International Journal of Parallel Programming, Vol. 22, No. 5, 1994

The Hierarchical Task Graph as a Universal Intermediate Representation[1]

Milind Girkar[2] and Constantine D. Polychronopoulos[3]

Received September 23, 1993

This paper presents an intermediate program representation called the Hierarchical Task Graph (HTG), and argues that it is not only suitable as the basis for program optimization and code generation, but it fully encapsulates program parallelism at all levels of granularity. As such, the HTG can be used as the basis for a variety of restructuring and optimization techniques, and hence as the target for front-end compilers as well as the input to source and code generators. Our implementation and testing of the HTG in the Parafrase-2 compiler has demonstrated its suitability and versatility as a potentially universal intermediate representation. In addition to encapsulating semantic information, data and control dependences, the HTG provides more information vital to efficient code generation and optimizations related to parallel code generation. In particular, we introduce the notion of precedence between nodes of the structure whose grain size can range from atomic operations to entire subprograms.

KEY WORDS: Intermediate program representation; Parafrase-2 compiler; hierarchial task graph; precedence.

1. INTRODUCTION

The intermediate program representation presented and analyzed in this paper was motivated by the requirements of Parafrase-2, a multilingual parallelizing compiler designed and developed at the University of Illinois.

[1] This work was supported in part by the National Science Foundation under Grant No. NSF-CCR-89-57310, the U. S. Department of Energy under Grant No. DOE-DE-FG02-85ER25001, and a grant from Texas Instruments Inc.

[2] Sun Microsystems, Inc. Mountain View, California 94043.

[3] Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign-Urbana, Illinois 61801.

0885-7458/94/1000-0519$07.00/0 © 1994 Plenum Publishing Corporation

The multilingual aspect of the compiler gave rise to the question of a Universal Intermediate Representation (UIR) powerful enough to serve as the IR for languages as diverse as Fortran and C.

In this paper, we present the result of our research on the design and analysis of Parafrase-2's intermediate representation, called the Hierarchical Task Graph. Up until the late 80's, the majority of commercial and experimental compilers used a combination of program structures—such as the abstract syntax tree (AST) and the control flow graph (CFG)—as the intermediate representation of source code, on which code optimization and generation could be performed.

The proliferation of parallel programming introduced new challenges for compiler writers regarding optimizations specific to parallelization, and parallel code generation. For a number of years these challenges were dealt with the introduction of new or derived program representations, such as the data dependence graph. Our work aims at the design of a single intermediate program representation which encapsulates all information necessary to carry out traditional optimizations, parallelization, and code generation. Thus far, our results suggest that the HTG can serve in this capacity as a powerful intermediate representation for languages such as Fortran and its many dialects and C. In Ref. 1, preliminary results suggest that the HTG can be suitable for functional languages as well.

In this paper, we give the formal definition of the HTG, discuss its properties, present the details of its derivation, and demonstrate its use as an intermediate representation which explicitly specifies program parallelism at all granularity levels. In the context of Parafrase-2, the HTG has already been used as the basis for restructuring (parallel source code generation),[2] for machine code generation,[3] and for studying how parallelism at different granularity levels can be exploited.[4] We claim that the properties of the HTG make it a suitable target for conventional languages as well as for languages with explicit parallel syntax.

The organization of the paper is as follows: In Section 1.1 we review related work and juxtapose the HTG to other similar IRs. Section 2 gives the formal definition and derivation of the HTG, and discusses control flow normalization to facilitate the construction of the hierarchy of operations. Section 3 discusses the next step, namely the augmentation of the HTG with control and data dependence information. In particular, Section 3.5 presents the closure graph which is used to introduce the notion of precedence constraints among program operations. Section 4 discusses briefly implementation of the HTG in Parafrase-2, and Section 5 gives some concluding remarks.

1.1. Problem Definition and Previous Work

The control flow graph[5] of a program is used as the starting point in the process of detecting parallelism in sequential programs. There are four main problems encountered in extracting functional parallelism:

1. Factoring out the loops in the control flow graph so that the body of each loop is a directed acyclic graph of tasks;
2. Extracting parallelism from such an acyclic graph of tasks, through the notions of control and data dependence and synchronization of the dependences between tasks;
3. Eliminating redundant synchronizations;
4. Parallel code generation.

Many traditional optimizations can be done more efficiently by partitioning the flow graph into intervals. This partitioning defines a hierarchical structure on the flow graph.[5] Our interest in a hierarchical structure is to identify each node in the hierarchy as a task that consists of subtasks. An interval in a flow graph consists of a natural loop plus an acyclic structure that hangs from the nodes of that loop.[5] This makes it unsuitable to be identified as a task. Instead, we follow the approach in Ref. 6, where the hierarchy was based on strongly connected regions, and derive the HTG. A graphical hierarchical representation for SISAL[7] programs was proposed in Ref. 8. However, this was not developed to detect parallelism in programs (indeed the parallelism was specified in the intermediate structure itself); instead it was used to study the partitioning and scheduling of parallel programs. As such, the problems that we will discuss related to control and data dependences are not studied in Ref. 8.

Dependences arising from the flow of control in the program were handled earlier by converting them into data dependences through *if conversion*[9] or through the use of *mode functions*.[10] The general notion of *control dependence* was first formalized in Ref. 11. This allowed the separate treatment of control and data dependences. Since then control dependences have been studied explicitly for various purposes. In Ref. 12, an algorithm for loop distribution in the presence of arbitrary control flow was presented. The *program dependence graph* proposed in Ref. 11 was used for vectorization in Ref. 13. In Refs. 14 and 15, control dependences were used to specify functional parallelism, referred to as DAG parallelism. Control dependence was also used to formulate parallel processes on the basis of intervals (as opposed to our approach of using strongly connected regions) in Ref. 16. Our work differs from Cytron *et al.*[14] primarily in the treatment of data dependences. We allow explicit synchronization based on execution conditions.[17]

Central to the process of eliminating redundant synchronizations is the notion of *precedence* between nodes; precedences define which nodes execute *before* other nodes. Precedence was used in Refs. 18 and 19, in the context of static race detection; in Section 3 we point out the important differences between our problem and the work done by Callahan and Subhlok.[18] An analogous relation of nodes that may not execute simultaneously was studied for static deadlock detection in Ada programs by Masticola and Ryder.[20] We show that in our case, precedence can be completely formalized in graph theoretic terms. This is significantly different from previous approaches, which have defined precedence in terms of execution of the program. Optimizing synchronization across loop iterations has been studied.[21–26] However, all of these assume an absence of control dependences or the prior conversion of control dependences into data dependences.

2. THE HIERARCHICAL TASK GRAPH

In this section we take the first step toward the creation of the hierarchical task graph (HTG) from the control flow graph. The importance of a hierarchical structure for traditional compiler optimizations has long been recognized.[5] Its importance in the detection and management of parallelism has been recognized more recently.[8] We will build a hierarchical structure based on strongly connected regions[6] and eventually identify each node in the hierarchical structure as a task in the HTG. The HTG is a layered graph, in which each layer is a control flow graph, not unlike the original control flow graph except that it will be acyclic; each such graph also has associated control and data dependence graphs.

The derivation of the control and data dependence graphs is discussed later; however, we will continue to refer to the constructed graph here as the HTG.

2.1. Obtaining a Hierarchy

The classical approach to obtaining a hierarchy from the control flow graph as defined in Ref. 5 follows interval analysis[27]; this was used by Cytron *et al.*[16] However this technique has two drawbacks for our purposes:

1. Intervals are not strongly connected, which means that if a loop were to belong to an interval, the interval could potentially also contain other nodes not in the loop.
2. Irreducible flow graphs require special techniques like *node splitting*[28] in order to construct the complete interval-based hierarchy.

To alleviate these limitations, an alternative approach of identifying the intermediate nodes in the hierarchy as strongly connected regions was first suggested by Tarjan,[29] and later developed by Schwartz and Sharir.[6] We follow this approach in building the task graph. We first give an intuitive understanding of the hierarchical task graph and then a more rigorous formulation. For irreducible graphs, where a loop can have many entry nodes, we arbitrarily decide on one such node as *the* entry node. This entry node is defined by the depth first search done on the graph. This also means that for irreducible graphs, a different depth first search ordering can give rise to a different HTG. Once the hierarchy is built, we ensure that graphs (at all levels) have a single entry and exit point by adding additional nodes and arcs as necessary.

The loop hierarchy is different from that in Ref. 16, where the hierarchical structure of a program is generated from intervals.[5] Figure 1 illustrates a simple example where the loop and interval hierarchies are different.

Figure 2b illustrates the *hierarchical task graph* of the program fragment in Fig. 2a. At the top level of the hierarchy the graph consists of four nodes, of which A and B are *loop* nodes and D is a *compound* node corresponding to the control flow graphs of higher level structures such as loops and subroutines. At the next hierarchy level node B consists of three nodes, one of which (C) corresponds to a loop structure. Thus the flow graph of Fig. 2b can be identified as a three-level hierarchical task graph; at the third and lowest hierarchy it consists of 16 tasks corresponding to basic blocks. Each node is a single-entry/single-exit task at its own hierarchy level.

