To appear in the proceedings of:
International Conference on Parallel Architectures and Compilation Techniques (PACT)
Montreal, Canada, August 23-26, 1994

A PDG-Based Tool and Its Use in Analyzing Program Control Dependences*

Chris J. Newburn, Derek B. Noonburg, and John P. Shen; {newburn, derekn, shen}@ece.cmu.edu

Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213

Abstract: This paper explores the potential of a program representation called the program dependence graph (PDG) for representing and exposing programs' hierarchical control dependence structure. It presents several extensions to current PDG designs, including a node labeling scheme that simplifies and generalizes PDG traversal. A post-pass PDG-based tool called PEDIGREE has been implemented. It is used to generate and analyze the PDGs for several benchmarks, including the SPEC92 suite. In particular, initial results characterize the control dependence structure of these programs to provide insight into the scheduling benefits of employing speculative execution, and exploiting control equivalence information. Some of the benefits of using the PDG instead of the CFG are demonstrated. Our ultimate aim is to use this tool for exploiting multi-grained parallelism.

1. Introduction

A program representation called the program dependence graph (PDG) [6] explicitly represents all control dependences, and readily identifies control-equivalent basic blocks that may be far away from each other in the CFG. The hierarchical representation of the program control dependence structure in a PDG allows a large fragment of the code to be treated as a single unit. Since the work by Ferrante et al. [6], a number of other researchers have studied the use of PDG for code motion [8, 2, 5], program partitioning [13], code vectorization [4], register allocation [11], program slicing and software engineering [9,12], and code translation for dataflow machines [3].

PDGs appear well-suited for the exploitation of program parallelism of different granularities. Current compilers exploit fine-grained parallelism by overlapping the execution of instructions within a basic block or a small number of adjacent basic blocks, and medium-grained parallelism by overlapping the execution of loop iterations. Recent studies [10] suggest that additional parallelism can be harvested if the compiler is able to exploit control equivalence and use multiple instruction streams. This parallelism may not be strictly fine-grained or medium-grained. Multi-grained parallelism is an informal generalization of parallelism that can be achieved by the parallel execution of code fragments of varying sizes or granularities, ranging from individual instructions to loop iterations, to mixtures of loops and conditionals, to large fragments of code. PDGs appear to be well-suited for exploiting such parallelism, since such fragments are subgraphs of the PDG that can be hierarchically summarized, and because they explicitly maintain accurate control dependence information.

Our eventual goal is to develop a PDG-based compilation tool capable of performing powerful code transformations to exploit multi-grained parallelism. This paper presents our initial results, most of which focus on programs' control dependence structure. It proposes a number of extensions to the current PDG design in order to facilitate more generalized code motion of varying granularities, for even unstructured and irreducible programs. A node labeling scheme is introduced that simplifies and generalizes PDG traversal.

The initial portion of a prototype PDG-based tool called PEDIGREE is complete. It is a post-pass tool which can be used in conjunction with existing compiler front ends or disassemblers to perform optimizations on existing source or object code. Currently, PEDIGREE is able to parse large, unstructured and irreducible programs into its internal PDG representation. The PDGs of these programs are analyzed, and their control dependence structures are characterized. Section 2 introduces the PDG and presents our extensions to the PDG. The PDG node labeling

* Supported by National Science Foundation Graduate Research Fellowships, and ONR contract No. N0001491-J-1518.

scheme is summarized in Section 3. Section 4 briefly describes the implementation of PEDIGREE. Section 5 analyzes the PDGs generated by PEDIGREE for a number of benchmarks, including some from the SPEC92 suite. The advantages of using the PDG over the CFG are highlighted using collected statistics in Section 6. Section 7 provides a summary and suggests potential code transformations that can be implemented based on PEDIGREE.

2. The Program Dependence Graph

The program dependence graph [6] is essentially a hierarchical control dependence graph (CDG) and data dependence graph. The explicit representation of control dependences facilitates the identification of control-equivalent code. The hierarchical control dependence structure identifies and groups code constructs for special processing.

2.1. PDG Definitions

A control flow graph (CFG) represents a single procedure. Nodes in a CFG are basic blocks. An arc between nodes X and Y indicates one of the following: (a) a conditional or multi-way branch from X to Y, (b) an unconditional branch from X to Y, (c) X does not end with a branch and is immediately followed by Y, (d) X ends with a procedure call, and Y is the return target of the call. Only type (a) arcs represent true control dependences. They are traversed only on a specified condition, while the traversal of arcs of type (b), (c), and (d) does not depend on any condition. Figure 1b shows a typical CFG for the code in Figure 1a.

Nodes in a control dependence graph (CDG) may also be basic blocks. In a CDG, however, an arc from node X to node Y indicates that Y is immediately control dependent on X. This means that X ends with a conditional or multi-way branch, and that Y's execution depends on the value that determines the path out of X. This value is indicated by the arc label. Figure 1c shows the CDG derived from the CFG in Figure 1b. In the CFG, B6 follows B3, B4, and B5. However B6 is not control dependent on any of these; in fact, it is control equivalent to B3. CDG siblings that have exactly the same incoming arcs are control equivalent.

The program dependence graph (PDG) extends the CDG with hierarchical summary and data dependence information. The PDG has additional nodes and arcs to represent hierarchical relationships. PDG nodes hierarchically summarize regions of the CDG, rather than summarizing strongly-connected CFG components like a hierarchical task graph [7]. PDG nodes are classified into three basic types [8]. *Code* nodes represent basic blocks, and are leaf nodes in the PDG. The control dependences in the CDG are represented in arcs which emanate from *Predicate* nodes. *Region* nodes group together all nodes which share a common set of control dependences. Each *Code* node contains the set of instructions from the corresponding basic block. Conditional branch instructions are associated with *Predicate* nodes rather than *Code* nodes, since descendants of *Predicate* nodes are control dependent on the branch. Instructions can be broken into smaller sets, with *Code* nodes representing sub-blocks. Other PDG implementations, e.g. [2,4,6,8,13], generally use leaf nodes that represent statements rather than basic blocks. The representation of data dependences is orthogonal to the control dependence structure. Data dependences may be represented only within basic blocks if control flow order is maintained, or the representation may be as sophisticated as Gated Single Assignment form [3].

