Automatic Extraction of Functional Parallelism from Ordinary Programs

Milind Girkar and Constantine D. Polychronopoulos, *Member, IEEE*

***Abstract*— Thus far, parallelism at the loop level (or data-parallelism) has been almost exclusively the main target of parallelizing compilers. The variety of new parallel architectures and recent progress in interprocedural dependence analysis, suggest new directions for the exploitation of parallelism across loop and procedure boundaries (or functional-parallelism). This paper presents the *Hierarchical Task Graph (HTG)* as an intermediate parallel program representation which encapsulates minimal data and control dependences, and which can be used for the extraction and exploitation of functional, or task-level parallelism. The hierarchical nature of the *HTG* facilitates efficient task-granularity control during code generation, and thus applicability to a variety of parallel architectures. We focus on the construction of the *HTG* at a given hierarchy level, the derivation of the execution conditions of tasks which maximizes task-level parallelism, and the optimization of these conditions which results in reducing synchronization overhead imposed by data and control dependences. We present algorithms for the formation of tasks and their execution conditions based on data and control dependence constraints. Subsequently, we discuss the issue of optimization of such conditions and propose optimization algorithms. The *HTG*, which is being implemented in the Parafrase-2 compiler, is used as the intermediate representation of parallel Fortran and C programs for generating parallel source as well as parallel machine code.**

***Index Terms*—Code generation, control and data dependence, intermediate program representation, parallel languages, parallelizing compilers, synchronization.**

I. Introduction

THE familiar task graph, a directed acyclic graph, has become synonymous with a parallel program. Task graphs have been used as a convenient abstraction of parallel computations and programs in virtually all areas of parallel processing. In as many cases task graphs take the form of a partial ordering imposed on a set of nodes representing computations. Before the recent proliferation of parallel computers and parallel programming, the task graph abstraction (although hardly relevant to the structure of real computations), was a sufficiently powerful model for the theoretical work that characterized early research in parallel processing [8]. Even at present, and several years after the introduction of the first MIMD computers, the task graph remains essentially the only widely used model (for the analysis, scheduling, and simulation) of parallel programs. This is the case because most present-day parallel architectures and computations are structured around the cascaded fork–join or more familiarly, loop-level parallelism (at least in scientific applications). Loop-level parallelism, including the automatic detection of parallelism [4], [17], [24] and the generation of synchronization instructions [19], has been a well studied area.

However, with the shift of attention to nonloop parallelism, suggested by a variety of parallel architectures and made possible by interprocedural analysis, the task graph abstraction becomes quite obsolete in modeling parallel computations. Some of the inherent limitations of task graphs include their deterministic nature (all nodes of the graph are guaranteed to execute), lack of node and edge context (mapping instruction sequences of a given computation to the nodes of such a graph and the lack of a realizable notion of precedence constraints), and overall, their unsuitability to be targeted by parallelizing compilers as a powerful intermediate representation for programs with nonloop-level parallelism. Nonloop parallelism refers, for instance, to executing a number of different subroutine calls (or serial and/or parallel loops) concurrently. Task-level parallelism and high-level spreading have also been used as synonyms for nonloop parallelism.

This paper tackles the problem of automatic extraction of task-level parallelism from serial programs through a parallelizing compiler. Using the notion of loop hierarchies, and data and control dependence, we develop a new representation for parallel programs called the *Hierarchical Task Graph* or *HTG*. The *HTG* represents a powerful intermediate representation which encapsulates program parallelism of different types and scope levels, and is used for the generation and optimization of both parallel source and parallel machine code [13], [21]. The *HTG* is the intermediate representation used in the Parafrase-2 parallelizing compiler [22]. At present, only a handful of parallelizing compilers employ similar mechanisms for task-level parallelism [2], [11], [22].

This paper focuses on the derivation of the *HTG* based on data and control dependences, its fundamental properties, and optimizations. The *HTG* is built at the basic block level and augmented with data and control dependences, computed using approaches similar to those described in [4], [5], [7], [10], [12], [15], and [16]. Although the construction of the hierarchical task graph is briefly outlined, this paper emphasizes the fundamental techniques and optimizations used in the derivation of the *HTG* at a specific hierarchy level, namely that of the basic block.

Manuscript received October 5, 1990; revised May 13, 1991. This work was supported in part by the National Science Foundation under Grant NSF-CCR-89-57310, the U.S. Department of Energy under Grant DOE-DE-FG02-85ER25001, Control Data Corporation, and a grant from Texas Instruments, Inc.

The authors are with the Center for Supercomputing Research and Development, University of Illinois at Urbana–Champaign, Urbana, IL 61801.

IEEE Log Number 9103972.

Section II describes the motivations for our work and points out differences from existing methods. Section III discusses briefly the hierarchical task graph, and Section IV gives basic definitions and states a number of useful properties of control dependence graphs (CDG). Section V proposes a parallel execution model of such a graph. The augmentation of the CDG with data dependences is considered in Section VI. Section VII develops a platform for the computation of the execution conditions of tasks based on data and control dependences. Section VIII presents a technique for the optimization of control conditions in detail. Finally, Section IX covers parallel source code generation issues.

II. Our Approach and Related Work

The motivations and goals for this work were set forward in [20] where the need for a hierarchical program representation and its use in generating auto-scheduling code were outlined. This paper presents new research results focusing on fundamental compiling aspects, while architectural implications are discussed in [21] which gives a refined description of auto-scheduling, in relation with the work discussed in this paper.

The goals of our work can be outlined as follows: Given any serial or parallel source program (written in C, Fortran, or other imperative languages) we wish to compile it into an intermediate representation which will encompass parallelism at all levels, and will serve as the structure upon which all optimizations will be performed; this structure will also be the starting point for emanating both, parallel source, and parallel machine code. Since loop-level parallelism is a subject well studied up to now, the thrust of our work is on other types of unstructured parallelism, including parallelism at the basic block level and within a single loop iteration, parallelism across procedures and loops, and in general all types of unstructured parallelism. It provides the means necessary for parallelism extraction and it is orthogonal to other known approaches for loop-level parallelism. Moreover, emphasis was put on the hierarchical aspect of the task graph. The reasons were twofold: control over task granularity, and the ability to generate highly parallel code for different architectures and execution models. By preserving or exposing the hierarchical nature of computations and programs in an intermediate representation, one can control the granularity of generated tasks [21]. In addition, this hierarchy greatly facilitates other optimizations and memory allocation. However, quantitative measures regarding the amount of parallelism or the performance improvement that would result from extracting and exploiting parallelism at that level are out of the scope of this paper, and are the current focus of our work.