We give a more formal description of the process of constructing the HTG in the next two sections, following the treatment developed by Schwartz and Sharir.[6]

2.2. Loops

A *control flow graph* is a directed graph $G = (V, E)$ with unique nodes $ENTRY$, $EXIT \in V$ such that there exists a path from $ENTRY$ to every node in V and a path from every node to $EXIT$; $ENTRY$ has no incoming arcs, and $EXIT$ has no outgoing arcs. We will allow paths to contain zero arcs, specifically mentioning non-null paths when they must have at least one arc. An example control flow graph (from Ref. 5) along with the $ENTRY$ and $EXIT$ nodes is shown in Fig. 3.

Loops in a CFG can be detected by a depth first search (DFS) traversal as follows. We first do a DFS on the control flow graph with

ENTRY as the initial node. There are four possibilities for an arc (x, y) encountered while doing a depth first search at node x.[28]

1. Node y is unvisited, (x, y) is a *tree* arc.
2. There is a path from y to x consisting of tree arcs. We will call such a path a *tree path* and say that y is an *ancestor* of x in the depth first search tree. If the path is non-null, then we also say that x is a *descendant* of y. In this case (x, y) is a *back* arc.

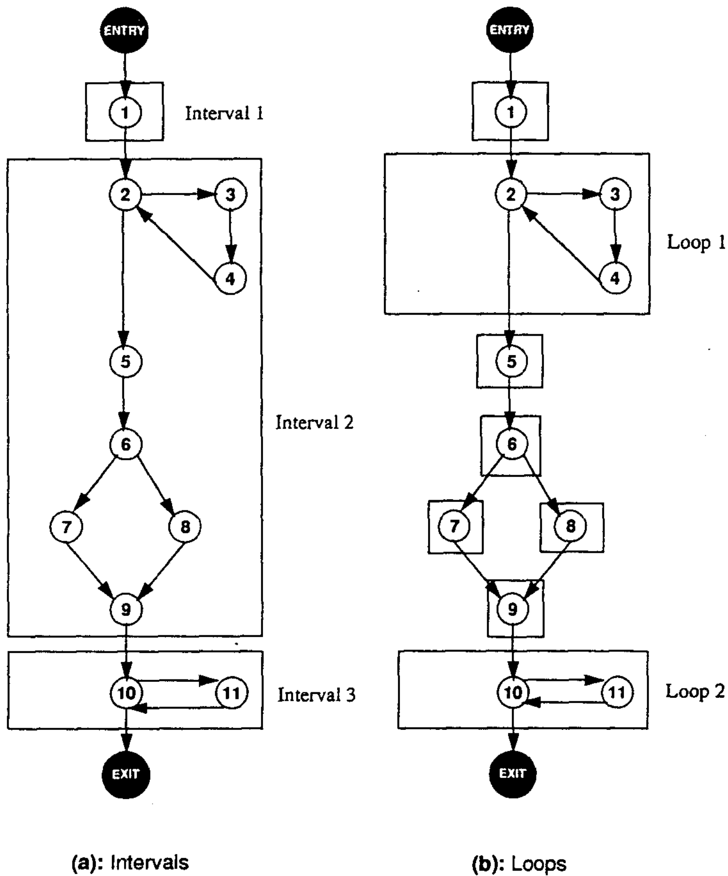

Fig. 1. Building hierarchies with intervals and loops.

3. Node y is a descendant of x in the depth first search tree; (x, y) is a *forward* arc.
4. Node y has been visited but is neither an ancestor nor a descendant of x; (x, y) is a *cross* arc.

A depth first search tree for our example graph of Fig. 3 is shown in Fig. 4. The set of back arcs is $\{(9, 1), (8, 3), (7, 4), (4. 3), (10, 7)\}$, $(1, 3)$ is a forward arc, $(6, 7)$ is a cross arc, and the rest are tree arcs. For an arc (x, y), x is the *source* and y is the *sink* of the arc. Let $H(G)$ be the set of all nodes that are sinks for back arcs.

$$H(G) = \{x: x \in V, \exists y \text{ such that } (y, x) \text{ is a back arc}\}.$$

For our example graph of Fig. 3, $H = \{1, 3, 4, 7\}$.

Node x *dominates* node y, denoted by $x \Delta_d y$, iff every path from *ENTRY* to y contains x.[5] A node always dominates itself. Let B be the set

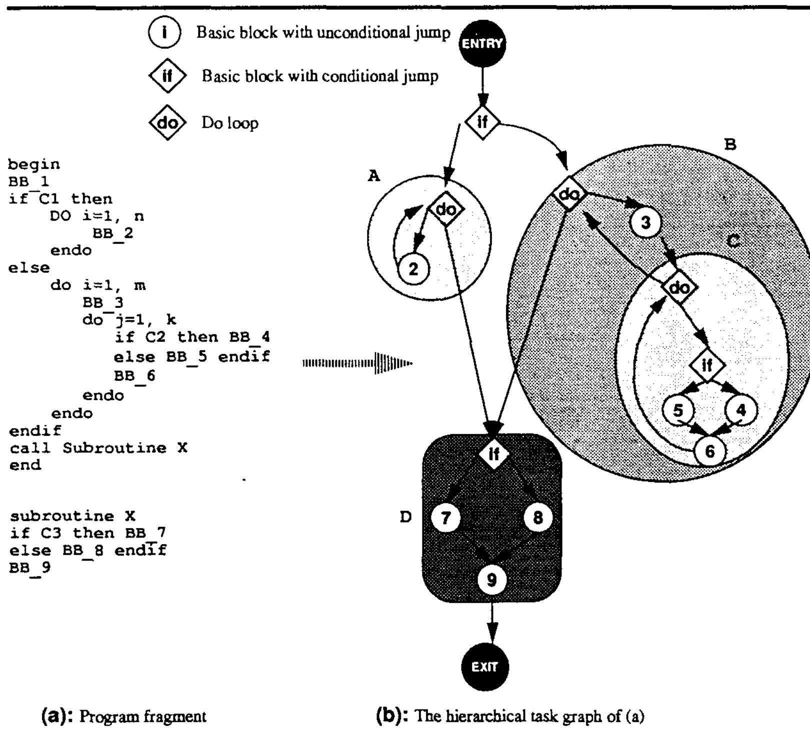

Fig. 2. Hierarchical task graph.

of arcs whose sinks dominate their sources. A graph $G = (V, E)$ is *reducible* if the graph $G = (V, E - B)$ is acyclic. Lemma 2. 1 states that for reducible graphs, B is equal to the set of back arcs.[5]

Lemma 2.1. For reducible graphs, e is a back arc iff $e \in B$.

Let $T(x)$ denote the descendants of x in the depth first search tree.

$$T(x) = \{y: y \in V, \exists \text{ a non-null path made up of tree arcs only from } x \text{ to } y\}.$$

Lemma 2.2. If $y \in T(x)$ then $T(y) \subset T(x)$.

Proof: Follows from the definition. □

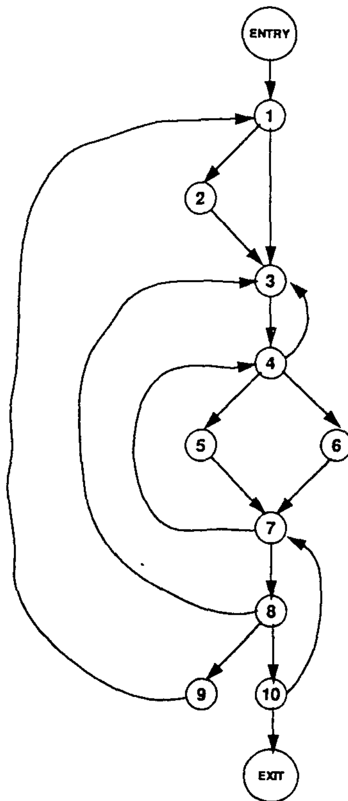

Fig. 3. A control flow graph.

The *loop* associated with back arc (x, y), $L(x, y)$, is defined as y plus the set of nodes n such that there is a path P in G from n to x, and if z is any node on P, then $z \in T(y)$. Node y is called the *header* of the loop. In the traditional definition of a loop[5] (for reducible graphs), $TL(x, y)$, a node n belongs to $TL(x, y)$ if it is either y or there is a path P from n to x which does not contain y. For irreducible graphs, the traditional definition will not work, because it is possible for a node (for example, a) outside the loop to have a path into the body of a loop which excludes the header; in such a case, according to the traditional definition, a would belong (incorrectly) to the loop. For reducible graphs, as Lemma 2.3 shows, the two definitions are identical.