PDG arcs represent two distinct concepts. *Control dependence arcs* represent control dependences, and thus must emanate from a *Predicate* node. They correspond to the arcs of the CDG, and are labeled similarly. Conditional branches use two arc labels, T and F. Multi-way branches, e.g. from case statements, have a unique arc label for each branch target. Unlabeled arcs, which emanate from *Region* nodes, are called *grouping arcs*. They represent transitive, rather than immediate, control dependence for a set of nodes with a common set of control dependences. Arcs which emanate from the same node can be ordered according to the original control flow order [8]. This ordering is necessary only to preserve inter-node data dependences if they are not explicitly represented. It also simplifies data flow analysis and allows regeneration of a CFG identical to that from which the PDG is generated.

Figure 1d shows the original PDG representation [6] constructed from the CDG in Figure 1c. It has the same overall structure, but *Region* nodes have been added for each set of CDG nodes with control dependences in common.

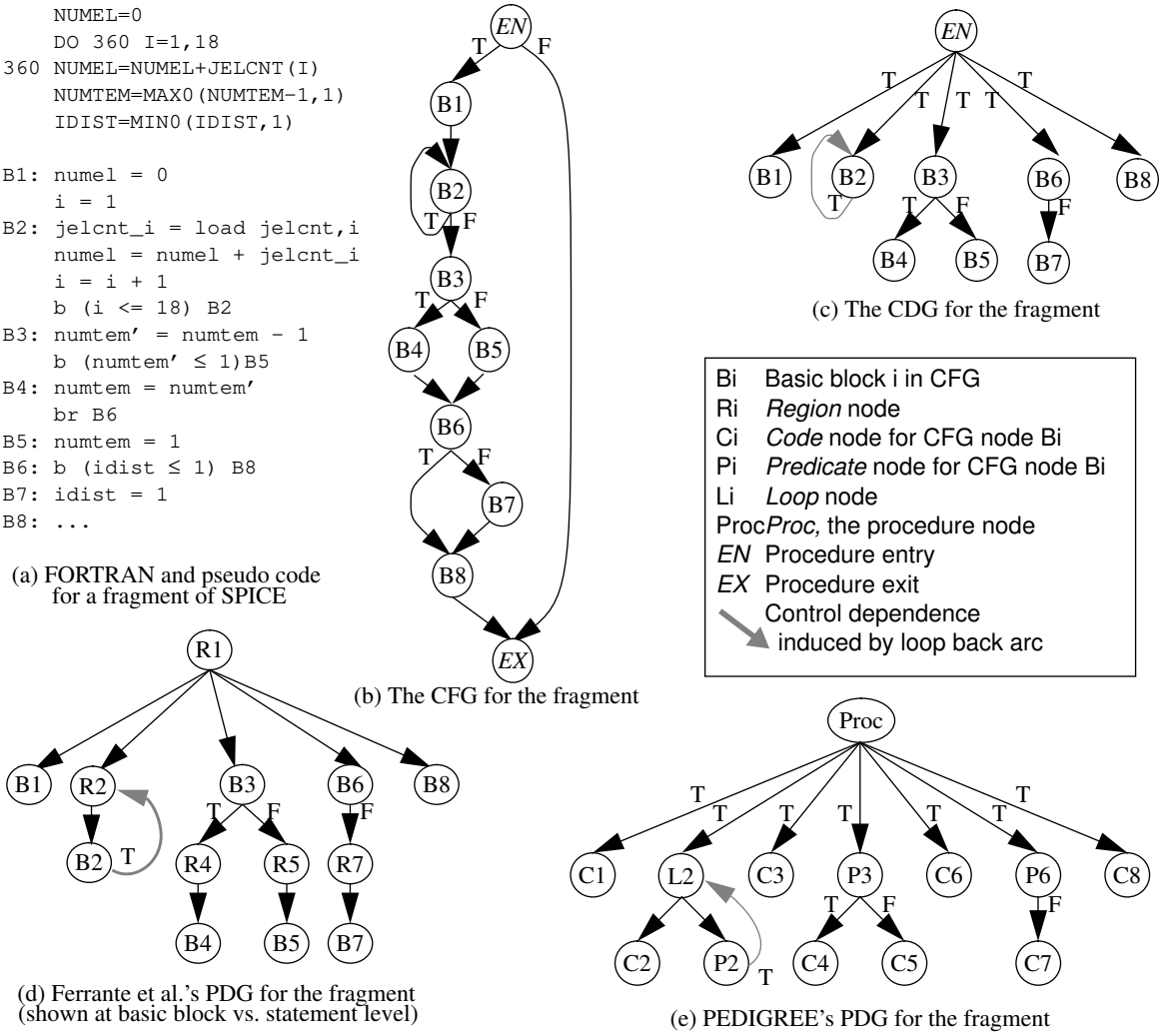


Figure 1. (a) FORTRAN example from spice, and its (b) CFG, (c) CDG, (d) PDG as specified in [6], and (e) our PDG

2.2. PDG Extensions

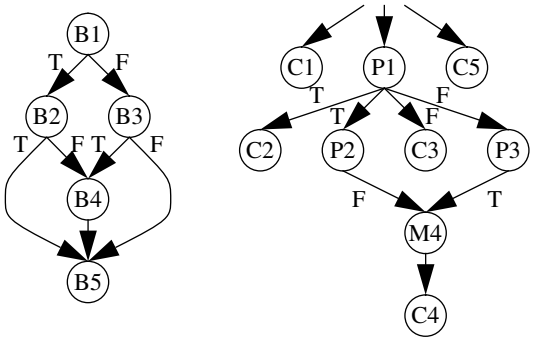
This work extends the PDG to better support exploitation of multi-grained parallelism. First, the *Region* node type described above is subclassified. This is done to identify and isolate PDG phenomena like cycles, regions with multiple control dependences, and irreducible loops, so that they can be handled more efficiently. Second, new node types are added to represent procedure calls and procedure entry points. Finally, each PDG node is labeled to indicate its position within the PDG. These labels facilitate efficient PDG traversal and computation of inter-node relationships such as dominance, reachability and control dependence. The first two extensions are presented in the following subsections; node labeling is presented in Section 3. Figure 1e exemplifies this type of PDG.

2.3. Subclassification of Region Nodes

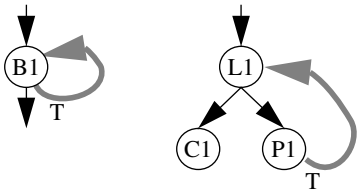
The generalized *Region* node type is subclassified to represent certain control dependence structures which should be handled differently by the compiler. The special treatment of the region associated with one of these special *Region* nodes is indicated by the region type.

