dependence edges and data dependence edges are added to the existing control/parallel flow edges and conflict edges in his tCFG using standard algorithms. Krinke's interference dependence edges are identical to the def-use conflict edges we described as part of the CCFG representation. Thus, we use the CCFG representation in place of Krinke's tCFG representation as a basis, but otherwise follow his algorithms for building our version of his tPDG and subsequently performing slicing using the tPDG.

A tPDG is used to represent the main driver procedure and each of the other procedures in a whole program. To perform interprocedural analysis, we extend the concept of a tPDG to a tSDG for the purpose of performing interprocedural static slicing of parallel programs.

We develop the tSDG as a threaded counterpart of the SDG for sequential programs [2,5]. It serves as the basis for our algorithm that performs static slicing of interprocedural threaded programs. In order to build our tSDG, we start with our CCFGs for the main program and each procedure, and build the tSDG. We briefly summarize these techniques in Figure 4.

To properly handle return statements with or without return values, we use edges defined by Livadas [5]: a) the affect-return edge b) return-control edge and c) return-link edge. The summary information at a call site is the union of transitive dependences, affect-return dependences, and return-link dependences. A transitive dependence edge exists from an actual-in node to an actual-out node if the formal-out node that corresponds to the actual-out node is intraslice-path reachable from the formal-in node that corresponds to the actual-in node. We say that there exists an intraslice-path from the formal-out node to the formal-in node in a tSDG if there is a backwards path from the latter to the former consisting of the following types of edges; control dependence, data dependence, declaration dependence, affect-return, and return-control, and interference dependence. Determination of the transitive dependences of a procedure is equivalent to determining for each formal-out node, all the formal-in nodes from which the formal-out node is intraslice-path reachable. If a slice on the formal-out node is never affected by the formal-in node, then the actual-out node can be deleted.

The algorithm described in Figure 4 for building our tSDG from our CCFGs is similar to building a SDG from PDGs with some modifications to account for the threaded nature of our representations. We treat shared variables (including threadprivate) similar to the treatment suggested in previous work [2] for global and static variables. Shared variables are handled by introducing them as additional pass-by-reference parameters to the procedures that use or define them. Furthermore, we have to modify the definition of transitive dependences to include those introduced by shared variable dependences. In order to do this, we have to include interference dependence edges in our definition of intraslice-paths, which leads to summary information at a call site that captures the shared memory parallel dependences in a tSDG. To compute transitive dependence edges, we apply Krinke's static intraprocedural threaded slicing algorithm [3].

Algorithm Construct_tSDG

Input: CCFG for each procedure in the program P.
Output: threaded System Dependence Graph for P

1: Initialize call sequence graph (CSG), a linked list of call sites to main
2: Begin partial solution of main CCFG by initiating computations of control/parallel flow, data, and interference dependences
3: Upon finding a call to a new procedure, calculation of the dependences of calling procedure is suspended (partial solution preserved); called procedure is pushed onto top of CSG;
4: **if** called procedure is already solved, reflect summary information of called procedure back onto callee site, pop top of CSG, resume calculation of dependences in calling procedure.
5: **else** call site, entry nodes are created,
6: **for each** passed by reference actual parameter, an actual-in node is created and an actual-out node is created, also the corresponding formal-in nodes and formal-out nodes are built.
7: introduce a call edge from the call site vertex to the corresponding procedure-entry vertex
8: **for each** actual-in node at a call site, introduce a parameter-in edge from the actual-in to its corresponding formal-in node
9: **for each** formal-out node, introduce a parameter-out edge from each formal-out node to its corresponding actual-out node.
10: new dependence calculation is initiated at called procedure, compute control, data, and interference dependences in called procedure, if formal-out node is never modified, mark it.
11: compute transitive dependence edges (edges from actual-in nodes to actual-out nodes) by determining for formal-out nodes, if it is intraslice-path reachable from formal-in nodes using Krinke's static intraprocedural slicing algorithm [3]. If formal-out node is not modified, delete actual-out node.
12: reflect summary information (transitive dependences, affect-return dependences and return-link dependences) of called procedure back onto callee site; pop the top of CSG.
13: **endif**
14: resume calculation of dependences in calling procedure (current top of CSG), until CSG becomes empty.

Figure 4. Algorithm for constructing tSDG

5. Interprocedural slicing with the tSDG

Given the construction of the tSDG as described in Figure 4, we can perform interprocedural slicing of the tSDG based on the algorithm suggested by Livadas [5] for sequential programs. The presence of summary edges that represent transitive data dependences, including shared variable data dependences, due to procedure calls permits interprocedural slices to be computed in two passes, each of which is a simple reachability problem. Interprocedural slicing uses summary edges to move across a call without descending into it. This prevents descending into a procedure and returning by way of some unrealizable execution path (sidesteps the call context problem). In pass 1, the backwards traversal starts at node x, the statement where the slice initiated in P, and follows the following types of edges: control dependence, data dependence, declaration dependence, return-control, parameter-in, transitive dependence, affect-return, and/or call. Thus, traversal does not descend into procedures called by P.