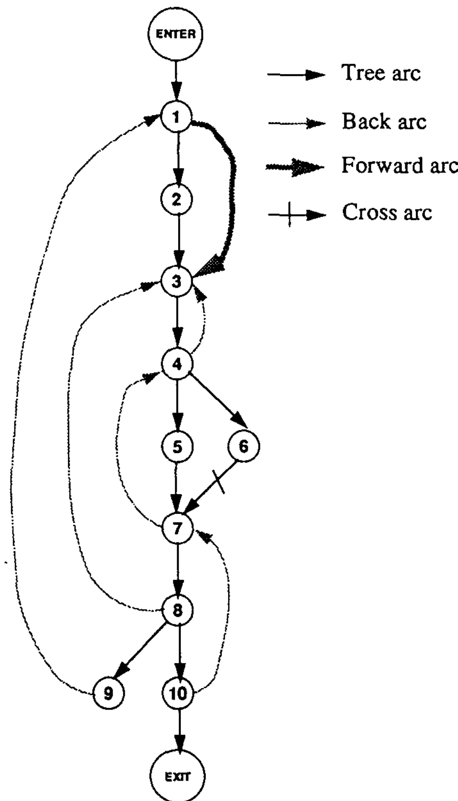

Fig. 4. Depth first search tree for Fig. 3.

Lemma 2.3. If G is reducible, then for any back arc (x, y), $L(x, y) = TL(x, y)$.

Proof: Let $n \in L(x, y)$. If $n = y$, then $n \in TL(x, y)$ by definition. Otherwise, there is a path P from n to x such that all nodes on P belong to $T(y)$. By definition, $y \notin T(y)$ and hence y cannot lie on P. This implies $n \in TL(x, y)$ and shows that $L(x, y) \subseteq TL(x, y)$.

Let $n \in TL(x, y)$. If $n = y$, then $n \in L(x, y)$ by definition. Otherwise there is a path P from n to x which does not contain y. Let z be any node on the path P. If G is reducible, then $y \Delta_d x$. Since there is a path from z to x that excludes y and $y \Delta_d x$, $y \Delta_d z$. This implies that in the depth first search tree, z must be a descendant of y and hence $z \in T(y)$. Thus, $n \in L(x, y)$, or $TL(x, y) \subseteq L(x, y)$. □

The loops for our example graph in Fig. 3 are given in Table I.

Lemma 2.4. Let $L(a, b)$ and $L(c, d)$ be two loops. If $b \in L(c, d)$ and $b \neq d$, then $L(a, b) \subset L(c, d)$.

Proof: Since $b \in L(c, d)$ and $b \neq d$ there is a path $P_1 = \langle b = b_0, b_1, b_2, ..., b_n = c \rangle$ such that $b_i \in T(d)$ for $0 \leqslant i \leqslant n$. In particular, $b \in T(d)$ and by Lemma 2.2, $T(b) \subset T(d)$.

Let $x \in L(a, b)$. If $x = b$, then by hypothesis, $b \in L(x, d)$. Otherwise, there is a path $P_2 = \langle x = a_0, a_1, a_2, ..., a_m = a \rangle$ such that $a_i \in T(b)$ for $0 \leqslant i \leqslant m$. Since $T(b) \subset T(d)$, $a_i \in T(d)$ for $0 \leqslant i \leqslant m$. Consider the path $P_3 = \langle x = a_0, a_1, a_2, ..., a_m = a, b = b_0, b_1, b_2, ..., b_n = c \rangle$ from x to c formed by composing P_2, the arc (a, b) and P_1. Every node on P_3 belongs to $T(d)$ and hence $x \in L(x, d)$. Thus, $L(a, b) \subseteq L(x, d)$. Note that there cannot be a tree path from b to d (as one exists from d to b) and hence $d \notin L(a, b)$. This proves $L(a, b) \subset L(c, d)$. □

Lemma 2.5. Let $b \neq d$ and $Z = (L(a, b) \cap L(c, d))$ be nonempty. Then Z is either $L(a, b)$ or $L(c, d)$.

Table I. Loops for Figure 3

Back arc	Loop
$L(9, 1)$	$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
$L(8, 3)$	$\{3, 4, 5, 6, 7, 8, 10\}$
$L(7, 4)$	$\{4, 5, 6, 7, 8, 10\}$
$L(10, 7)$	$\{7, 8, 10\}$
$L(4, 3)$	$\{3, 4, 5, 6, 7, 8, 10\}$

Proof: Let $x \in Z$. If $x = b$, then we can apply Lemma 2.4 ($b \neq d$, $b \in L(c, d)$), and obtain $L(a, b) \subset L(c, d)$, and hence Z is $L(a, b)$. Similarly, if $x = d$, Z is $L(c, d)$. Hence, let $x \neq b$ and $x \neq d$. Since $x \neq b$ and $x \in L(a, b)$, there exists a path, P_1, from x to a such that every node on P_1 belongs to $T(b)$. In particular, $x \in T(b)$ and let the tree path from b to x be $P_2 = \langle b = b_0, b_1, b_2, \ldots, b_n = x \rangle$. Clearly, $b_i \in L(a, b)$, $1 \leqslant i \leqslant n$ (consider the path consisting of the portion of P_2 from b_i to x augmented with P_1). Similarly, there is a tree path $P_3 = \langle d = d_0, d_1, d_2, \ldots, d_m = x \rangle$ such that $d_i \in L(c, d)$, $1 \leqslant i \leqslant m$. P_2 and P_3 give us two tree paths from two distinct nodes d and b to x. Since each node (except *ENTRY*) has exactly one incoming tree arc, this is possible only when b lies on the path P_3 or when d lies on the path P_2. Without loss of generality, let d lie on path P_2. Then $d \in L(a, b)$, and by Lemma 2.4 $L(c, d) \subset L(a, b)$, implying $Z = L(c, d)$. □

Let S_x be the set of sources of back arcs with sink x.

$$S_x = \{ y : y \in V, \exists \text{ back arc } (y, x) \}.$$

Next we define the set of strongly connected regions, $I(x)$, for all nodes x in $H(G)$.

$$I(x) = \bigcup_{y \in S_x} L(y, x)$$

For the example of Fig. 3, we get Table II.

2.3. HTG

Let $\mathscr{L}'(G) = \{ I(x) : x \in H(G) \} \cup \{ V \}$. $\mathscr{L}'(G)$ is the set of strongly connected regions of G with an additional element, V.

Theorem 2.1. For any two elements A and B in $\mathscr{L}'(G)$,

$$A \cap B = \begin{cases} \varnothing \\ A \\ B \end{cases}$$

Table II. Strongly Connected Regions for Figure 3.

Node x	Region $I(x)$
$I(1)$	$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
$I(3)$	$\{3, 4, 5, 6, 7, 8, 10\}$
$I(4)$	$\{4, 5, 6, 7, 8, 10\}$
$I(7)$	$\{7, 8, 10\}$

Proof: The theorem is obviously true if either A or B is V; hence the only nontrivial case is when $A = I(x)$ and $B = I(y)$ for some x, $y \in H(G)$. If $x = y$, then $I(x) = I(y)$, so let $x \neq y$. By definition,

$$I(x) = \bigcup_{x_i \in S_x} L(x_i, x)$$

and similarly

$$I(y) = \bigcup_{y_i \in S_y} L(y_i, y)$$

Assume $I(x) \cap I(y)$ is nonempty and let $z \in (I(x) \cap I(y))$. Then there exist $x_a \in S_x$ and $y_b \in S_y$ such that $z \in L(x_a, x)$ and $z \in L(y_b, y)$. By Lemma 2.5, $L(x_a, x) \cap L(y_b, y)$ is either $L(x_a, x)$ or $L(y_b, y)$. Without loss of generality, let $L(x_a, x) \cap L(y_b, y)$ be $L(x_a, x)$. This implies that $x \in L(y_b, y)$. Hence, by Lemma 2.4, $L(x_i, x) \subset L(y_b, y)$ for all $x_i \in S_x$ and thus $I(x) \subset L(y_b, y) \subseteq I(y)$. □

For any element $A \in \mathscr{L}'$, we define $P(A)$ to be the set $\{B: B \in \mathscr{L}', A \subset B\}$. For the example of Fig. 3, we get Table III. If C, $D \in P(A)$, then since $C \cap D$ is nonempty (both supersets of A), by Theorem 2.1 either $C \subseteq D$ or $D \subseteq C$. This defines a total ordering on $P(A)$. Hence, if $P(A)$ is nonempty, there exists a unique minimum element $P_{\min}(A)$ in $P(A)$ such that for any element $B \in P(A)$, $B \neq P_{\min}(A)$, $P_{\min}(A) \subset B$. We define the function $f\colon \mathscr{L}' \to (\mathscr{L}' \cup \{\text{NULL}\})$ by

$$f(A) = \begin{cases} \text{NULL} & \text{if } P(A) = \varnothing \\ P_{\min}(A) & \text{otherwise} \end{cases}$$

Since $A \subseteq V$ for all $A \in \mathscr{L}'$, $P(A)$ is the empty set ($f(A) = \text{NULL}$) only when $A = V$. We define $\mathscr{L}$ to be $\mathscr{L}' \cup V$. For the example graph of Fig. 3, $\mathscr{L}' = \{I(1), I(3), I(4), I(7), V\}$, and $\mathscr{L} = \{I(1), I(3), I(4), I(7), \text{V, ENTRY},$

Table III. Ancestors of Members of $\mathscr{L}'$ for Figure 3

A	$P(A)$
$I(1)$	$\{V\}$
$I(3)$	$\{I(1), V\}$
$I(4)$	$\{I(3), I(1), V\}$
$I(7)$	$\{I(1), I(3), I(4), V\}$
V	$\{\ \}$

EXIT, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}. We extend the domain of f to $\mathscr{L}$. If $x \in V$, let $P(x)$ be the set $\{B: B \in \mathscr{L}', x \in B\}$. Again, the intersection of any two elements in $P(x)$ is nonempty as x belongs to all of them and $P(x)$ is nonempty as $V \in P(x)$ for all $x \in V$. As before, we can prove the existence of $P_{\min}(x)$, and we define $f(x) = P_{\min}(x)$. Finally, for notational convenience later, we define $f^0(x)$ to be the identity function and $f^n(x)$ to be the composition of f n times, for any $x \in \mathscr{L}$.