Multiple-predecessor (*MultiPred*) nodes group sets of nodes that share multiple immediate control dependences. *MultiPred* nodes can arise from unstructured control flow like `goto` and `break` statements. A *MultiPred* node signals the compiler that special actions, such as code replication or liveness analysis, must be taken when moving

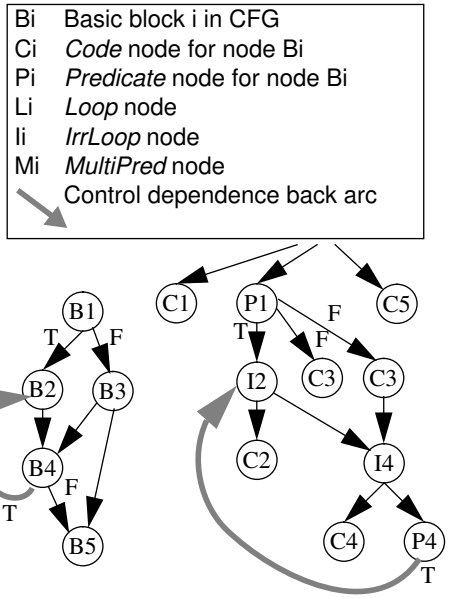
code into or out of its region. Note that, in general, *MultiPred* nodes (other than *Loop* and *IrrLoop* nodes – see below) can be eliminated by node splitting [6]. In Figure 2a, *MultiPred* node M4 is necessary because C4 is control dependent on two different *Predicate* nodes.



(a) CFG and PDG to illustrate the *MutliPred* node



(b) CFG and PDG to illustrate the *Loop* node



(c) CFG and PDG to illustrate the *IrrLoop* node

Figure 2. New *Region* node types: (a) *MultiPred* node; (b) *Loop* node; (c) *IrrLoop* nodes.

A *Loop* node is a special kind of *MultiPred* node which groups together all nodes in the body of a loop. *Loop* nodes are defined more generally than in [8]. The back arc which induces a loop [1] also induces a set of backward control dependences. These are represented by a single control dependence back arc, whose head is the *Loop* node and whose tail is the last *Predicate* that determines if the back arc is taken. A *Loop* node will have at least two incoming arcs: an entry condition, and a control dependence back arc, so it must be a *MultiPred* node. *Loop* nodes are used to identify where cycles occur in the PDG and where loop optimization techniques should be applied. Figure 2b shows a simple loop and its corresponding *Loop* node.

Irreducible loops [1] are represented similarly by a set of *IrrLoop* nodes. Such loops have an *IrrLoop* node for each entry point, since each entry node has different control dependences on nodes outside the loop. Irreducible loops have a single control dependence back arc. It emanates from the last *Predicate* node that determines whether the back arc is taken, and it is incident on the *IrrLoop* node that corresponds to the CFG node with the lowest depth-first number. The PDG in Figure 2c contains two *IrrLoop* nodes, I2 and I4, because the loop has two entry conditions. *Loop* and *IrrLoop* nodes identify and isolate all cycles in the PDG. This simplifies PDG traversal and processing. In addition to *MultiPred, Loop,* and *IrrLoop* nodes, other *Region* node types can be defined. For example, *Par* nodes can be used to partition nodes for parallel execution on different processors.

Hierarchical summary information is maintained in every PDG node. In the current implementation, this information includes execution frequency [13], register and memory variable usage [2], liveness and reaching definitions [1], and estimated parallelism of the region [2, 8]. Summary information facilitates efficient code transformation by summarizing all of the dependence analysis information needed for that region. A PDG node may be inserted specifically to group and summarize a code fragment for analysis or code motion.

2.4. Additional Node Types

A single *Proc* node represents the entry point(s) of a procedure, and there is one *Proc* node in each PDG. The body of a procedure can be considered control dependent on the procedure entry, so *Proc* nodes are a special type of *Predicate* node. *Call* nodes are leaf nodes which represent procedure calls and call instructions.

If a PDG is constructed for every procedure in a program, then a *Call* node can contain a pointer to the called procedure's PDG. An implicit call graph of the program is then present, with the PDGs as nodes, and the procedure pointers within *Call* nodes as arcs. Summary information for a procedure can be maintained in the *Proc* or *Call* nodes to support inter-procedural analysis and demand-driven inlining.

With this PDG design, a code fragment of varying size and granularity, including a mixture of control-equivalent regions of different granularities, can be summarized by a *Predicate* or *Region* node and treated as a single unit. This enables these fragments to be moved past other such fragments in what can be called multi-grained code motion. Identification of different types of regions indicates how each region should be treated. The next section describes a node labeling scheme that facilitates generalized PDG traversal for multi-grained code motion.

3. Node Labeling and PDG Construction

A node labeling scheme is proposed which labels each node in a PDG, as well as in a CFG, so that its position relative to any other node may be determined by comparing their labels. This facilitates computation of reachability, dominance, and control dependence relationships, as well as node-to-node traversals. Without such a labeling scheme, these operations would require expensive traversals from the root to each node. The labeling scheme presented here handles unstructured and irreducible code. In addition, it allows inserted nodes to be labeled without global changes to other node labels. Finally, a labeled PDG can be constructed during CFG labeling with minimal additional computation. A brief description of the labeling scheme follows.

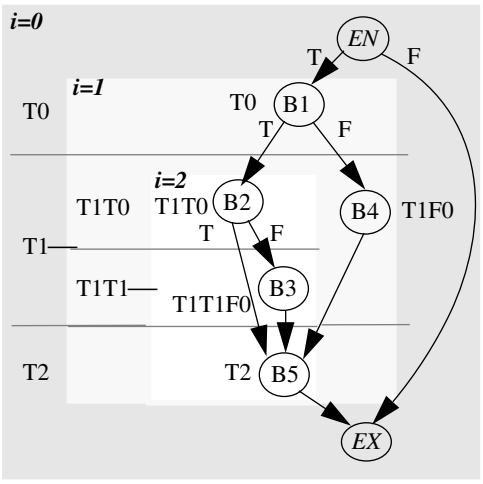
3.1. Program Control Dependence Structure

The structure of a program is determined by its patterns of control flow. There are three important control flow patterns: splits, joins, and back arcs. A *split* is a CFG node with multiple successors because of a conditional or multiway branch. It has one successor for each branch target, or "side", of the conditional. For example, an if-then-else construct has a split with two successors, one for the "then side" and one for the "else side", and a case statement has a split with one side for each case. *Joins* are CFG nodes with multiple predecessors. These occur when two or more paths rejoin. Each split has a corresponding *structured join*, which is the first CFG node where all paths from that split rejoin. In structured code, all joins are structured joins. In Figure 3b, the splits are *EN*, B6, and B7. The structured joins for these splits are *EX* for *EN,* and B11 for both B6 and B7. B10 is not a structured join. A back arc is a control-flow arc such that the tail of the arc has a higher depth-first number [1] than the head.