The loop structure of a compiled program is used to construct the hierarchical task graph, thus capturing the hierarchy at the loop level. Each level of the hierarchy is processed separately in deriving control and data dependences, constructing the *HTG* as a layered graph with layers corresponding to loop or subroutine bodies. Parallelism at the loop level is assumed to be extracted using existing schemes [4], [9], [16], [24]. Therefore, in what follows, we ignore issues pertaining to parallelism across loop iterations. This implies that whenever data dependences are considered in the rest of the paper, they are assumed to be nonloop carried dependences.

Our work builds on the notion of control dependence as defined in [3] and [6] (for the purpose of masked vector instructions), and its generalization as defined in [11] and [12]. The program dependence graph and its versions, as defined in [11] and [12], have proven to be powerful intermediate representations for optimization and source code generation. Our work differs from that in [10] and [11] in the following ways: Unlike in [10] where the (implicit) hierarchy is based on intervals [1], our hiearchy is based on loop structures; this would yield different context for each level of the hierarchy. Another major difference is the inclusion of data dependences; [11] consider the use of synchronization to enforce data dependences only for identically control dependent tasks. Our approach is general in that synchronization is not restricted between identically control dependent tasks. The result is (potentially) more parallelism. Hence, the similarity of our approach and that of [11] lies only in the common use of control dependences as defined in [12]; however, our approach differs in all other aspects, including the way source code is generated.

We proceed by formalizing the notion of execution conditions based on data and control dependences (similar to [15]), and by tackling the problem of optimizing such conditions. In this paper we propose efficient algorithms for optimizing execution conditions, and consider implementation issues. Finally, the HTG and the execution conditions associated with task nodes are used as the basis for implementing an auto-scheduling environment, under which a program schedules and manages its tasks during its execution [21].

In [10] static partitioning is used whereby tasks in the program dependence graph can be merged to reflect execution costs on various architectures. Our approach is quite opposite in that determination of task granularity is performed dynamically, accommodating thus not only different architectures, but also potential reconfigurations of the same machine which may happen during program execution (e.g., due to multiprogramming). This is achieved by instrumenting the HTG with special code during the auto-scheduling phase; the details of source code generation for different architectures are given in [13], while intermediate code generation for granularity control is discussed in [18] and [21].

III. The Hierarchical Task Graph

The hierarchical task graph is a directed acyclic graph $HTG = (HV, HE)$ with unique nodes $START$ and $STOP$ belonging to HV such that there exists a path from $START$ to every node in HV and a path from every node to $STOP$; $START$ has no incoming arcs, and $STOP$ has no outgoing arcs. Each node in HV can be of one of the following types:

1) *simple* node representing a task that has no subtasks,
2) *compound* node representing a task that consists of other tasks in an *HTG*, or
3) *loop* node representing a task that is a loop whose iteration body is an *HTG*.

Reducible programs [1] can be easily mapped into a hierarchical structure based on loops by taking care of the following objections:

1) Loops that are not contained in each other and are not disjoint—In this case it can be shown that the loops must have a common header and we combine the loops into a common loop. This is done by creating an extra node and having all back arcs go to the new node instead of back to the header. We now add a single back arc from the new node to the header [Fig. 1(a)].
2) Loops with many exits—This violates the constraint that there can be at most one *STOP* node for the iteration body of a loop. This is solved by creating an extra node and forcing all exits out of a loop to go through this node [Fig. 1(b)]. This can be done by the addition of extra variables to "remember" which exit to follow from the new node.

The loop hierarchy is different from [10], where the hierarchical structure of a program is generated from intervals [1]. Fig. 2 illustrates a simple example where the loop and interval hierarchies are different.

Fig. 3(b) illustrates the *hierarchical task graph* of the program fragment in Fig. 3(a). 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. 3(b) is a three-level hierarchical flow 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. Given the definition of the hierarchical loop structure of a program, the remainder of this paper focuses on the construction of the *HTG* at a particular hierarchy level through data and control dependence analysis. This process would be repeated for each level of the hierarchy.

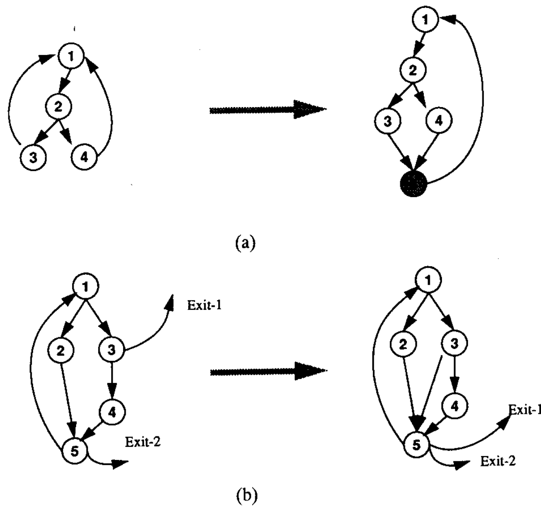


Fig. 1. Solving problems with loops to get a hierarchy.

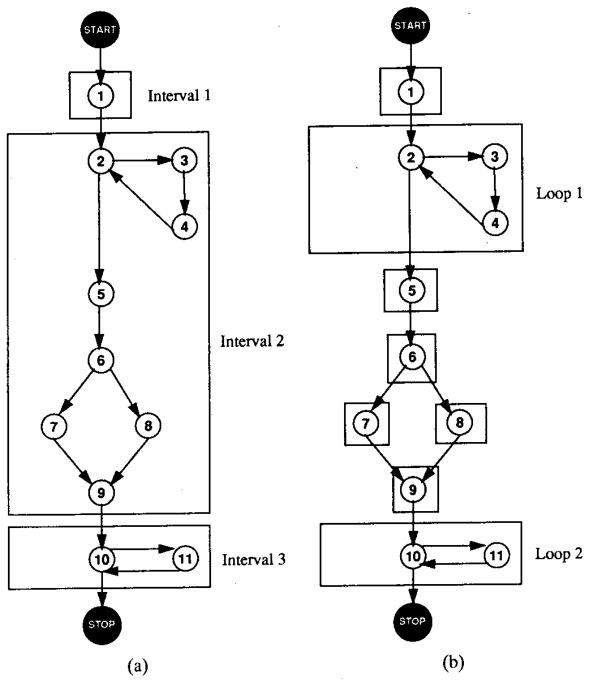


Fig. 2. Building hierarchies with intervals and loops.

IV. The Control Dependence Graph (CDG)

A *control flow graph* is a directed graph $CFG = (V, E)$ 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. The HTG is an acyclic control flow graph at any level, where each node represents either a simple node, compound node, or a loop. Although we shall be dealing mainly with acyclic control flow graphs, we relax this restriction in this section to state more general properties of the control dependence graph, explicitly mentioning CFG to be acyclic whenever needed.