The function f defines a rooted tree with root V on elements of $\mathscr{L}$. This is illustrated in the tree of Fig. 5. There is an arc from A to B only if $f(A) = B$. $P(A)$ is the set of proper ancestors of A in the tree.

We are now in a position to define the nodes and the control flow arcs in the HTG. The hierarchical task graph (at any level) is a directed acyclic graph HTG $= (HV, HE)$. Each node, X, in HV can be of one of the following types:

1. *start* node. The start node has no incoming arcs, and there is a path from it to every node in HV.

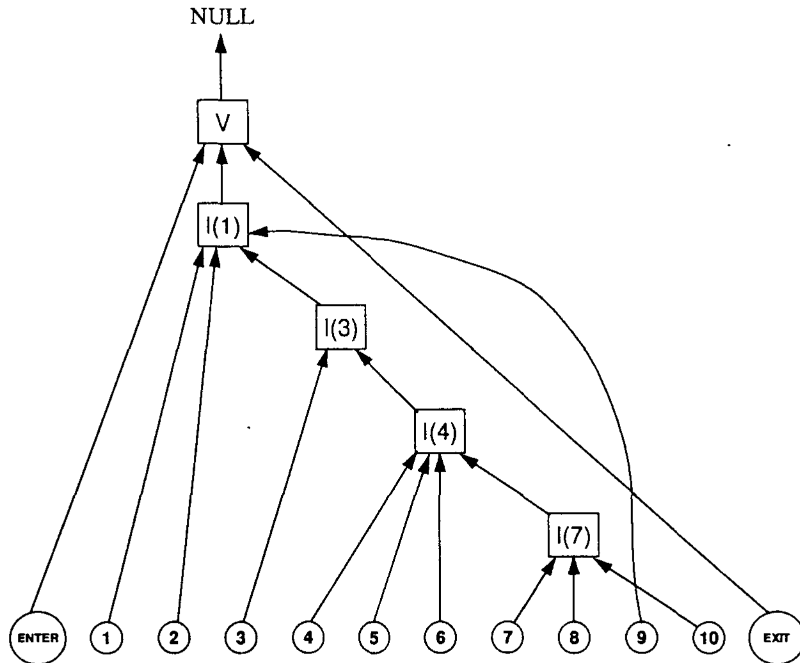

Fig. 5. Hierarchical loop structure for Fig. 3.

2. *stop* node. The stop node has no outgoing arcs, and there is a path from every node in HV to it.
3. *simple* node representing a task that has no subtasks.
4. *compound* node representing a task that consists of other taks in an HTG. Each such compound node has an underlying subgraph $HTG(X) = (HV(X), HE(X))$. We use $START(X)$, $STOP(X) \in HV(X)$ to denote the start and stop nodes in $HTG(X)$.
5. *loop* node representing a task that is a loop whose iteration body is an HTG similar to the body for the compound node.

We first construct all the HTG nodes; all elements of $\mathscr{L}$ correspond to an HTG node. We will denote this correspondence by the function g from $\mathscr{L}$ to the set of HTG nodes; i.e., if $x \in \mathscr{L}$, then $g(x)$ is the corresponding HTG node. In addition to this, there are start and stop nodes corresponding to x if $x \in \mathscr{L}'$. Thus the set of HTG nodes at all levels is given by

$$V_{HTG} = \{g(x) \mid x \in \mathscr{L}\} \cup \{\mathrm{START}(g(x)) \mid x \in \mathscr{L}'\} \cup \{\mathrm{STOP}(g(x)) \mid x \in \mathscr{L}'\}.$$

Next we define the type of all nodes, X, in V_{HTG}.

1. $X = \mathrm{START}(g(x))$. X is of type *start*.
2. $X = \mathrm{STOP}(g(x))$. X is of type *stop*.
3. $X = g(x), x \in \mathscr{L}$.
   (a) $x \in V$. X is of type *simple*.
   (b) $x = V$. X is of type *compound*.
   (c) $x = I(y)$ for some $y \in H(G)$. X is of type *loop*.

The only HTG nodes which are not targets of g are the various start and stop nodes. We define the function F from V_{HTG} to $V_{\mathrm{HTG}} \cup \{\mathrm{NULL}\}$ corresponding to the function f.

1. $X = \mathrm{START}(g(x))$. $F(X) = g(x)$.
2. $X = \mathrm{STOP}(g(x))$. $F(X) = g(x)$.
3. $X = g(x), x \in \mathscr{L}$.
   (a) $x \in V$. $F(X) = g(f(x))$.
   (b) $x = V$. $F(X) = \mathrm{NULL}$.
   (c) $x = I(y)$ for some $y \in H(G)$. $F(X) = g(f(x))$.

Note that if we make the trivial assumption that $g(\mathrm{NULL}) = \mathrm{NULL}$, then

$$F(g(x)) = g(f(x)) \tag{2.1}$$

Next we define the HV set for every constructed compound and loop HTG node, X, $HV(X) \subseteq V_{\text{HTG}}$.

$$HV(X) = \{Y: F(Y) = X\}$$

We now construct the arcs in the HTG. Each arc (a, b) in the flow graph G will result in new arcs being added to the HTG. For every arc (a, b) in the G, we do the following (see Fig. 6):

1. Find z, the least common ancestor of $f(a)$ and $f(b)$ in the f tree (if $f(a) = f(b)$, then $z = f(a)$). Let $z = f^n(a) = f^m(b)$, $n, m \geqslant 1$.
2. Add arcs $(g(f^i(a)), \text{STOP}(g(f^{i+1}(a))))$ to $HE(g(f^{i+1}(a)))$ for $0 \leqslant i \leqslant n-2$.
3. Add arcs $(\text{START}(g(f^i(b)), g(f^{i-1}(b))))$ to $HE(g(f^i(a)))$ for $1 \leqslant i \leqslant n-1$.
4. If (a, b) is not a back arc, add the arc $(g(f^{n-1}(a)), g(f^{m-1}(b)))$ to $HE(g(z))$.
5. If (a, b) is a back arc, the arcs $(g(f^{n-1}(a)), STOP(g(z)))$ and $(START(g(z)), g(f^{m-1}(b)))$ are added to $HE(g(z))$.

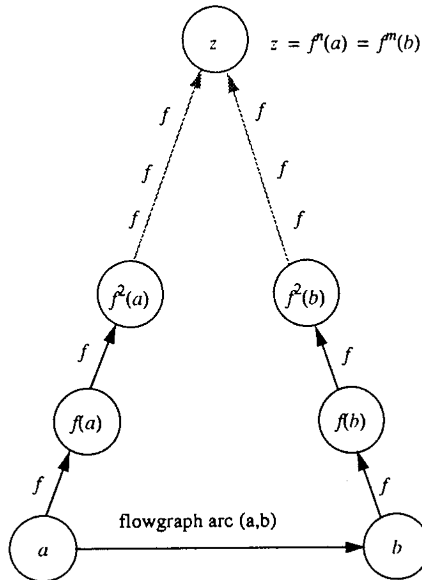

Fig. 6. Considering flowgraph arcs.

Two additional arcs, $(START(g(V)), g(ENTER))$ and $(g(EXIT), STOP(g(V)))$, corresponding to the *ENTRY* and *EXIT* nodes of the flow graph, are also added to $HE(g(V))$. We note from the construction and Eq. (2.1) that if an arc is added between HTG nodes X and Y, then $F(X) = F(Y)$ and further that by the process of factoring out the back arcs in the original flow graph, the graph $\text{HTG}(X) = (HV(X), HE(X))$ is acyclic for any compound or loop node X.

The HTG graph of node $g(I(1))$ in our example is shown in Fig. 7. The nodes in the flow graph for $g(I(1))$ are $START(g(I(1)))$, $STOP(g(I(1)))$, $g(1)$, $g(2)$, $g(9)$, $g(I(3))$, as $f(1) = f(2) = f(9) = f(I(3)) = I(1)$. We have shown how to construct the control flow graphs of the HTG at each level in this section. In the next section we will define the control and data dependence graphs on the HTG at each level, making the construction of the HTG complete.