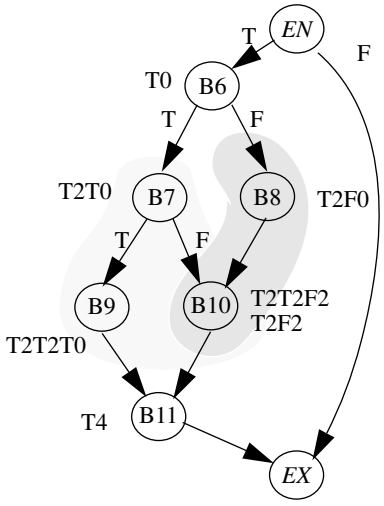
Nodes must be labeled such that a comparison of the labels indicates reachability, dominance, and control dependence. Node Y is *reachable* from node X iff they are on the same side of all enclosing conditionals and Y is on some path from X to *EX* that does not include back arcs. X *dominates* Y iff all paths to Y from *EN* pass through X. Y is *control dependent* on X if X is a split, and Y is inside that split's conditional. In order to evaluate these relations, the following information must be maintained at each node: 1) the conditionals it is nested within, 2) which sides of each enclosing conditional the node is on, and 3) relative position on each of those sides. Conditionals may be nested, producing the different conditional nesting levels illustrated in Figure 3a. The *EN* and *EX* nodes are at level 0. A node at conditional nesting level *i* is nested within *i* conditionals.

3.2. Node Labeling

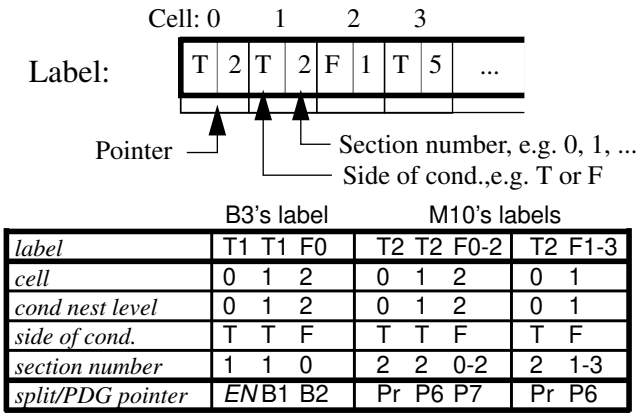
A label consists of an array of cells, where cell *i* describes the node's position within the enclosing conditional whose split is at nesting level *i*. Each cell has two fields. The first specifies which side of that conditional the node is on, e.g. with a truth value of T or F. The second is the section number, which indicates the relative position on that side of the conditional. It can take on values like 0,1,2... . When inserting a new node between two existing nodes, non-integer values can be used for section numbers to avoid having to modify section numbers of any existing nodes. Section numbers increase in the forward direction. Each cell also has a pointer to the CFG split or immediate PDG ancestor in the innermost enclosing conditional. This pointer aids in node labeling and PDG traversal of unstructured code. Figure 3c shows the elements of a label, and shows the correspondence of labels' cells to nodes with three examples.



(a) CFG example, showing conditional levels i = 0, 1 and 2 and increasing section numbers within each level



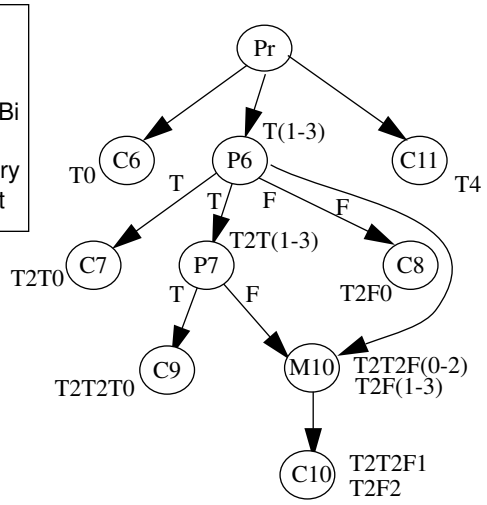
(b) CFG example, showing overlapping sides of a conditional, with B10 on both sides



	B3's label	M10's labels	
label	T1 T1 F0	T2 T2 F0-2	T2 F1-3
cell	0 1 2	0 1 2	0 1
cond nest level	0 1 2	0 1 2	0 1
side of cond.	T T F	T T F	T F
section number	1 1 0	2 2 0-2	2 1-3
split/PDG pointer	*EN* B1 B2	Pr P6 P7	Pr P6

(c) Elements of CFG and PDG labels

Pr	*Proc* node
Bi	Basic block i
Ci	*Code* node for Bi
Pi	*Predicate* node for Bi
Mi	*MultiPred* node
EN	CFG procedure entry
EX	CFG procedure exit



(d) PDG for (b)

Figure 3. Labeling examples. (a) CFG for structured code, (b) CFG for unstructured code, (c) Elements of CFG and PDG labels, (d) PDG for the CFG in (b).

Figure 3a illustrates conditional nesting levels. Split *EN* is at level 0, B1 is at level 1, and B2 is at level 2. Figure 3c illustrates the makeup of the label for B3, T1T1F0. It has three cells since it has three enclosing conditionals. Cell 0, T1, describes B3's position within the outermost conditional: it is on the true side and it is in section 1. B1 is in section 0 and B5 is in section 2 at nesting level 0. Cell 1, T1, describes B3's position within the enclosing conditional whose split is at level 1, B1. It is on the true side, and in section 1. Because B3 shares a common prefix, T1—, with B2, and it has a larger section number than B2 in cell 1, it must be reachable from B2. Finally, it is the first node in the innermost conditional nesting level, and thus has a section number of 0 in the last cell. Cell 1 of B4's label, F0, indicates that it is on the false side of the enclosing conditional at level 1.

Standard programming constructs such as if-then-else, multi-way branches and loops form structured code. Unstructured code allows control to flow from one side of a conditional to another, such that some nodes in that conditional are on more than one side of the conditional. A label can only describe position with respect to one side of a conditional, so such nodes are assigned one label for each side of a conditional that they are on. In Figure 3b, a conditional is formed by the split node B6 and structured join B11, which are at level 1. The side of this conditional associated with the T arc from B6 has nodes B7, B9 and B10. The other side of the conditional consists of

nodes B8 and B10. B10 is on both sides of the conditional because arcs B7-B10 and B8-B10 bridge between sides of the conditional. Thus B10 has two labels, one for each side of B6, with opposite truth values in cell 1.