Node x *dominates* node y, denoted by $x\Delta_d y$, iff every path from $START$ to y contains x [1]. A node always dominates itself. We use $x\not\Delta_d y$ to denote x does not dominate y.

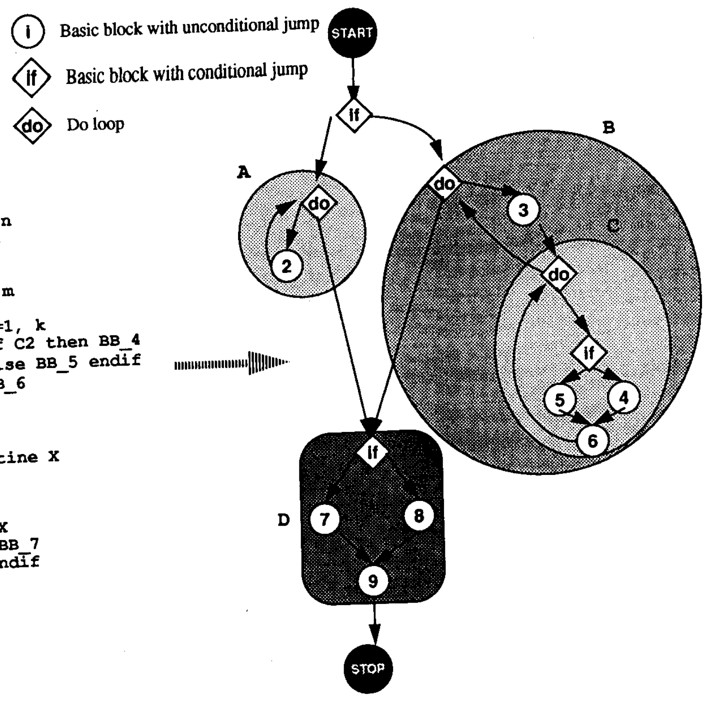
Node y *post-dominates* node x, denoted by $y\Delta_p x$, iff every path from x to $STOP$ (not including x) contains y [12]. A node never post-dominates itself. We use $y\not\Delta_p x$ to denote y does not post-dominate x. The *reflexive closure* of the post-dominance relation will be denoted by $\bar{\Delta}_p$, $y\bar{\Delta}_p x$ iff $y\Delta_p x$ or $y = x$. The following is well known [12].

Lemma 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 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

```
begin
BB_1
if C1 then
     DO i=1, n
          BB_2
     endo
else
     do i=1, m
          BB_3
          do j=1, k
               if C2 then BB_4
               else BB_5 endif
               BB_6
          endo
     endo
endif
call Subroutine X
end

subroutine X
if C3 then BB_7
else BB_8 endif
BB_9
```



Basic block with unconditional jump
Basic block with conditional jump
Do loop
START
If
A
B
C
D
do
2
3
do
If
5
4
6
If
7
8
9
STOP

(a) (b)

Fig. 3 Hierarchical task graph. (a) Program fragment. (b) The hierarchical task graph of (a).

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.*

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 nonnull path $P = < x, a, \cdots, y >$, such that for any $z \in P$ (excluding x and y) $y\Delta_p z$.

Our definition of control dependence differs only slightly from [12] 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 $y\bar{\Delta}_p a$.

The *control dependence graph CDG*, of a control flow graph CFG, is defined as the directed graph with labeled arcs, $CDG = (CV, CE)$ 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 [12] as follows. If (x, y) is any branch in CFG, then it can be shown that

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, $LCA(x, y)$, 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 $x - y$.

The *transitive closure* of δ_c will be denoted by δ_c^*, $x\delta_c^* y$ iff there exists a nonnull path from x to y in CDG. The reflexive closure of δ_c^* will be denoted by $\bar{\delta}_c^*$, $x\bar{\delta}_c^* y$ iff $x\delta_c^* y$ or $x = y$.

The remaining part of this section states a number of useful properties of the CDG which are directly or indirectly used in later optimization phases. However, the reader can safely skip the remaining of this section without loss of algorithmic/procedural details. Proofs are omitted for brevity and can be found in [14]. Some of the proofs have appeared elsewhere before, we cite a reference wherever appropriate. Lemma 2 characterizes the relation δ_c^* in terms of paths in CFG, δ_c^* corresponds to the notion of the range of a branch given in [23].

Lemma 2: $x\delta_c^* y$ iff there exists a nonnull path, P, in CFG from x to y such that for any z in P, $z \not\Delta_p x$.

Lemma 2 is useful in proving various properties of the CDG. Lemma 3 [11] states that if node x post-dominates node y, then x also post-dominates any descendant of y in the CDG.

Lemma 3: If $x\Delta_p y$ and $y\bar{\delta}_c^* z$, then $x\Delta_p z$.

Lemma 4: If $x\delta_c^* y$, $z\Delta_p y$, then either $z\Delta_p x$ or $x\delta_c^* z$.

Lemma 2 and Lemma 4 can be used to derive the following important theorem.

Theorem 1: CDG is cyclic iff CFG is cyclic.

Lemma 5 [11] determines when two nodes in the CDG may not share common descendants when CFG is acyclic.

Lemma 5: Let CFG be acyclic. If $x\Delta_p y$ (or $y\Delta_p x$), then there is no node z such that both $x\bar{\delta}_c^* z$ and $y\bar{\delta}_c^* z$ hold.

Lemma 6 characterizes the paths in CFG when CFG is acyclic.

Lemma 6: Let CFG be acyclic. If there is a path from x to y in CFG, then there exists a unique node z such that $z\bar{\Delta}_p x$ and $z\bar{\delta}_c^* y$.

Lemma 7 characterizes nodes which post-dominate $START$ in CFG.

Lemma 7: $x\bar{\Delta}_p START$ iff for all a such that $a\delta_c x$, $x\Delta_d a$.

When CFG is acyclic, Lemma 7 yields Corollary 1.

Corollary 1: Let CFG be acyclic. Then $x\bar{\Delta}_p START$ iff x has no incoming arcs in the CDG.

V. Parallel Execution of Acyclic CFG

In the absence of data dependences, the parallel execution of CFG is based on CDG [11] 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, say 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 1, CDG is acyclic and hence it is obvious that the parallel execution will terminate.

Let $\mathcal{S}$ and $\mathcal{P}$ denote the *sequential* and *parallel* execution of CFG, respectively. $\mathcal{S}$ specifies a single path P in CFG from $START$ to $STOP$. The parallel execution of CFG specifies a tree in the CDG [11].