3. AUGMENTING THE HTG WITH CONTROL AND DATA DEPENDENCES

In Section 2 we showed how the HTG can be built from the control flow graph of the program. In this section we concentrate on the acyclic

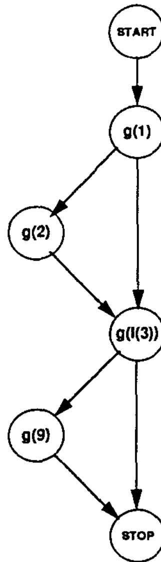

Fig. 7. Flowgraph for $g(I(1))$.

control flow graph corresponding to an HTG loop or compound node. We augment it with control and data dependence graphs and show parallelism can be extracted from such a graph and study the problems encountered in the process.

3.1. Preliminaries

Our starting point will be the control flow graph of any compound or loop HTG node. We will denote this graph by $CFG = (V, E)$ with unique nodes $START$, $STOP \in V$ such that there exists a path from $START$ to every node in V and a path from every node to $STOP$; $START$ has no incoming arcs, and $STOP$ has no outgoing arcs. Figure 8 shows an example acyclic control flow graph with 11 nodes and 13 arcs.

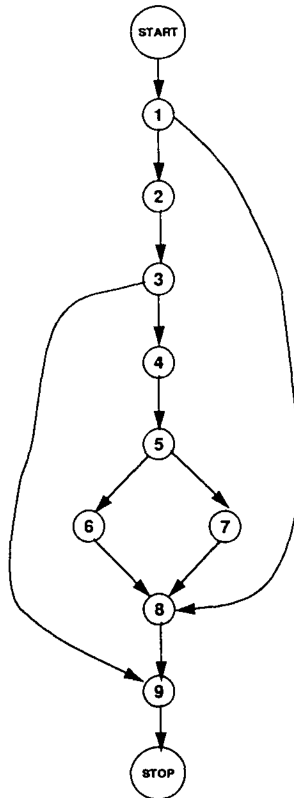

Fig. 8. Example control flow graph.

Node y *post-dominates* node x, denoted by $y\Delta_p x$, iff every path from x to *STOP* (not including x) contains y.[11] A node never post-dominates itself. We use $y\not\Delta_p x$ to denote y does not post-dominate x. The *reflexive closure* of the post-dominance relation will be denoted by $\bar{\Delta}_p$, $y\bar{\Delta}_p x$ iff $y\Delta_p x$ or $y = x$. The following is well known.[11]

Lemma 3.1. Let y and z be distinct nodes. For any x, if $y\bar{\Delta}_p x$ and $z\bar{\Delta}_p x$, then either $y\Delta_p z$ or $z\Delta_p y$.

Lemma 3.1 suffices to show that the set of post-dominators of a node x form a chain. The least element in the chain is called the *immediate post-dominator* of x. The set of post-dominators of a node x is nonempty (except when x is the STOP node) as STOP $\Delta_p x$. Hence, all nodes except *STOP* have a unique immediate post-dominator. If we draw an arc from x to y whenever x is an immediate post-dominator of y, the resulting graph is a tree rooted at *STOP* and called the *post-dominator tree*. Figure 9 shows the post-dominator tree for our example control flow graph of Fig. 8.

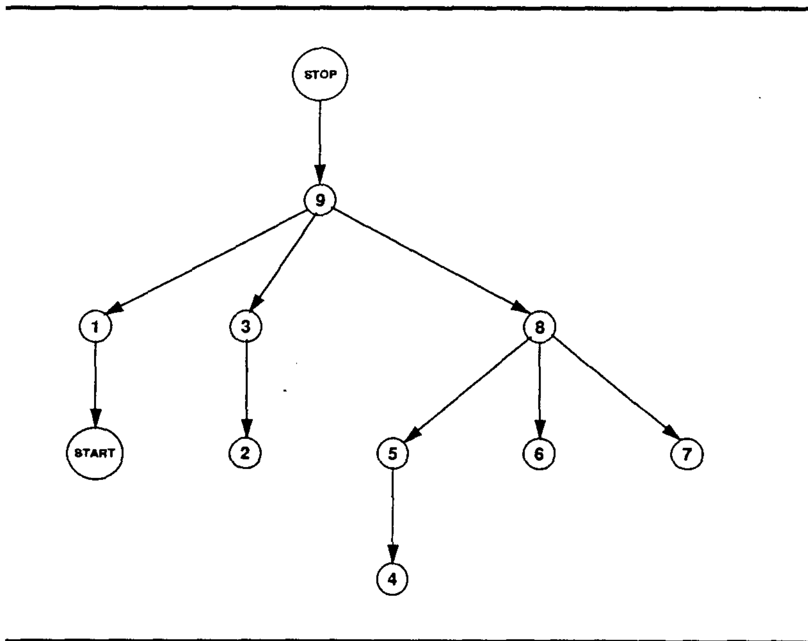

Fig. 9. Post-dominator tree for Fig. 8.

3.2. The Control Dependence Graph (CDG)

Node y is *control dependent* on node x with *label* $x-a$ ((x, a) is an arc in CFG), denoted by $x\delta_c y$, iff

1. $y \not\Delta_p x$, and
2. $\exists$ a non-null path $P = \langle x, a, ..., y \rangle$, such that for any $z \in P$ (excluding x and y), $y \Delta_p z$.

Our definition of control dependence differs only slightly from Ref. 11, where nodes were restricted to have at most two outgoing arcs; we relax this restriction. An immediate consequence of the definition is that if $x\delta_c y$ with label $x-a$, then $y\bar{\Delta}_p a$. We will say that an arc (x, a) is a *branch* if x has more than one outgoing arc. Note that the labels in the control dependence relation can only be branches.

The *control dependence graph* CDG, of a control flow graph CFG, is defined as the directed graph with labeled arcs, $\text{CDG} = (CV, CE)$ such that

1. $CV = V$ and
2. $(x, y) \in CE$ with label $x-a$ iff $x\delta_c y$ with label $x-a$.

CDG can be built from CFG using the post-dominance tree[11] as follows. If (x, y) is any branch in CFG, then the following can be shown.

1. Let z be the immediate ancestor of x in the post-dominator tree. Then the least common ancestor of x and y in the post-dominator tree, $\text{LCA}(x, y)$, is either x or z.
2, All nodes on the path from y to z (not including z) in the post-dominator tree are control dependent on x with label $x-y$.

Figure 10 shows the control dependence graph for our example control flow graph.

The *transitive closure* of δ_c will be denoted by δ_c^*, $x\delta_c^* y$ iff there exists a non-null path from x to y in CDG. This corresponds to the notion of the range of a branch given in Ref. 30. The reflexive closure of δ_c^* will be denoted by $\bar{\delta}_c^*$, $x\bar{\delta}_c^* y$ iff $x\delta_c^* y$ or $x = y$.

We state the following important theorem; a proof can be found in Ref. 2. A related result for forward control dependence graphs is proved in Ref. 14.

Theorem 3.1. CDG is cyclic iff CFG is cyclic.

We will be dealing exclusively with acyclic control flow graphs from now on. In an acyclic CFG it is possible to assign a unique number to each node in such a way that if there is a path from a node numbered x to a

node numbered y, then $x < y$. From now on, we will assume that such a numbering has been done in our CFG, and we will refer to a node numbered x as simply node x.

3.3. The Data Dependence Graph (DDG)

Node y *conflicts* with node x if either x or y share access to a common memory location, at least one of which is a "write" operation. Conflicts induce a data dependence[9,31–34] relation among nodes. Exactly one of the following can occur between two distinct nodes x and y.

1. y is reachable from x in CFG.
2. x is reachable from y in CFG.
3. x is not reachable from y, and y is not reachable from x in CFG.

If x and y conflict with each other, then we say that y is *data dependent* on x in Case 1 (denoted by $x\delta_d y$), and x is data dependent on y in Case 2 ($y\delta_d x$). In Case 3 the conflict does not matter and can be ignored; we will assume that Case 3 does not occur.

The *data dependence graph* DDG = (DV, DE) is defined as the directed graph such that

1. $DV = V$ and
2. $(x, y) \in DE$ if $x\delta_d y$.

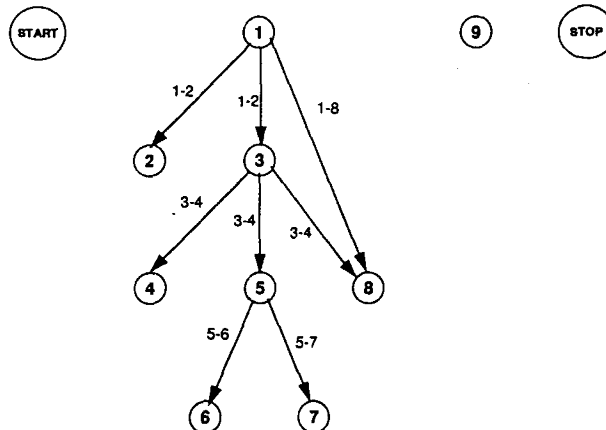

Fig. 10. Control dependence graph for Fig. 8.