Labels of *Predicate* and *Region* nodes have ranges for their section numbers in the last cell, *l*, instead of single values. The range bounds the possible section number values for cell *l* in the labels of the descendants of that node. Hence the labels of the root of a region can be inspected during traversal to determine if a particular labeled node is in that region. For example, in Figure 3d, all of P7's descendants have a label in the range (T2T1, T2T3), abbreviated T2T(1-3). The section number range is distinct from the multiple labels associated with *MultiPred* nodes. For example, M10 has two labels, T2T2F(0-2) and T2F(1-3).

3.3. PDG Construction

CFG node labels are assigned in depth-first order [1]: each node is labeled prior to its successors. The label(s) for a node are constructed from its predecessors' labels. Labels are essentially a list of conditionals whose splits have not been rejoined. Labels have a new cell appended for every conditional their node is nested within. At a structured join, labels from predecessors on all sides of a conditional are merged into a single, shorter label.

A PDG and its node labels may be constructed as the CFG is labeled. After a CFG node is labeled, a PDG *Code* node is created, along with a *Predicate* or *Call* node if the CFG node ends with a conditional branch or call instruction, respectively. A *Loop* or *IrrLoop* node is generated for each loop entry, and *MultiPred* nodes are generated for nodes with multiple control dependences. *Code* nodes are assigned the labels of the corresponding CFG nodes.

The immediate control dependence of CFG node X_C on the split node S_C of its innermost enclosing conditional is indicated by the last cell of X_C's label. A control dependence arc is added in the PDG from the *Predicate* node S_P that corresponds to S_C to the PDG node X_P that corresponds to X_C. The last cell of X_P's label points to S_P. In Figure 3, B6 has corresponding *Code* and *Predicate* nodes C6 and P6. C6 has the same label as B6, while P6's labels bound the labels of C7-C10. Since C10 has multiple immediate control dependences, a *MultiPred* node M10 is created for it. As seen in Figure 3c, both of M10's labels have their pointers set to P6 in cell 1. The section number ranges for *Predicate* and *Region* node labels are assigned after all of their children have been constructed.

3.4. Uses of Node Labels

With this node labeling scheme, evaluation of reachability, dominance, and control dependence are quite efficient. Let A[*i*] be the i^{th} cell of label A, and let *d* be the first cell index for which labels A and B differ. A $\prec$ B if A[*d*] and B[*d*] have the same truth value, and A[*d*] has a lower section number. For example, T2T1 $\prec$ T2T2F1 but T2T0 $\not\prec$ T2F2. Node B is reachable from A iff there exists a label of A, A_i, and a label of B, B_j, such that $A_i \prec B_j$. In Figure 3a, B2 is reachable from B1 since T0 $\prec$ T1T0. B2 (T1T0) is not reachable from B4 (T1F0) because their labels have different truth values in the first differing cell. Node A dominates node B iff for all of B's labels, B_j, there exists a label of A, A_i, such that $A_i \prec B_j$ and the first cell in which A_i and B_j differ is the last cell of A_i. The definition of postdominance is similar. In Figure 3b, B7 does not dominate B8 because they are on different sides of a conditional. B6 dominates B10, but B7 does not dominate B10. B10 postdominates B8. Control dependence is indicated with a pointer in cell *i* to the split at conditional nesting level *i*, on which the node is control dependent.

Labels uniquely describe a location in the PDG that is a target for traversal, since no two nodes have the same labels. PDG traversal from node *N* toward a target label *T* proceeds one PDG node at a time. The direction from *N* — right, left, up or down — is determined by comparing *T*'s label(s) with the label(s) of *N* and its neighbors. If the next neighbor in the direction of traversal is past the target, traversal stops. Unstructured code creates multiple paths in the PDG, between a *MultiPred* and the first *Predicate* that dominates the code in the *MultiPred* region. Non-overlapping traversal along multiple paths is very difficult without guidance from a labeling scheme. Consider traversal from M10 in Figure 3d to C6 via P6, along the two paths. P6 is the first node common to all paths up from M10, as determined by the pointers to P6 in M10's labels, shown in Figure 3c. Non-overlapping traversal from M10 along one path proceeds only up to P6 without visiting it, while the second should then proceed through P6 to C6. The target for traversal along the first path should be inside P6's region but before all of its children, such as T1.5, since T1 $\prec$ T1.5 $\prec$ T2—. The target label for the other path toward C6 remains T0. T0 is before T1,

so traversal continues to the left of P6. Such traversal is important for code motion, since it allows multiple copies of code moved out of a *MultiPred* region to be unified at the dominating *Predicate.* Section 5 addresses the complexity of label comparison and PDG construction. It uses collected data to illustrate their efficiency.

4. PEDIGREE: A PDG-Based Framework

The implementation of a post-pass, PDG-based, retargetable compilation environment called PEDIGREE is in progress. PEDIGREE is implemented in C++, following an object-oriented design philosophy. The unique treatment of each kind of PDG node, for summary, traversal, and scheduling, is encoded into the members and member functions of different classes. The software architecture has four layers. Lower layers generally provide services to higher layers. At the bottom, the *basic layer* handles fundamental data structures such as graphs. The *representation layer* deals with the CFG and PDG program representations. This includes constructing the CFG and PDG, and providing generic traversal functions. The *scheduling layer* provides low-level transformations related to code motion, including functions to perform data dependence analysis and memory disambiguation, as well as functions to actually move regions in the PDG. At the top, the *guidance layer* makes decisions regarding scheduling heuristics. This includes identifying parallelism-lean regions and attempting to fill them with code from parallelism-rich regions. The first two layers have been completed. Implementation of the third is nearing completion.

Currently, PEDIGREE performs the following functions. It parses assembly files generated by a high-level language compiler for an instruction set architecture specified in the architecture description file. It constructs a CFG for each procedure, constructs a labeled PDG, analyzes the PDG, and generates statistics. It also regenerates the CFG from the PDG in order to verify correct PDG construction. PEDIGREE has been used to collect the data for the SPEC92 benchmarks. It is currently targeted to the DEC Alpha 21064, but can be easily retargeted.