A node is executed in $\mathcal{S}$ if it lies on P. A label $x{-}y$ will be *true* in $\mathcal{P}$ if the arc $x{-}y$ lies on P. According to the above model, a node x is executed in $\mathcal{P}$ when there is a path in the CDG, $< a_0, a_1, \cdots, a_n = x >$ such that a_0 is a node with no incoming arcs and $a_j\delta_c a_{j+1}$ $(0 \le j < n)$ with some true label $a_j{-}b_j$. Lemma 8 states a property of nodes which execute in $\mathcal{P}$, which is used repeatedly herein. If a and b execute, then either b will lie on the path in the CDG that led to the execution of a or vice versa; if neither is the case, then a and b cannot share a common descendant in the CDG.

Lemma 8: Let CFG be acyclic and let nodes a and b both execute in $\mathcal{P}$ with corresponding paths in the CDG, $< a_0, a_1, \cdots, a_n = a >$ and $< b_0, b_1, \cdots, b_m = b >$ such that a_0 (b_0) is a node with no incoming arcs and $a_j\delta_c a_{j+1}$ ($b_j\delta_c b_{j+1}$) for $0 \le j < n$ ($0 \le j < m$) with true label $a_j - c_j$ ($b_j - d_j$). Then one of the following statements is true.

1) $n \le m$ and the paths agree on the first n elements; i.e., $a_i = b_i$ for $0 \le i \le n$.
2) $m \le n$ and the paths agree on the first m elements; i.e., $b_i = a_i$ for $0 \le i \le m$.
3) a and b have no common descendant in CDG; i.e., $\nexists$ z such that $a\bar{\delta}_c^* z$ and $b\bar{\delta}_c^* z$.

Proof: Assume that the first two statements are false. Then there exists a j such that

1) $0 \le j \le \min(m, n)$,
2) $a_j \ne b_j$ and
3) $a_l = b_l$ for $0 \le l < j - 1$.

If $j = 0$, since a_0 and b_0 have no incoming arcs, it follows from Corollary 1 that $a_0\bar{\Delta}_p START$ and $b_0\bar{\Delta}_p START$. By Lemma 1 either $a_0\Delta_p b_0$, or $b_0\Delta_p a_0$. If $j \ne 0$, then $a_{j-1} = b_{j-1}$ and $a_{j-1}\delta_c a_j$ and $a_{j-1}\delta_c b_j$. Further, they must have the same label as only one label can be true from a_{j-1}, and both a_j and b_j were executed in $\mathcal{P}$. Hence, $a_{j-1} - c_{j-1} = b_{j-1} - d_{j-1}$. By definition of control dependence, $a_j\bar{\Delta}_p c_{j-1}$ and $b_j\bar{\Delta}_p c_{j-1}$ and by Lemma 1 either $a_j\Delta_p b_j$ or $b_j\Delta_p a_j$. Thus, we have proved that either $a_j\Delta_p b_j$ or $b_j\Delta_p a_j$. Clearly, $a_j\bar{\delta}_c^* a$ and $b_j\bar{\delta}_c^* b$. If the third statement is also false, then there exists a z such that $a\bar{\delta}_c^* z$ and $b\bar{\delta}_c^* z$. Using transitivity we get $a_j\bar{\delta}_c^* z$ and $b_j\bar{\delta}_c^* z$. This contradicts Lemma 5. □

The following theorem states the correctness of the parallel execution.

Theorem 2: Let CFG be acyclic. The parallel execution of CFG executes the same nodes as the sequential execution. (A proof is given in [14].)

VI. The Data Dependence Graph (DDG)

Node y *conflicts* with node x if either x or y share access to a common variable, at least one of which is a "write" operation. Conflicts induce a data dependence [3], [4], [7], [17], [24] relation among nodes. Exactly one of the following can occur between two distinct nodes x and y.

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

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 as x and y will not be executed together in any execution instance of CFG and can be ignored. If CFG is the iteration body of a loop, we are restricting ourselves to loop-independent dependences [3], [4], [24].

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

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

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.

VII. Conditions for Execution of Task Nodes

With the addition of data dependences, when a node is to be executed, it must be verified whether the nodes on which it is data dependent have completed execution or are not going to be executed; in both cases the data dependences are satisfied. This can be done by defining conditions for each node so that the condition evaluates to true only when the node is ready; i.e., it must be executed and the data dependences, if any, be satisfied.

Conditions contain *literals* representing nodes (x) or arcs ($x{-}y$) in CFG. The condition (x) will be true when node x has finished execution. The condition ($x{-}y$) will be true when

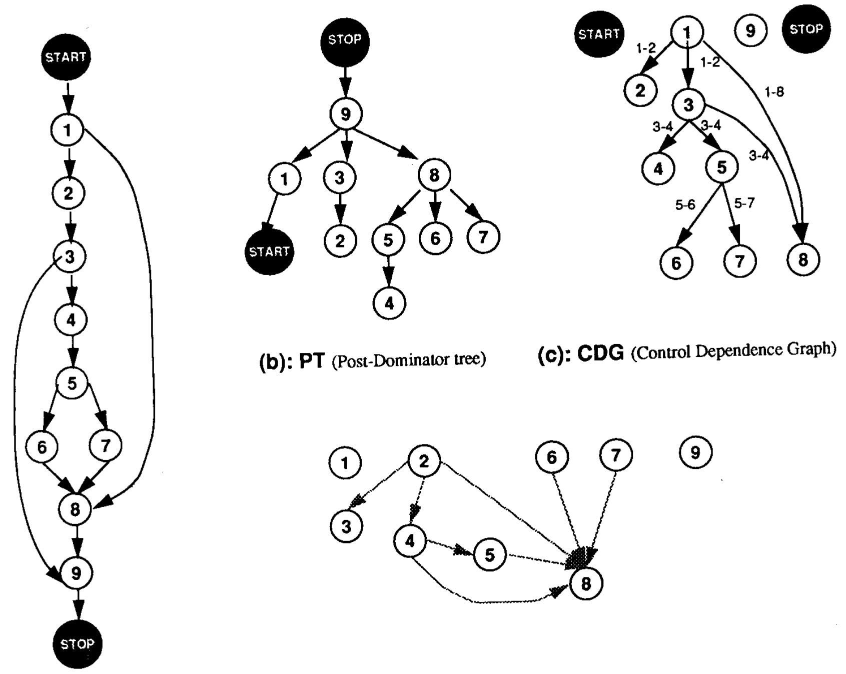

START
STOP
(b): PT (Post-Dominator tree)
(c): CDG (Control Dependence Graph)
1-2
1-8
3-4
5-6
5-7

Fig. 4. An example. (a) CGF (Control Flow Graph). (b) PT (Post-Dominator tree). (c) CDG (Control Dependence Graph). (d) DDG (Data Dependence Graph).

node x finishes execution and in addition control follows the arc $x-y$ in CFG. A node x will be ready to execute when:

1) The control conditions which force the execution of x are true.
2) If $y\delta_d x$, then either y has finished execution or it is known that y will not be executed.