Figure 11 shows a possible data dependence graph for our example control flow graph.

Note that since $x\delta_d y$ implies a path from x to y in CFG, the graph containing the arcs of both CFG and DDG is also acyclic owing to the acyclicity of CFG. Similarly, the graph containing the arcs of both CDG and DDG is also acyclic. In the terminology of Ref. 34, we are restricting ourselves to dependences whose direction vectors consist of only "=" components.

3.4. Parallel Execution

In the absence of data dependences, the parallel execution of CFG is based on CDG[14] where identically control dependent nodes are executed in parallel:

1. Initially, only nodes that do not have any incoming arcs in the CDG begin execution in parallel.
2. After executing a node, for example, x, if label $x-a$ is true (i.e., the branch $x-a$ would have been taken in the sequential execution of CFG), then all nodes y such that $x\delta_c y$ with label $x-a$ start execution in parallel.

The execution terminates when all nodes finish execution. By Theorem 2.2, CDG is acyclic and hence it is obvious that the parallel execution will terminate.

Let $\mathscr{S}$ and $\mathscr{P}$ denote the *sequential* and *parallel* execution of CFG respectively. $\mathscr{S}$ specifies a single path, P, in CFG from *START* to *STOP*. The parallel execution of CFG specifies a set of trees in the control dependence graph; in the forward control dependence graph, which is connected, it specifies a single tree.[14]

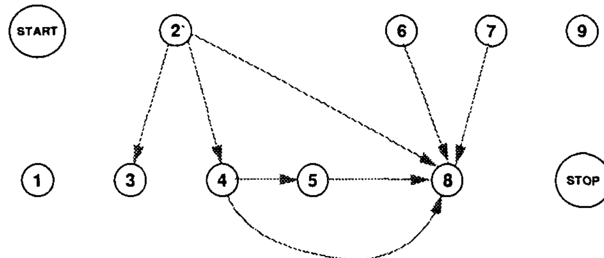

Fig. 11. Possible data dependence graph for Fig. 8.

A node is executed in $\mathscr{S}$ if it lies on P. A label $x-y$ will be *true* in $\mathscr{P}$ if the arc $x-y$ lies on P. According to this model, a node x is executed in $\mathscr{P}$ when there is a path in the CDG, $\langle a_0, a_1,..., a_n = x\rangle$ such that a_0 is a node with no incoming arcs and $a_j \delta_c a_{j+1}$ $(0 \leqslant j < n)$ with some true label $a_j - b_j$.

The following theorem states the correctness of the parallel execution. A proof is given in Ref. 2.

Theorem 3.2. Let CFG be acyclic. The parallel execution of CFG executes the same nodes as the sequential execution.

3.5. Precedence and the Closure Graph (CDDG)

Based on this execution model, we try to define the idea of one node executing before another. Intuitively, node y can *execute before* node x if there is a parallel execution such that y can begin execution before x. We aim to formalize this definition purely in graph theoretic terms. Again, in the absence of data dependences, the following definition works. Node y can execute before node x if there is a path in the CFG from *START* to *STOP* which contains x and y (this must be true for y and x to be executed in the same execution) and there is no path in the CDG from x to y. In our example graph, clearly there is a path in CFG from *START* to *STOP* containing 3 and 4. Node 3 can execute before 4 because there is no path from 4 to 3 in the CDG; however 4 cannot execute before 3 as $3\delta_c 4$. A more interesting case is of the nodes 2 and 3, where we find that 2 can execute before 3 and 3 can execute before 2 as 2 and 3 are disconnected in the CDG. Note that we cannot say 6 can execute before 7 because even though there is no path from 7 to 6 in the CDG, there is no path in the CFG from *START* to *STOP* containing both 6 and 7.

With the addition of data dependences, our execution model undergoes some changes. Now, before a node y can start execution, it should also be checked to determine if the necessary data dependence conditions are also satisfied. Thus, if there is a data dependence arc $x\delta_d y$, before y can start execution, we have to make sure that x has finished execution or that x will not be executed at all. Our initial impulse would be to try to formalize this definition in terms of paths in the CDG + DDG. However, this is rather hard as the following two *incorrect* definitions illustrate.

Node y can execute before node x if there is a path in the CFG from *START* to *STOP* which contains x and y and there is no path in the *CDG + DDG* from x to y. This is a reasonable first attempt, but we show that this definition is too restrictive. It is possible for node y to execute before node x even though there is a path from x to y in the CDG + DDG

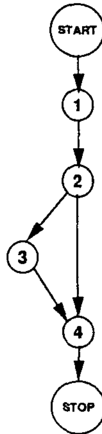

Fig. 12. Another example flow graph.

as the following example shows. Consider the control flow graph shown in Fig. 12 with its CDG + DDG graph as shown in Fig. 13. Clearly, there is a path from 1 to 4 in the CDG + DDG as $1\delta_d3$ and $3\delta_d4$. However, 4 can conceivably execute before 1 in the following execution sequence.

1. Node 2 starts and finishes execution.
2. The branch 2–4 is taken.
3. Node 4 can now begin execution as it is now clear that 3 will not execute and hence the dependence 3–4 can be ignored.
4. Node 1 begins execution.

The problem with this definition was that the original path in CDG + DDG was through node 3, and node 3 is not executed in the

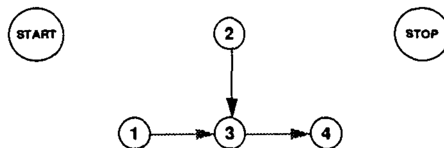

Fig. 13. CDG + DDG for Fig. 12.

counter-example. We can solve this problem by requiring that the path in CDG + DDG be restricted on only those nodes that are executed in the CFG. We first formally define a restriction on a graph.

If P is a path in a graph, let $N(P)$ be the set of nodes in P. If $G(V, E)$ is a graph and $A \subseteq V$, let G_A be the subgraph $G_A(V', E')$ of G *restricted* on A where

1. $V' = A$ and
2. $(x, y) \in E'$ if $x, y \in A$ and $(x, y) \in E$.

Now we can try to fix our previous definition and use the following instead. Node y *can execute before* node x if there exists a path P in CFG from $START$ to $STOP$ containing x and y such that there is no path from x to y in $(\text{CDG} + \text{DDG})_{N(P)}$. This gets rid of the preceding problem in the counter example, because once the branch 2–4 is taken, we will be restricted in the CDG + DDG on the nodes $\{START, 1, 2, 4, STOP\}$ and there is no path from 1 to 4 in the CDG + DDG when restricted on this set.

This definition is still in error; it is possible according to the definition for us to say that node y can execute before x even when such a thing will not occur. Consider the example control flow graph shown in Fig. 14 with its control and data dependence graph shown in Fig. 15. If we consider the path $P = \langle START, 1, 3, STOP \rangle$, we see that there is no path from 1 to

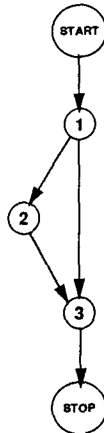

Fig. 14. Example control flow graph.

3 in the $(\text{CDG}+\text{DDG})_{N(P)}$ as $2 \notin N(P)$; hence according to the definition node, 3 should be able to execute before node 1. However, we see that this can never happen, because for node 3 to start execution it has to know whether 2 has finished execution or 2 will not execute at all. In either case, node 1 will have finished execution, and hence node 3 will always start after node 1 has finished. To correct this problem we need to define a new graph from the CDG + DDG, which we call the closure graph.

The *closure* of the data and control dependence graph $\text{CDDG} = (\text{CDV}, \text{CDE})$ is defined as the directed graph with arcs such that

1. $CDV = V$ and
2. $(x, y) \in \text{CDE}$ if $x\delta_c^* y$ or $\exists z$ such that $x\bar{\delta}_c^* z$ and $z\delta_d y$.

Note that an arc from x to y in CDDG implies a path from x to y in CDG + DDG, and thus CDDG is also acyclic. Figure 16 shows the closure graph for our data and control dependence graphs in Figs. 10 and 11. The thick arcs in Fig. 16 represent additional arcs which were not present in the CDG + DDG.

We can now give the precise definition of when a node can execute before another. We say that node y *can execute before* node x if there exists a path P in CFG from *START* to *STOP* containing x and y such that there is no path from x to y in $\text{CDDG}_{N(P)}$.

We say that node x *precedes* node y (written $x \prec y$) if for any path P in CFG from *START* to *STOP* containing x and y there is a path from x to y in $\text{CDDG}_{N(P)}$. Note that for any such path P, the path in $\text{CDDG}_{N(P)}$ will consist only of nodes on P between x and y. If we use $x \not\prec y$ to denote x does not precede y, then y can execute before x is equivalent to $x \not\prec y$.