5. PDG Analysis

The current implementation of PEDIGREE has been used to analyze several benchmarks. A subset of ten benchmarks, including nine from the SPEC92 suite, are selected for presentation here. This section presents initial results for these benchmarks in two areas: general PDG statistics and PDG construction and labeling complexity.

5.1. General PDG Statistics

Table 1 lists the benchmarks used in this paper, and presents the distribution of the different types of PDG nodes. The presence of *MultiPred* nodes indicates that compilers must be able to handle the unstructured code in these benchmarks. The number of *MultiPred* nodes may be unexpectedly high for some of these benchmarks, but un-

Table 1: PDG Statistics

source	Benchmark	lines of assembly	# procedures	# PDG nodes	# PDG arcs	# *Code* nodes	# *Predicates*	# *Call* nodes	# *MultiPreds*	# *Loop* nodes	# *IrrLoops*	Multi-way br, (avg # targets)
SPECfp92	ear	16902	113	2812	2849	1511	504	578	38	106	0	2 (18)
	nasa7	10095	18	985	1100	512	229	93	0	133	0	0 (-)
	spice*	115013	111	13615	14734	7498	2817	2614	343	555	20	21 (6.2)
	tomcatv	1815	1	170	189	95	34	20	0	20	0	0 (-)
SPECint92	compress	4302	16	762	791	435	165	130	17	16	0	1 (10)
	eqntott	14319	62	2527	2691	1396	577	367	61	125	0	12 (7.5)
	espresso	74277	361	15246	15874	8314	3170	2690	197	711	0	6 (12.8)
	li	31868	357	6141	6132	3274	1118	1270	149	122	0	10 (7.6)
	sc	44198	151	7544	7892	4157	1554	1460	161	222	0	22 (18.3)
other	lloops	3142	2	318	367	160	53	52	0	51	0	0 (-)

* One spice procedure not processed at this time.

structured code can arise from `goto` and `break` statements, and from returns from within a procedure. Many of the programs are call-intensive, suggesting that inlining and inter-procedural analysis could provide considerable benefit. The conditional nesting depth can be larger than the nesting of `if` statements. The body of all loops that have an exit test at their entry point are control dependent on the exit *Predicate*. Multi-way branches are fairly common in integer code. The last column of Table 1 shows that the average number of targets ranges from 4 to 18. The number of PDG nodes, arcs, and labels are an indication of the memory requirements of the representation. The average number of PDG nodes per basic block is approximately 1.7.

5.2. Node Labeling Statistics

A node's label length is equal to its conditional nesting depth. Table 2 shows that it is largest for benchmarks with unstructured and irregular code, like spice, sc, and li. The cases with the largest average and maximum number of labels arise in parsers with switch statements whose cases have returns in them. Because the returns create a path from the multi-way branch to the exit, all the paths from that split are not rejoined until the procedure exit node.

The time for CFG labeling, PDG construction, and label comparisons depends primarily on the length and number of labels at each node. The complexity of label comparisons is $\mathbf{O(CU^2)}$, where **C** is the conditional nesting depth, and **U** is the number of labels per node. Since the average and maximum values of these parameters shown in Table 2 are small, label comparisons are generally efficient. Table 2 also shows labeling and PDG construction time, in seconds, on a 133 MHz Alpha 3000/400. This time is most affected by the number of PDG nodes, the number of labels per node, conditional nesting depth, and loop nesting depth. The implementation has yet to be optimized.

Table 2: Control Structure and Labeling Statistics, Ordered by Number of PDG Nodes

Benchmark name	Number of basic blocks	Avg. instr. per basic block	Number of PDG nodes	Avg. cond. nesting depth	Max cond. nesting depth	Avg # of labels per node	Max. # of labels per node	Avg. loop nesting depth	Max. loop nesting depth	Labeling, PDG construction (s)
tomcatv	97	7.78	170	1.92	4	1.00	1	1.52	3	0.91
lloops	164	9.99	318	1.01	2	1.00	1	0.69	3	2.37
compress	467	4.00	762	8.61	19	2.57	48	0.52	2	3.98
nasa7	548	9.13	985	2.01	5	1.00	1	1.44	4	4.09
ear	1737	4.36	2812	2.74	18	1.14	16	0.36	3	6.27
eqntott	1765	4.11	2933	6.40	16	2.53	84	0.81	7	17.11
li	3493	3.16	5323	8.23	21	2.74	1024	0.25	2	21.61
sc	4459	3.70	7544	10.03	24	3.86	81	0.75	7	229.29
spice	6850	7.11	12237	8.07	23	3.56	178	1.18	7	3500.91
espresso	8331	4.03	14008	3.85	15	1.29	49	0.58	4	197.22

6. Scheduling Scopes

Several researchers [2,3,4,5,6,8,11,13] have proposed the PDG as a better representation than the CFG for use in code optimization. The size of the scope for code scheduling is a key factor in code optimization. The results in this section highlight the available scopes for code motion and scheduling using the PDG representation, and show how they compare with available scopes using the CFG representation. Since the data dependence analysis of PEDIGREE is not yet complete, data dependences and specific code motion techniques are not addressed here.

6.1. Scheduling Scope Without Speculation

A PDG explicitly represents all control dependences. *Region* node are used to identify a set of control-equivalent basic blocks. All control-equivalent basic blocks can be grouped into one scope for fine-grained scheduling (subject to data dependences). These basic blocks may be separated by potentially large distances in the CFG, making it difficult for CFG-based techniques to harvest such parallelism. Table 3 presents the average sizes of these con-

Table 3: Scope Size Without Speculation

Benchmarks		ear	nasa7	spice	tomcatv	compress	eqntott	espresso	li	sc	lloops	WHM
PDG	B.B.	4.8	9.0	7.3	22.8	2.6	3.1	5.2	4.2	3.2	90.8	5.1
	Instr	24.3	87.8	57.1	242.9	10.3	14.4	25.2	18.5	12.2	1038.6	23.9
CFG	B.B.	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	Instr.	5.4	9.8	8.4	8.0	4.3	5.0	4.6	4.1	4.0	10.2	5.6

trol-equivalent PDG scopes and non-speculative CFG scopes. It is assumed that code is not moved to blocks with lower execution frequency. For example, moving code from outside a conditional onto all paths of the conditional requires replication and results in code explosion. This assumption limits PDG scope sizes more than CFG scope sizes. The weighted harmonic mean of the number of control-equivalent basic blocks, across all benchmarks, is 5.1, and the corresponding number of instructions is 23.9. For comparison, the analogous average scope sizes that can be obtained using CFGs without speculation are 1.0 basic blocks and 5.6 instructions. The ability to identify control-equivalent basic blocks enables PDGs to increase the average scope size for fine-grained scheduling to over five times that of the CFGs. Potentially, this ability could significantly increase instruction-level parallelism and resultant performance. Actual increases will depend on data dependences.