In pass 2 of the algorithm, nodes are identified that can reach node x from procedures called by P, or from procedures that call P transitively. The backwards traversal follows the following types of edges: control dependence, data dependence, declaration dependence, return-control, parameter-out, transitive dependence, affect-return, and/or return-link. Thus, traversal does not ascend into procedures that called P. The transitive flow dependence edges from actual-in to actual-out nodes make ascents unnecessary. When either pass is traversing a return-control edge, the node at the end of the return-control edge (a return statement) is marked as being in the slice and the slicing follows the control dependence edge to its origin, where slicing continues. Finally, the nodes of the interprocedural slice are the union of the nodes from both passes. Slicing can be performed even if a program contains calls to unknown or system procedures, as long as transitive dependences are known.

In Figure 6, we illustrate interprocedural slicing on a tSDG representation for the simple multi-procedure OpenMP program shown in Figure 5. Control, data, interference, parameter in/out, call, and transitive dependence edges are all shown. Note that this program is written in OpenMP C, so all parameters are assumed to be passed by value, except the parameter *c* which is passed by reference. Thus, there is no data dependence from *c=12* to the *printf* because the pass by reference parameter *c* is assumed to be killed by the execution of *work*. Also, the *a=u_out, b=V_out,* and *d=x_out* are all missing because they are passed by value. At statement 14 in Figure 5, variable *v* is designated as a shared variable. Thus, an interference dependence potentially exists between statements 17 and 23, and between statements 20 and 23. As shown in Figure 6, interference dependence edges are inserted between the corresponding nodes in the tSDG. A key difference between a tSDG and an SDG representation is the inclusion of interference dependence edges in the tSDG. The definition of "intra-slice path reachable" is modified to include interference dependence edges.

Consider a slice starting at statement 8. We are interested in identifying which nodes in the tSDG can potentially impact the value of *b* at this point in the program. Thus, our slicing criterion is <*S8, b*>. The shaded nodes in Figure 6 represent the result of performing the two-pass slicing algorithm described in this section. The statement 8 is data dependent on the formal-out vertex for parameter *w*. The formal-out vertex for parameter *w* is intra-slice path reachable from the formal-in vertices for *u,v,* and *w*. If interference dependence edges had not been included in the definition

```
1:  int main () {
2:      int a, b, c, d;
3:      a =10;
4:      b = 6;
5:      c=12;
6:      d=23;
7:      work(a, b, &c, d);
8:      b = c / 2;
9:      printf("%d : %d : %d : %d", a, b, c, d);
10:     return 0;
11:     }
12:
13: void work (int u, int *v, int w, int x) {
14: #pragma omp parallel sections  shared (v)
15: #pragma omp section
16:     if (u > v) {
17:         v = u + v;
18:     }
19:     else {
20:         v = v * v;
21:     }
22:     #pragma omp section
23:         *w = v + *w;
24:         x = x + *w;
25:     #pragma omp end parallel
26:     }
```