We point out some special cases.

1. Since the trivial null path from x to x exists in the CDDG, $x \prec x$.
2. If there is no path in CFG from *START* to *STOP* containing x and y, then trivially, $x \prec y$ and $y \prec x$.
3. $x \not\prec y$ does not necessarily imply $y \prec x$.

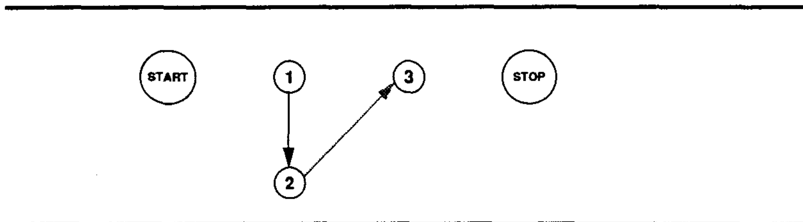

Fig. 15. The CDG + DDG for Fig. 14.

The notion of precedence was also defined in Ref. 18 in a different context. In Ref. 18, the definition is in terms of parallel executions and time as opposed to the graph theoretic one given earlier. We summarize some of the important aspects by which this work differs from Ref. 18.

1. The problem of proving a given parallel program correct with respect to a sequential interpretation is considered in Ref. 18. As such, the parallelism is given by the constructs used in the program rather than detected from a sequential program.
2. Although conditional statements are used in some of the proofs in Ref. 18, explicit control dependences are not used.
3. In Ref. 18, the synchronization arcs arise due to **post** and **wait** statements in the program. Hence, a node can begin execution when *any* of its predecessors finish execution. In our description of the problem, the analog to synchronization arcs in Ref. 18 are the data dependence arcs. However, a node can begin execution only when *all* data dependences incident to it are satisfied. The satisfaction of data dependences is also not equivalent to the completion

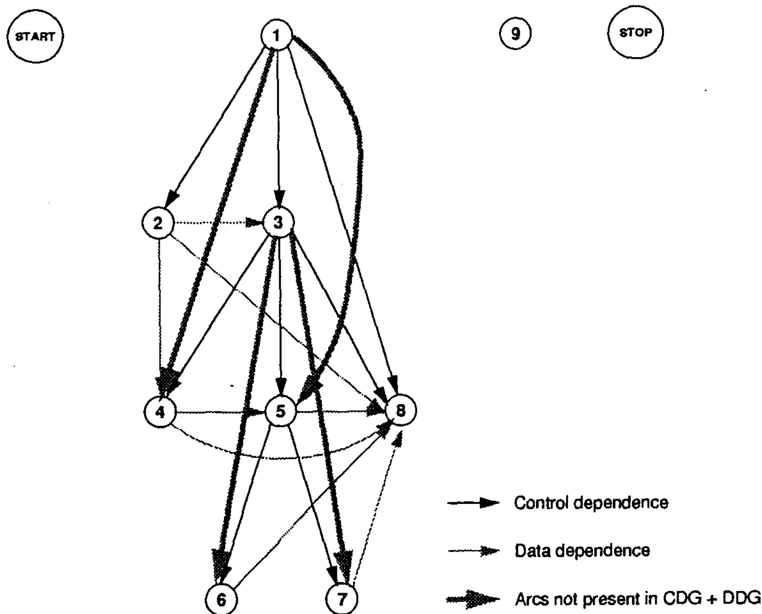

Fig. 16. The closure graph for Fig. 8.

of the execution of the source node; data dependences can also be satisfied when it can be determined that the source node will not be executed. This changes the nature of the proofs and also may be the reason that it does not seem possible to define the notion of precedence on the basis of the graph terms alone in Ref. 18.

3.6. Example to Illustrate Precedence

Consider the example flow graph and its CDG + DDG shown in Figs. 12 and 13 respectively. The closure graph is shown in Fig. 17. The additional arc (2, 4) is present because $2\delta_c 3$ and $3\delta_d 4$. Consider the problem of determining whether $1 \prec 4$ (this is the same as determining 4 can execute before 1). There are two paths in the CFG from *START* to *STOP* containing both 1 and 4; $P_1 = \langle START, 1, 2, 4, STOP \rangle$ and $P_2 = \langle START, 1, 2, 3, 4, STOP \rangle$. There is no path from 1 to 4 in $\mathrm{CDDG}_{N(P_1)}$, but there is a path from 1 to 4 in $\mathrm{CDDG}_{N(P_2)}$, namely, $\langle 1, 3, 4 \rangle$. Due to the existence of path P_1, 4 can execute before 1, or in other words, $1 \not\prec 4$.

4. IMPLEMENTATION

The HTG has been implemented as part of the Parafrase-2 compiler.[35] Details of this implementation are given in this section. The HTG was built according to the procedures outlined in previous sections and some static analysis of programs was done to measure the "amount" of functional parallelism.

4.1. Parafrase-2

Parafrase-2 is a multilingual restructuring compiler. A block diagram of Parafrase-2 is shown in Fig. 18. The *core* of Parafrase-2 works on intermediate data structures used to represent the source program. Optimiza-

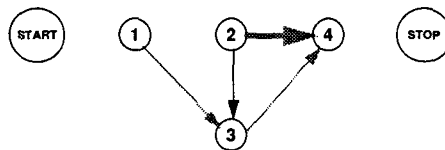

Fig. 17. The closure graph for Figs. 12 and 13.

tions on the code are done through *passes* which operate on these data structures. The HTG is used as the main data structure in the auto-scheduling part of the compiler. The HTG is built from the loops detected in the flow graph of the program (generated by the *flow* and *induction* passes[36]) and is only slightly different from the construction described in Section 2. It has the following type of nodes.

1. *start* nodes.
2. *stop* nodes.

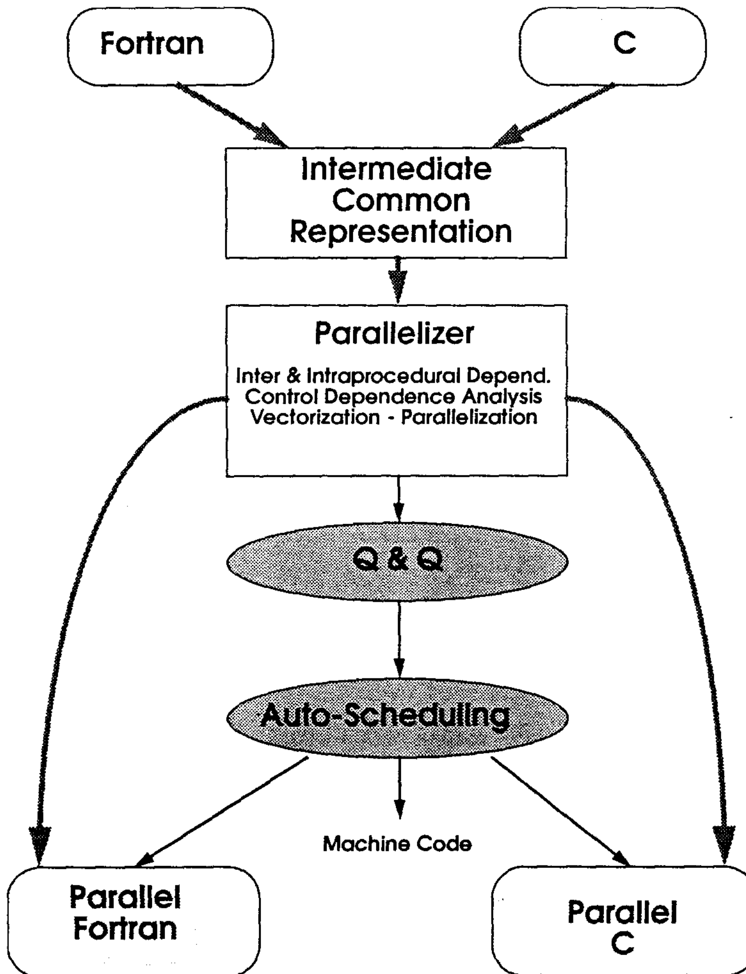

Fig. 18. The Parafrase-2 restructuring compiler.

3. *basic* nodes. These correspond to simple statements in the program.
4. *call* nodes. These correspond to statements having a subroutine call.
5. *loop* nodes. These correspond to the loops in the program.
6. *compound* nodes. These correspond to the basic blocks in the flow graph. The top level of the HTG is also a compound node.

4.2. Experiment

An experiment was done to analyze statically the effect of functional parallelism in a program. It was designed to measure the nature of the parallelism obtained. In the absence of proper timing measurements, the following two approximate measures were used.