The difference in scope sizes is primarily due to the intrinsic weakness of the CFG: not explicitly representing control dependence information. The control equivalence of basic blocks before and after a conditional (or called procedure) cannot be determined, in general, with traversal along a single path. Consequently, code motion techniques that traverse CFG paths would not be able to determine this equivalence and would need to conservatively assume that these two basic blocks are control dependent. Use of the CFG effectively induces "false" control dependences. The scope for fine-grained code motion is restricted by these false dependences. Control-equivalent basic blocks that are dispersed throughout the program can be easily identified using the PDG.

6.2. Scheduling Scope With Speculation

Speculative scheduling involves moving instructions across basic block boundaries, or to be more precise, moving instructions so that they are executed prior to the resolution of their control dependence conditions. To perform speculative code motion, a speculation scope is usually identified by grouping multiple basic blocks that are adjacent in the CFG. The degree of speculation is typically measured in terms of either the number of basic block boundaries crossed or the number of conditional branches crossed. The opportunity for code motion generally increases with a larger scope, which can be obtained with higher degrees of speculation.

Figure 5 presents the PDG and CFG speculation scope sizes for various degrees of speculation (*DOS)*. Figure 5 presents corresponding scope sizes when speculation across loop back arcs is considered. Given a DOS of *N*, the

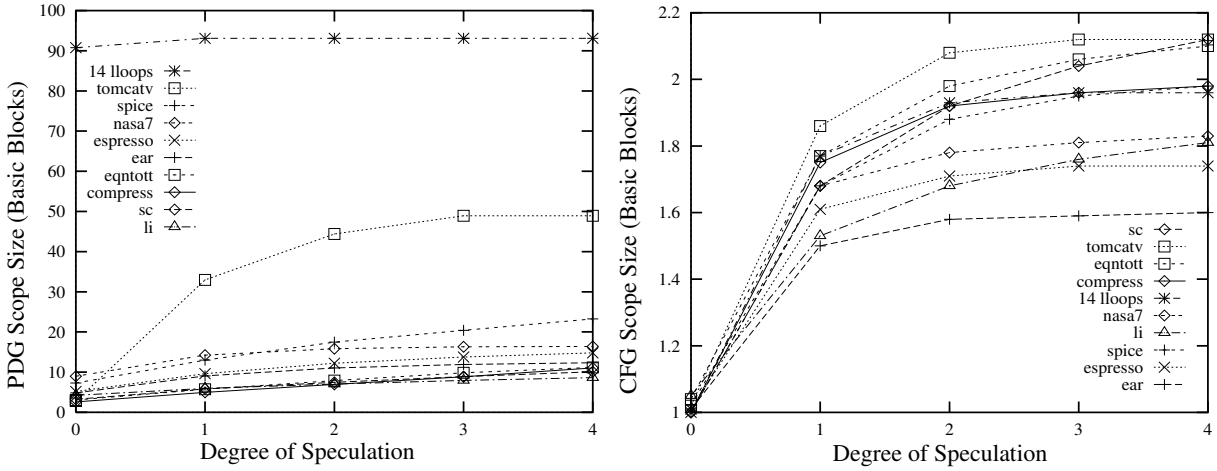


Figure 4. Scope size as a function of degree of speculation, for the PDG (a) and the CFG (b)

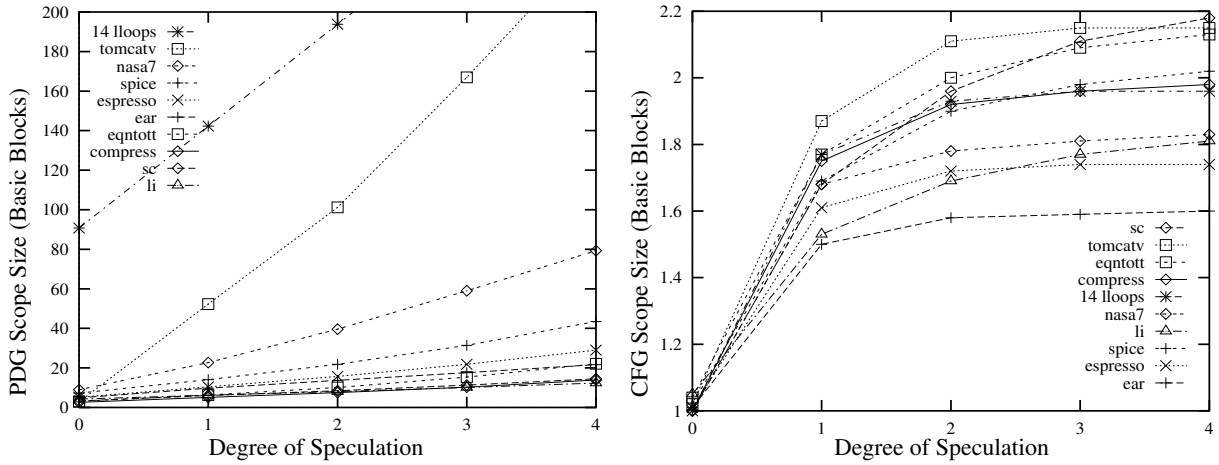


Figure 5. Scope size as a function of degree of speculation, considering speculation across loop back arcs, for the PDG (a) and the CFG (b).

CFG speculation scope size indicates the average number of basic blocks that are reachable from a reference basic block by traversing up to *N* divergences of control flow, i.e. *N* splits in the forward direction or *N* joins in the backward direction. CFG scopes are not permitted to extend past procedure calls. The PDG speculation scope is determined by crossing up to *N* true control dependences. Only control dependences are considered, not the original control flow order. Thus intervening *Call* nodes are ignored. Inlining should further increase PDG scope sizes.

The average PDG scope size in basic blocks across all benchmarks is 9.0 for DOS 1, and 14.7 for DOS 4. When speculation across loop back arcs is permitted, this number increases to 25.6 for DOS 4. The average scope size in instructions is 37.0 for DOS 1 and 57.7 for DOS 4. The corresponding CFG scope sizes are 1.7 (DOS 1) and 1.9 (DOS 4) basic blocks, and 8.4 (DOS 1) and 9.4 (DOS 4) instructions. PDG speculation scope size is 5.3 times greater on average than the CFG speculation scope size for DOS 1, and 7.7 times greater for DOS 4. Clearly, for speculative scheduling with a given degree of speculation, the PDG provides larger scopes for scheduling than the CFG. Furthermore, certain speculations in the CFG are not truly speculative, due to the false control dependences induced by the traversal of paths in a CFG. Consequently, some basic blocks in the speculation scope of degree *N* are not necessarily speculative of that degree.