An example will make this clear. Consider the control flow graph shown in Fig. 4(a). Its post dominator tree and control dependence graph are shown in Fig. 4(b) and (c). Let the data dependence graph be as shown in Fig. 4(d). Since we will be dealing with acyclic graphs, we will assume from now on that the nodes are numbered in such a way that if there is an arc (x, y) in the CFG, $x < y$.

The conditions for node 5 to be executed are that the branch from node 3 to node 4 is taken (the necessary control condition), and either node 4 has finished execution or it is determined that it will not be executed at all (the necessary data dependence condition). Node 4 will not be executed when the branch 3–9 or 1–8 is taken. Thus, the condition for the execution of node 5 can be compactly represented by $3\text{–}4 \wedge (4 \vee 3\text{–}9 \vee 1\text{–}8)$. When nodes 3 and 4 finish execution, they will try to update the condition of node 5 (provided it has not been executed yet), and if the update causes the condition to be true, node 5 can be executed. We now formally define the procedure for deriving the conditions for a node.

A. Control Dependence Conditions

The control conditions for a node x are easily derived from the CDG. Let x in the CDG be control dependent on $a_1, a_2, \cdots, a_n$ with labels $a_1\text{–}b_1, a_2\text{–}b_2, \cdots, a_n\text{–}b_n$, respectively. Then the condition for x is

$$a_1\text{–}b_1 \vee a_2\text{–}b_2 \vee \cdots \vee a_n\text{–}b_n.$$

B. Data Dependence Conditions

For data dependence conditions we need to know when a node will not be executed. It is easier to do the reverse computation; that is find the nodes that will not be executed if a branch is taken in the flow graph. We first define $REAC(x)$ to be the set of nodes reachable from a node x in the CDG.

$$REAC(x) = \{y | x\delta_c^* y\}.$$

This can be done by a simple depth-first traversal of the CDG. The sets $REAC(x)$[1] were also defined and used in [11]. Thus, for the example control dependence graph shown in Fig. 4(c), we have $REAC(3) = \{4, 5, 6, 7, 8\}$, $REAC(1) = \{2, 3, 4, 5, 6, 7, 8\}$, $REAC(5) = \{6, 7\}$ and for all other nodes, x, $REAC(x) = \emptyset$. Similarly, we define the set of nodes reachable from a branch in the CDG, denoted by $REAC(x\text{–}y)$.

$$REAC(x\text{–}y) = \{a | \exists z \text{ such that } x\delta_c z \text{with label } x\text{–}y \text{ and } z\bar{\delta}_c^* a\}.$$

Obviously, $REAC(x\text{–}y) \subseteq REAC(x), \forall x$. In our example we get $REAC(1\text{–}2) = \{2, 3, 4, 5, 6, 7, 8\}$, $REAC(1\text{–}8 = \{8\}$, $REAC(3\text{–}4) = \{4, 5, 6, 7, 8\}$, $REAC(3\text{–}9) = \emptyset$, $REAC(5\text{–}6) = \{6\}$ and $REAC(5\text{–}7) = \{7\}$. Based on these definitions we can define $BranNeg(x\text{–}y)$ for a branch $x\text{–}y$ in the CFG.

$$BranNeg(x\text{–}y) = REAC(x) - REAC(x\text{–}y).$$

[1]In [11] $REAC(x)$ included x also.

In the process we also define the set $Neg(x)$ for each node x. $Neg(x)$ is the set of all branches in CFG whose traversal bypasses the execution of x (a formal proof is given in Lemmas 9 and 10).

$$Neg(x) = \{y\text{–}z \mid x \in BranNeg(y\text{–}z)\}.$$

or equivalently,

$$Neg(x) = \{y\text{–}z \mid x \in REAC(y) \text{ and } x \notin REAC(y\text{–}z)\}.$$

In the example we have $BranNeg(1\text{–}2) = BranNeg(3\text{–}4) = \emptyset$, $BranNeg(1\text{–}8) = \{2,3,4,5,6,7\}$, $BranNeg(3\text{–}9) = \{4,5,6,7,8\}$, $BranNeg(5\text{–}6) = \{7\}$ and $BranNeg(5\text{–}7) = \{6\}$. From these we can compute $Neg(START) = Neg(1) = Neg(9) = Neg(STOP) = \emptyset$, $Neg(2) = Neg(3) = \{1\text{–}8\}$, $Neg(4) = Neg(5) = \{1\text{–}8, 3\text{–}9\}$, $Neg(6) = \{1\text{–}8, 3\text{–}9, 5\text{–}7\}$, $Neg(7) = \{1\text{–}8, 3\text{–}9, 5\text{–}6\}$ and $Neg(8) = \{3\text{–}9\}$.

We now prove some of the properties of the Neg sets.

Lemma 9: Let $p\text{–}q \in Neg(r)$. If label $p\text{–}q$ is true then r will not be executed.

Proof: If r executes, then there exists a path, $P_1 = \langle a_0, \cdots, a_n = r\rangle$ in the CDG such that a_0 has no incoming arcs and $a_i \delta_c a_{i+1}$ with true label $a_i\text{–}c_i$ for $0 \le i < n$. Since label $p\text{–}q$ is true, p has executed and there exists a path $P_2 = \langle b_0, \cdots, b_m = p\rangle$ in the CDG such that b_0 has no incoming arcs and $b_i \delta_c b_{i+1}$ with true label $b_i\text{–}d_i$ for $0 \le i < m$. We now apply Lemma 8 (substitute $a = r$, $b = p$) and get that one of the following is true.

1) $n \le m$ and $a_i = b_i$ for $0 \le i \le n$.
2) $m \le n$ and $b_i = a_i$ for $0 \le i \le m$.
3) $\nexists$ a z such that $r\bar{\delta}_c^* z$ and $p\bar{\delta}_c^* z$.

Since $p - q \in Neg(r)$, $r \in REAC(p)$ and $r \notin REAC(p\text{–}q)$. $r \in REAC(p)$ implies $p\delta_c^* r$ and since $r\bar{\delta}_c^* r$, clearly statement 3 is false as we can choose $z = r$. Statement 1 cannot be true because that would imply $r\bar{\delta}_c^* p$ (r would lie on P_2) contradicting the acyclicity of CFG as we know that $p\delta_c^* r$. If statement 2 is true, then $p = a_m$ (p would lie on P_1) and since the label $p\text{–}q$ is true, $p\delta_c a_{m+1}$ with label $p\text{–}q$. Since $a_{m+1}\bar{\delta}_c^* r$, $r \in REAC(p\text{–}q)$ giving a contradiction. Hence, none of the three statements is true proving that r will not be executed. □

Lemma 10: If r is not executed then exactly one of the labels in $Neg(r)$ will be true.