1. The ratio of the length of the longest path (in terms of the number of nodes on the path) in the derived CDDG to the length of the longest path in the original CFG of HTG loop and compound nodes. We denote this ratio by r.
2. The fraction of instances when two identically control dependent HTG nodes can be executed in parallel,

The HTG of program MDG, a molecular dynamic simulation program which is a part of the Perfect Benchmarks™,[37] was built and analyzed. Results of the experiment are given in Tables IV and V. We explain the entries in the tables. The entry of 15 ($.50 \leqslant r \leqslant .75$, Loop) in Table IV indicates that 15 loop nodes in the HTG had their iteration body (measured in number of nodes) shortened to between 50 and 75 percent of the original. The entry of 29.73 (Loop, Compound) in Table V indicates

Table IV. Ratio of Longest Paths in CDDG and CFG for Program MDG

	Number of nodes	
Ratio r	Loop	Compound
$0 \leqslant r \leqslant .25$	0	29
$.25 < r \leqslant .50$	1	54
$.50 < r \leqslant .75$	15	27
$.75 < r \leqslant 1.0$	38	180

Table V. Parallelism in Identically Control Dependent Nodes for Program MDG

	Basic	Call	Loop	Compound
Basic	76.18	21.94	—	—
Call	21.94	17.65	—	—
Loop	—	—	4.55	29.73
Compound	—	—	29.73	44.92

that whenever a loop and a compound node have been found to be identically control dependent they can be executed in parallel in 29.73 percent of those instances. Note that Table V is symmetric.

4.3. Discussion of the Results

The following observations can be made from the data obtained.

1. From Table IV, it is clear that the iteration bodies of loops or the bodies of basic blocks can be considerably shortened ($r \leqslant .50$ for $52 + 26 = 78$ compound nodes) by finding functional parallelism.
2. Table V indicates that when two HTG nodes are identically control dependent, they are independent of data dependences and hence can be executed in parallel a significant proportion of the time. Some of the entries in Table V are blank, because that particular combination of nodes will not occur because of the way the HTG is constructed. For example, a basic HTG node which corresponds to a simple statement is always nested inside a flowgraph node which corresponds to a compound HTG node. Thus it is always the compound HTG node which is at the level of other loop or compound nodes in the HTG and never the basic HTG node. From Table V it can be seen that the largest amount of parallelism is available among simple statements. The parallelism among simple statements will lead to significant decreases in execution time if this parallelism is nested inside loops; otherwise it may not have much of an effect, as normally the execution time of statements without subroutine calls will be small. Another encouraging item of interest is that about 17.65 percent of the time a pair of subroutine calls can be done in parallel with each other.

5. CONCLUSION

As presented in this paper, the HTG can serve as an intermediate program representation for a variety of high-level languages. In addition to consolidating the control flow, data dependence, control dependence, and syntax tree structures into a single representation, the HTG exposes the notion of precedence relations between program statements of different granularity. Precedence information is vital for efficient and correct parallel code generation. The HTG has been implemented in Parafrase-2 as the IR target for Fortran and C, as well as the source for program optimizations and parallel code generation.

REFERENCES

1. M. Furnari, Compiling scheme into the HTG, In *Proc. of the 1992 Workshop on Parallel Languages and Compilers* (August 1991).
2. M. Girkar, Functional Parallelism: Theoretical Foundations and Implementation, PhD. Thesis, Center for Supercomputing Research and Development, University of Illinois, Urbana-Champaign (January 1992).
3. J. Moreira, The Design and Performance of an Auto-scheduling Environment, PhD. Thesis, Center for Supercomputing Research and Development, University of Illinois, Urbana-Champaign, In preparation, (September 1993).
4. C. Beckmann, Hardware and Software for Functional and Fine Grain Parallelism, PhD. Thesis, Center for Supercomputing Research and Development, University of Illinois, Urbana-Champaign (1993).
5. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, Massachusetts (March 1986).
6. J. T. Schwartz and M. Sharir, A design for optimizations of the bitvectoring class, Technical Report, Courant Computer Science Report No. 538, Courant Institute of Mathematical Sciences, New York University (September 1979).
7. J. McGraw *et al.* SISAL: Streams and iteration in a single assignment language, language reference manual, Version 1.2. Technical Report M-146, LLNL (March 1985).
8. V. Sarkar, Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors, PhD. Thesis, Computer Systems Laboratory, Department of Electrical Engineering and Computer Science, Stanford University (April 1987).
9. J. R. Allen, Dependence analysis for subscripted variables and its application to program transformations, PhD. Thesis, Department of Mathematical Sciences, Rice University, Houston, Texas (April 1983).
10. U. Banerjee, Speedup of Ordinary Programs, PhD. Thesis, Department of Computer Science, University of Illinois, Urbana-Champaign (October 1979).
11. J. Ferrante, K. J. Ottenstein, and J. D. Warren, The program dependence graph and its use in optimization, *ACM Trans. on Programming Languages and Systems*, **9**(3):319–349 (July 1987).
12. K. Kennedy and K. McKinley, Loop distribution with arbitrary control flow, In *Proc. Supercomputing '90*, Los Alamitos, California pp. 407–416, IEEE Computer Society Press. (November 1990).

13. H. Baxter and H. Bauer, III, The program dependence graph and vectorization, In *Conference Record of the 16th ACM Symp. on the Principles of Programming Languages*, pp. 1–10 (January 1989).
14. R. Cytron, M. Hind, and W. Hsieh, Automatic generation of DAG parallelism, In *Proc. of the 1989 SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 54–68 (July 1989).
15. W. Hsieh, Extracting parallelism from sequential programs, Master's Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (May 1988).
16. R. Cytron, J. Ferrante, and V. Sarkar, Experiences using control dependence in PTRAN, In D. Gelernter, A. Nicolau, and D. A. Padua, (edits.), *Languages and Compilers for Parallel Computing*, MIT Press, pp. 186–212 (1990).
17. H. Kasahara, H. Honda, M. Iwata, and M. Hirota, A compilation scheme for macro-dataflow computation on hierarchical multiprocessor systems, unpublished manuscript (1989).
18. D. Callahan and J. Subhlok, Static analysis of low-level synchronization, In *Proc. of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 100–111 (May 1988).
19. E. Duesterwald, Static concurrency analysis in the presence of procedures, Technical Report 91-6, Department of Computer Science, University of Pittsburgh (March 1991).
20. S. Masticola and B. Ryder, A model of Ada programs for static deadlock detection in polynomial time, In *Proc. of the 1991 ACM Workshop on Parallel and Distributed Debugging*, (1991).
21. D. Jayasimha, Communication and Synchronization in Parallel Computation, PhD. Thesis, Center for Supercomputing Research and Development, University of Illinois, Urbana-Champaign (1988).
22. V. Krothapalli and P. Sadayappan, Removal of redundant dependences in doacross loop with constant dependences, In *Proc. of the Third SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 51–60 (April 1991).
23. Z. Li and W. Abu-Sufah, On reducing data synchronization in multiprocessed loops, *IEEE Trans. on Computers*, **C-36**(1):105–109 (January 1987).
24. S. P. Midkiff and D. A. Padua, Compiler algorithms for synchronization, *IEEE Trans. on Computers*, **C-36**(12):1485–1495 (December 1987).
25. S. P. Midkiff and D. A. Padua, A comparison of four synchronization optimization techniques, Technical Report No. 1135, Center for Supercomputing Research and Development, University of Illinois, Urbana-Champaign (June 1991).
26. P. Shaffer, Minimization of interprocessor synchronization in multiprocessors with shared and private memory, In *Proc. of the 1989 Int. Conf. on Parallel Processing*, Volume III, pp. 138–141, St. Charles, Illinois (August 1989).
27. F. E. Allen and J. Cocke, A program data flow analysis procedure, *Commun. of the ACM*, **19**(3):137–147 (March 1976).
28. M. Hecht, *Flow Analysis of Computer Programs*, Elsevier North-Holland, Inc., New York, (1977).
29. R. E. Tarjan, Testing flow graph reducibility, *J. of Computer and Syst. Sci.*, **9**(3):355–365 (December 1974).
30. M. Weiser, Programmers use slices when debugging, *Comm. of the ACM*, **25**(7):446–452 (July 1982).
31. R. Allen and K. Kennedy, Automatic translation of FORTRAN programs to vector form, *ACM Trans. on Programming Languages and Systems*, Vol. 9 No. 4 (October 1987).
32. U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers (1988).

33. D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. J. Wolfe, Dependence graphs and compiler optimizations, In *Proc. of the 8th Annual ACM Symp. on Principles of Programming Languages*, pp. 207–218 (January 1981).
34. M. J. Wolfe, *Optimizing Supercompilers for Supercomputers*, The MIT Press, Cambridge, Massachusetts (1989).
35. C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten, Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors, In *Proc. of the 1989 Int. Conf. on Parallel Processing*, St. Charles, Illinois (August 1989).
36. C. D. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten, Parafrase-2 programmer's manual, CSRD Internal document.
37. G. Cybenko, L. Kipp, L. Pointer, and D. Kuck, Supercomputer performance evaluation and the Perfect Benchmarks, In *Proc. of the Int. Conf. on Supercomputing*, Amsterdam, Netherlands (March 1990).nce graph and its use in optimization, *ACM Trans. on Programming Languages and Systems*, 9(3):319–349 (July 1987).