Space does not permit presentation of more detailed data for individual code modules, but some insights are provided here. The bigger the procedure, the larger the PDG scopes, and the greater the benefits of the PDG vs. the CFG. For very small procedures without procedure calls, the PDG and CFG scope sizes are similar, although CFG scopes rise more slowly with DOS. The increase in scope size in the PDG levels off quickly for very shallow PDGs (e.g. espresso's cubestr) and more slowly for deeper ones (espresso's mv_reduce). Calls are very limiting to CFG scopes. Table 2 indicates that ear, spice, li and lloops have a high ratio of *Call* nodes to *Code* nodes. This may explain why these benchmarks have smaller CFG scope sizes. As can be observed from a comparison of Figure 5 and Figure 5, the relative contribution of speculation across loop back arcs is small for low DOS. For higher DOS, when other speculation potential is exhausted, consecutive loop iterations offer a continuous supply of parallelism. Loop speculation accounts for only 18% of the scope size increase of 4.7 basic blocks from DOS 0 to DOS 1. But it accounts for 77% of the scope size increase of 6.2 basic blocks from DOS 3 to DOS 4.

Two key differences between the CFG and the PDG are that the PDG represents the control dependences instead of the control flow of the CFG, and that the PDG has the capability for hierarchical representation using region nodes. The former removes the undesirable artifacts of the original sequential code. The latter allows encapsulation of information and more efficient traversal of the PDG during code motion. Initial results from the PEDIGREE tool appear to support the claims of the proponents of the PDG [2, 5, 6, 8, 13].

7. Summary and Potential PDG-Based Optimizations using PEDIGREE

The PDG described here is useful in exposing hierarchical control dependence structure. This information can be

used to determine the kind of transformations that should be applied to each section of the program, e.g. speculation, loop optimization, and replication. PDG analysis suggests the benefits of increased scope size for different degrees of speculation, and can provide guidance in selecting appropriate aggressiveness for each procedure. The use of accurate control dependence information to avoid unnecessary speculation provides a 5x increase in scope size. Aggressive scheduling techniques are needed to exploit parallelism for non-loop code on wide architectures. Exploiting even a small fraction of the additional parallelism discussed here is a significant step toward that goal.

The existing PDG-based tool, PEDIGREE, provides a framework for future development of a number of useful capabilities. It can be extended to take advantage of PDG features to 1) increase the scheduling scope and guide speculation with accurate control dependence information, 2) perform inlining and inter-procedural analysis, 3) move any region of code along an unstructured path, 4) schedule for multiple-instruction stream architectures, and 5) perform binary optimization. The data from Section 6 suggests that the PDG scopes for code motion for a given degree of speculation are significantly larger than those of a CFG. Fine-grained code scheduling techniques can be implemented in PEDIGREE to take advantage of the larger scope. The *Call* and *Proc* nodes added here facilitate inlining, which can further increase scope size. They can also be used to summarize parallelism, variable usage, and execution frequency, in order to inform a global guidance layer of the benefits of inlining, and to facilitate interprocedural analysis. The implementation of inlining and use of branch profiling data will be completed shortly.

The PDG's features provide a solid basis for partitioning and scheduling code for multiple-instruction stream architectures. Such features include its ability to represent parallel schedules, its summary of parallelism information that guides trade-offs in the redistribution of parallelism, and its support for generalized multi-grained code motion. Parallelism studies [10] indicate that exploiting control parallelism in multiple instruction streams can lead to a three-fold performance increase over a single instruction stream for non-loop-intensive code.

The post-pass PEDIGREE tool currently parses assembly files and constructs the PDG for a given architecture specified in an architecture description file. Using this feature, PEDIGREE can be easily retargeted for other assembly languages. There is potential for using this tool to perform translation and optimization on existing object code. This will permit the effective execution of a large volume of existing software on new platforms without recompilation from the source. Future exploration of this potential application using PEDIGREE is planned.

References

[1] A. Aho, R. Sethi and J. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[2] V. Allan, J. Janardhan, R. Lee and M. Srinivas, "Enhanced Region Scheduling on a Program Dependence Graph", In *Proc. MICRO 25*, 1992, 72-80.

[3] R. Ballance, A. Maccabe and K. Ottenstein, "The Program Dependence Web: A Representation Supporting Control-, Data, and Demand-Driven Interpretation of Imperative Languages", In *Proc. PLDI*, 1990, 257-271.

[4] W. Baxter and H. Bauer III, "The Program Dependence Graph in Vectorization", In *Proc. PoPL*, 1989, 1-11.

[5] D. Bernstein and M. Rodeh, "Global Instruction Scheduling for Superscalar Machines", In *Proc. PLDI,* 1991, 241-255.

[6] J. Ferrante, K. Ottenstein and J.Warren. "The Program Dependence Graph and Its Use in Optimization", *ACM TOPLAS* 9(3), July 1987, 319–349.

[7] M.Girkar and C. Polychronopoulos, "Automatic Extraction of Functional Parallelism from Ordinary Programs," IEEE Transactions on Parallel and Distributed Systems, 3(2), March 1992, 166-178.

[8] R.Gupta and M. L. Soffa. "Region Scheduling", In *Proc. Second Int'l Conf. on Supercomputing*, 1987, 141–8.

[9] S. Horwitz and T. Reps, "The Use of Program Dependence Graphs in Software Engineering", In *Proc. International Conference on Software Engineering,* Melbourne Australia, 1992, 392-411.

[10] M. Lam and R. Wilson, "Limits of Control Flow on Parallelism", In *Proc. ISCA*, 1992, 46-57.

[11] C. Norris and L. Pollock, "Register Allocation over the Program Dependence Graph," In *Proc. PLDI*, 1994.

[12] K.Ottenstein and S. Ellcey, "Experience Compiling FORTRAN to Program Dependence Graphs", Software — Practice and Experience, 22(1), 1992, 41-62.

[13] V. Sarkar, "Automatic Partitioning of a Program Dependence Graph into Parallel Tasks", IBM Journal of Research and Development, 35(5/6), 1991, 779-804.