Proof: We first prove that at least one of the labels in $Neg(r)$ will be true. Suppose that none of the labels in $Neg(r)$ is true. We define sets R_i, $x \in R_i$ iff $x\bar{\delta}_c^* r$ and the maximum length of a path in CDG from x to r is i. The sets R_i correspond to ancestors of r in the CDG arranged in layers. Clearly, there exists an n such that $R_i = \emptyset$ for $i > n$ and $R_n \ne \emptyset$. It is easy to see the following:

1) $R_0 = \{r\}$,
2) If $x \in R_i$ for $i > 0$ then $x\delta_c^* r$ and hence $r \in REAC(x)$ and
3) If $x \in R_n$ then x has no predecessors in CDG.

We prove that no nodes in R_i for $0 \le i \le n$ will be executed. The proof is by induction on i.

Basis ($i = 0$): $R_0 = \{r\}$ and by hypothesis r is not executed.

Induction step: Assume that none of the nodes in $R_0, \cdots, R_{i-1}$ is executed. We will show that no node in R_i ($i > 0$) is executed. Suppose that $x \in R_i$ and x is executed. Then one of the branches from x has label true, let that branch be $x\text{–}y$. If $r \notin REAC(x\text{–}y)$, then since $r \in REAC(x)$, $x\text{–}y \in Neg(r)$ which contradicts our supposition that none of the labels in $Neg(r)$ is true. If $r \in REAC(x\text{–}y)$, then there exists a z such that $x\delta_c z$ with label $x\text{–}y$ and $z\bar{\delta}_c^* r$ and z is executed as label $x\text{–}y$ is true. The maximum length of a path from z to r is at least one less than i, hence $z \in R_j$ for some j, $0 \le j < i$. This is a contradiction because we assumed that none of the nodes in R_j is executed.

Since $R_n \ne \emptyset$, let $x \in R_n$. We just proved that if r is not executed and none of the labels in $Neg(r)$ is true, then x will not be executed. However, x has no predecessors in CDG and hence is always executed. This is a contradiction. Hence, at least one of the labels in $Neg(r)$ is always true.

Now we show that exactly one of the labels in $Neg(r)$ is true. Suppose that labels $s\text{–}t$ and $u\text{–}v$ belonging to $Neg(r)$ are true. Clearly, $s \ne u$ as only one label can be true from one node. Also, since $s\text{–}t \in Neg(r)$, $r \in REAC(s)$ and $r \notin REAC(s\text{–}t)$. Similarly, $r \in REAC(u)$ and $r \notin REAC(u\text{–}v)$. If labels $s\text{–}t$ and $u\text{–}v$ are true then s and u are executed and there are corresponding paths in the CDG, $< a_0, a_1, \cdots, a_n = s >$ and $< b_0, b_1, \cdots, b_m = u >$ such that a_0 (b_0) is a node with no incoming arcs and $a_j \delta_c a_{j+1}$ ($b_j \delta_c b_{j+1}$) for $0 \le j < n$ ($0 \le j < m$) with true label $a_j\text{–}c_j$ ($b_j\text{–}d_j$). We apply Lemma 8 (substitute $a = s$, $b = u$) and get that one of the following is true.

1) $n \le m$ and $a_i = b_i$ for $0 \le i \le n$.
2) $m \le n$ and $b_i = a_i$ for $0 \le i \le m$.
3) $\nexists$ a z such that $s\bar{\delta}_c^* z$ and $b\bar{\delta}_c^* z$.

Since $r \in REAC(s)$ and $r \in REAC(u)$, $s\delta_c^* r$ and $u\delta_c^* r$, hence statement 3 is false as we can choose $z = r$. If statement 1 is true, then $s = b_n$ and $n < m$ as $s \ne u$. Since $s\text{–}t$ is the only true label from s, $s\delta_c b_{n+1}$ with label $s\text{–}t$. Since $b_{n+1}\bar{\delta}_c^* u$ and $u\delta_c^* r$, $b_{n+1}\bar{\delta}_c^* r$. This gives $r \in REAC(s\text{–}t)$ which is a contradiction. We get a similar contradiction if statement 2 is true. Hence, exactly one of the labels in $Neg(r)$ is true. □

We can now define the data dependence condition for a node x when it is dependent on another node y. Let $Neg(y)$ be $\{a_1\text{–}b_1, a_2\text{–}b_2, \cdots, a_n\text{–}b_n\}$. Then the data dependence condition is

$$y \lor a_1\text{–}b_1 \lor a_2\text{–}b_2 \lor \cdots \lor a_n\text{–}b_n.$$

If x is data dependent on other nodes, then we take a conjunction of all conditions. The conditions for all the nodes in the example flow graph of Fig. 4 are shown in Table I. A blank entry indicates that the condition is always true.

VIII. Optimization of Data and Control Dependences

Since the conditions for a node will be repeatedly updated and evaluated for satisfaction, it is important that they be as simple as possible. For example, some of the data dependences need not be synchronized by way of execution conditions because they are always satisfied by other control and data dependences which have been enforced. In such cases the

TABLE I
UNOPTIMIZED CONDITIONS FOR ALL NODES IN FIG. 4

Node	Condition
START	-
1	-
2	1-2
3	1-2 ∧ (2 ∨ 1-8)
4	3-4 ∧ (2 ∨ 1-8)
5	3-4 ∧ (4 ∨ 1-8 ∨ 3-9)
6	5-6
7	5-7
8	(1-8 ∨ 3-4) ∧ (2 ∨ 1-8) ∧ (4 ∨ 1-8 ∨ 3-9) ∧ (5 ∨ 1-8 ∨ 3-9) ∧ (6 ∨ 5-7 ∨ 1-8 ∨ 3-9) ∧ (7 ∨ 5-6 ∨ 1-8 ∨ 3-9)
9	-
STOP	-

data dependence term in the condition can be omitted. In this section we show how optimizations can be performed to simplify the conditions. Optimizations are done in two phases:

Phase I: Elimination of redundant dependences.

Phase II: Simplification of execution conditions.

A. Elimination of Redundant Dependences

Let $x \prec y$ (x *precedes* y) denote that x finishes before y in all parallel executions of the *CFG* in which both x and y execute. The following lemma will be useful in the optimization phase.

Lemma 11: If $x\delta_c^* y$ then $x \prec y$.

Proof: If both x and y are executed in an instance, $\mathcal{P}$, then there are two paths in CDG, $P_1 = \langle a_0, a_1, \cdots, a_n = x\rangle$ and $P_2 = \langle b_0, b_1, \cdots, b_m = y\rangle$ such that a_0 and b_0 have no incoming arcs and $a_j\delta_c a_{j+1}$ ($b_j\delta_c b_{j+1}$) with true label a_j-c_j (b_j-d_j) for all $j = 0, \cdots, n-1$ ($j = 0, \cdots, m-1$). We apply Lemma 8 and hence one of the following is true.