Figure 5. Example Multi-Procedure OpenMP

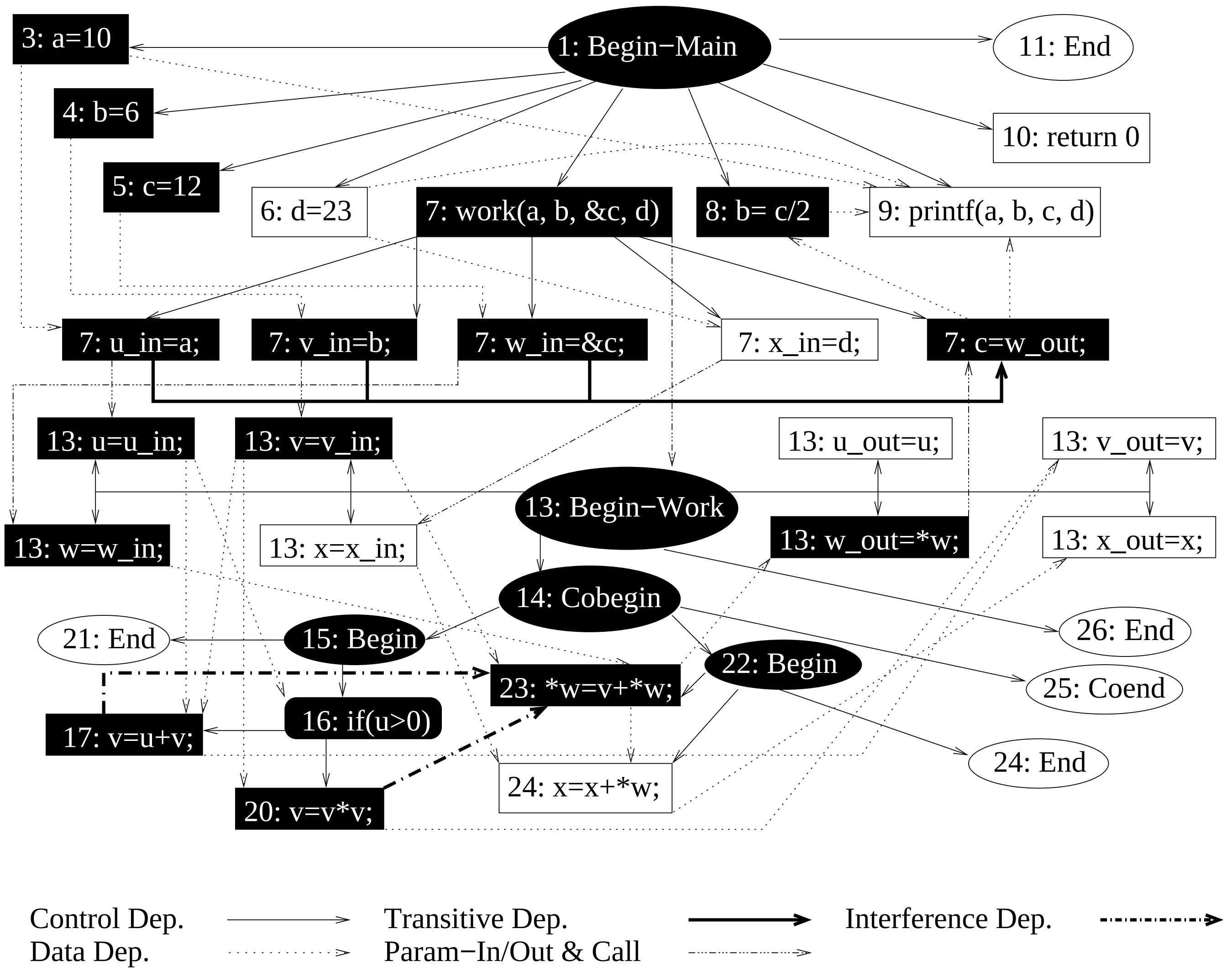

Figure 6. tSDG Program Slice of Example Multi-Procedure OpenMP Program

of intra-slice path reachable, then only the formal-in vertices for parameters *v* and *w* would have been found. Thus, the inclusion of interference dependence edges for shared variables allows the correct propagation of summary information for procedure *work* with respect to slicing on variable *b* at statement 8.

6. Conclusions and future work

A major contribution of this paper is the development of CCFG and tSDG intermediate representations to enable interprocedural slicing of OpenMP shared memory parallel programs. We showed how to represent OpenMP's **parallel** and worksharing constructs in CCFGs. We redefined transitive dependences in the tSDG to include the interference dependences that can occur in shared memory parallel programs. The inclusion of interference dependence edges in our definition of intraslice-paths for computing transitive dependences (summary information) at a call site allows us to capture the shared memory parallelism dependences in the tSDG. Once the tSDG is built, it can be used as input to perform interprocedural slicing of OpenMP programs. The interprocedural slicing algorithm described in this paper is based on a combination of the algorithms suggested by Krinke [3], and Livadas [5].

We are implementing our techniques by extending the Odyssey [7] compiler representation for CCFGs, building a tSDG phase and slicing tool. We hope to extend our techniques to handle additional realistic parallel programming paradigms like object-oriented concurrency and combined OpenMP and MPI programs.

7. References

[1] J. Cheng. Dependence analysis of parallel and distributed programs and its applications. In *Intl. Conference on Advances in Parallel & Distributed Computing*, 1997.

[2] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs, *ACM TOPLAS*, January 1990.

[3] J. Krinke. Static slicing of threaded programs. In *Proc. of ACM SIGPLAN Workshop on Program Analysis for Software Tools & Engineering*, Montreal, CA, June 1998.

[4] J. Lee. *Compilation Techniques for Explicitly Parallel Programs.* Ph.D. thesis, Department of Computer Science, Univ. of Illinois at Urbana-Champaign, 1999.

[5] P. Livadas and S. Croll. A new algorithm for the calculation of transitive dependences, *Journal of Software Maintenance*, vol 6, , pp. 100-127, 1994.

[6] S. Midkiff and D. Padua. Issues in the optimization of parallel programs. In *Proc. Int. Conf. On Parallel Processing*, vol. II, pp. 105-113, 1990.

[7] D. Novillo. *Analysis and Optimization of Explicitly Parallel Programs.* Ph.D. thesis, Depart. of Computing Science, University of Alberta, Edmonton, Canada, 2000.

[8] OpenMP Standard Board. *OpenMP Fortran Application Program Interface,* October 1997, Version 1.0, http://www.openmp.org..

[9] K Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM SIGPLAN Notices 19,5, May 1984.

[10] F. Tip. A survey of program slicing techiques. *Journal of Programming Languages,* vol 3., no. 3, pp 121-189, September 1995.

[11] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, vol. 10, pp. 352-357, 1984.