1) $n \le m$ and $a_i = b_i$ for $0 \le i \le n$.
2) $m \le n$ and $b_i = a_i$ for $0 \le i \le m$.
3) $\nexists$ a z such that $x\bar{\delta}_c^* z$ and $y\bar{\delta}_c^* z$.

Clearly, statement 3 is not true as we can choose $z = y$ and get $x\bar{\delta}_c^* y$ and $y\bar{\delta}_c^* y$. Statement 2 is false as it would imply $y = b_m = a_m\bar{\delta}_c^* x$ which along with $x\delta_c^* y$ contradicts acyclicity of *CFG*. Hence, statement 1 is true and we have $x = a_n = b_n$ and x lies on the execution path to y and therefore finishes execution before y begins execution. □

It is clear that if we can determine that $x \prec y$ then the data dependence $x\delta_d y$ (if present) is satisfied and does not need explicit synchronization. A general outline of our algorithm is given in Fig. 5. At any stage, we have information (the set S) for a subset of nodes determining which nodes precede other nodes, due to data dependences which have been enforced and control dependences. Whenever a new node (x) is added, this information is updated [step 3(a)] by checking if (y, x) can be added to S for all $y < x$. When we consider the data dependences incident on the new node [step 3(b)], we check if they are implied by previous data and control dependences or are really necessary. In the former case, we can ignore the dependence [step 3(b)i]. In the latter case, we must enforce the dependence (by synchronization), which may result in determining the execution order of other nodes, and hence may lead to an update in our information [step 3(b)ii].

We illustrate the method on the example control flow graph in Fig. 4. The working of the algorithm at various stages is shown in Fig. 6. The information available in set S is shown in Fig. 6 by two sets associated with each node x shown by square (where x is the first component) and round brackets (where x is the second component). Looking at the sets associated with node 3 in Fig. 6(a), one can see that the pairs (1,3), (2,3), (3,4), (3,5), (3,6), (3,7), (3,8) belong to S. Initially, S consists of pairs (x, y) where $x\delta_c^* y$. The **for** loops in Steps 3 and 3(b) imply that data dependence (a, b) will be considered before (c, d) if $b < d$ or if $b = d$ and $a > c$. For our example graph in Fig. 4(d), this order is (2,3), (2,4), (2,5), (7,8), (6,8), (5,8), (4,8), (2,8).

When dependence $2\delta_d 3$ is considered in step 3(b) it will have to be synchronized as $2 \not\prec 3$. However, after that is done, (2,3) will be added to S (step 3(b)ii). When determining whether $2 \prec 4$ in step 3(a) ($y = 2, x = 4$), the set A will evaluate to $\{3\}$ and since $(2,3) \in S$, (2,4) will also be added to S (step 3(a)iii). This causes the data dependence $2\delta_d 4$ to be considered redundant in step 3(b)i [see Fig. 6(a)].

The status of the algorithm when considering the question whether $(2,8) \in S$ (step 3(a), $y = 2, x = 8$) is shown in Fig. 6(b). The unique z is found to be 3 and hence A will again evaluate to $\{3\}$. Note that the set A does not consist of the predecessors of 8 in the control dependence graph (e.g., $1 \notin A$); instead it is the set of predecessors of 8 which can lie on a path from 2 to 8. Thus, (2,8) will be added to S. Subsequently, this information will be used to eliminate the dependence $2\delta_d 8$.

Fig. 6(c) shows the status when it has been determined that $7\delta_d 8$ needs to be enforced (step 3(b)ii). Enforcing new dependences may cause updates in current information. In this case because the dependence $7\delta_d 8$ was enforced, the pair (5,8) will be added to S (step 3(b)iiA). This in turn will cause (4,8) to be added to S because of the data dependence $4\delta_d 5$ (step 3(b)iiB). This propagation of information is shown in Fig. 7. This information will be used later to show that the dependences $5\delta_d 8$ and $4\delta_d 8$ are redundant and need not be enforced.

For our example graph of Fig. 4 the algorithm will determine that the dependences $2\delta_d 4$, $5\delta_d 8$, $4\delta_d 8$, and $2\delta_d 8$ are redundant.

B. Further Simplification (Control Dependence "Elimination")

The algorithm in Section VIII-A will inform us of the essential dependences which need to be satisfied before a node can commence execution. It may still be possible to simplify the condition further. For example, in a term due to dependence $y\delta_d x$, some of the literals in $Neg(y)$ can be omitted if they also belong to $Neg(x)$ as they will not result in the execution of x. Under certain conditions the simplification process can be very powerful yielding consistently "better" conditions as is illustrated by the following proposition which allows for the simultaneous removal of terms from the control/data conditions of x.

```
Data Dependence Optimization Algorithm
1) Number the nodes of $CFG$ such that if there is an arc $(x, y)$ then $x < y$.
2) Construct set $S$ to consist of pairs $(x, y)$ such that $x \prec y$. Initially, if $x\delta_c^* y$ then $(x, y) \in S$ ( see Lemma 11).
3) **for** each node $x$ in increasing order **do**
    a.  **for** each node $y$ from $x - 1$ **downto** 1 **do** /* check if $(y, x)$ can be added to $S$ */
        i.   Find unique $z$ such that $z\overline{\Delta}_p y$ and $z\overline{\delta}_c^* x$ (see Lemma 6).
        ii.  Let $A$ be the set of nodes $a$ such that $a\delta_c x$ and $z\overline{\delta}_c^* a$.
        iii. If $(y, a) \in S$ for each $a$ in $A$, add $(y, x)$ to $S$.
    b.  **for** each data dependence $y\delta_d x$ in decreasing order of $y$ **do** /* Now consider dependences into $x$ and check which are redundant */
        i.   if $(y, x) \in S$ then $y\delta_d x$ is redundant.
        ii.  Otherwise, $y\delta_d x$ needs to be enforced. Add $(y, x)$ to $S$. Enforcing $y\delta_d x$ may cause additional pairs to be added to $S$. This update of
             $S$ is done by repeatedly executing the following two steps until no further updates can be done:
             A.  If $(a, x) \in S$ then add $(z, x)$ to $S$ for all such $z$ such that $z\delta_c^* a$.
             B.  For a node $z$ if $z\delta_d a$, $(a, x) \in S$, and either $a\Delta_p z$ or $a\Delta_d z$, then add $(z, x)$ to $S$.
```

Fig. 5. Algorithm to eliminate redundant dependences.

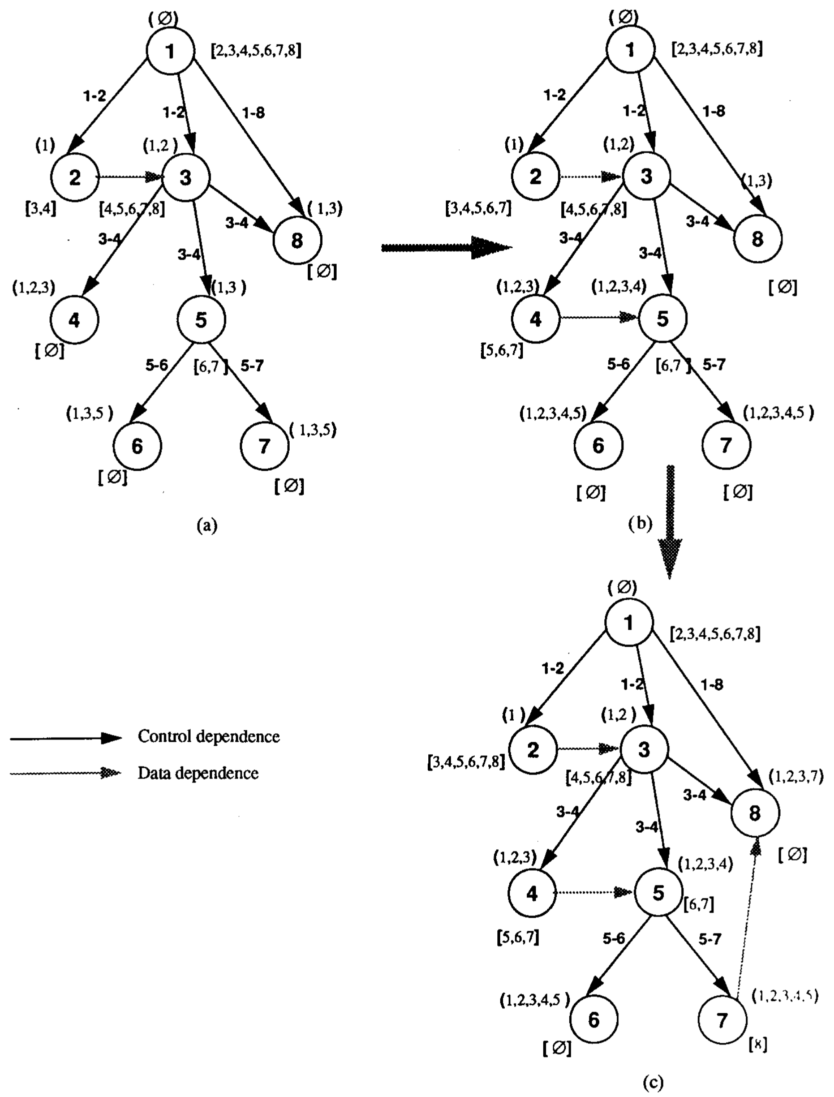


Fig. 6. Removing redundant data dependences. (a) Just before checking data dependence 2–4 precedes 4 and therefore dependence 2–4 is ignored. (b) Just before checking whether 2 precedes 8. All paths from 2 to 8 go through 3 and since 2 precedes 3 and 3 precedes 8, (2,8) will be added to S. (c) Just after enforcing data dependence 7–8. (7,8) is added to S.

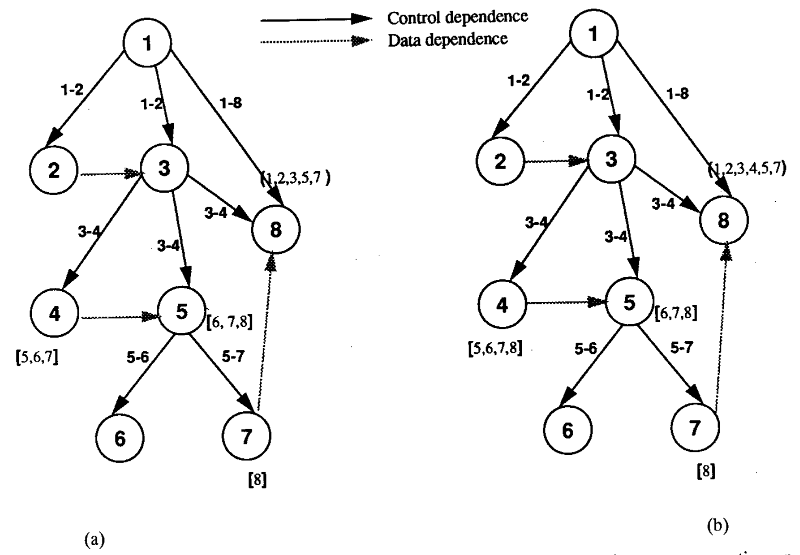


Fig. 7. Propagating information. (a) First step—propagation up to 5. (b) Second step—propagation up to 4.

Proposition 1: Let x be control dependent on $c_1,\cdots,c_m$ with labels $c_1-d_1,\cdots,c_m-d_m$, respectively, and let $Neg(y)=\{a_1-b_1,\cdots,a_n-b_n\}$. Then the condition for x is $(c_1-d_1 \vee \cdots \vee c_m-d_m) \wedge (y \vee a_1-b_1 \vee \cdots \vee a_n-b_n)$. Whenever $x\Delta_p y$, this condition can be replaced by the simpler condition $y \vee e_1-f_1 \cdots \vee e_p-f_p$, where each e_j-f_j is added as a result of some a_k-b_k as follows:

1) $e_j-f_j = a_k-b_k$ when $x\Delta_p a_k$.
2) $e_j-f_j = a_k-b_k$ when $a_k\delta_c x$ with label a_k-b_k.
3) $e_j-f_j = c_i-d_i$ for some $i \in 0\cdots m$ when $a_k\delta_c^* x$ and $c_i \in REAC(a_k-b_k)$.

Proof: ($\Rightarrow$) First, we prove that if the new condition is true, then x must execute and the dependence due to y is satisfied. For the condition to be true, one of the literals in the new condition has to be true. If the literal y is true, then y has finished execution, and since $x\Delta_p y$, x must execute.

If the literal e_j-f_j is true for some j, $1 \le j \le p$ then e_j-f_j is a result of exactly one of the cases mentioned above.

In Cases 1 and 2 the literal $e_j-f_j = a_k-b_k$ for some k, $1 \le k \le n$. Since $a_k-b_k \in Neg(y)$, y will not be executed (by Lemma 10). In Case 1, $x\Delta_p a_k$, and hence x must execute. In Case 2, x is control dependent on a_k with label a_k-b_k and hence x must execute.

In Case 3 the literal $e_j-f_j = c_i-d_i$ for some $i \in 0\cdots m$ where $a_k\delta_c^* x$ and $c_i \in REAC(a_k-b_k)$. Since $c_i\delta_c x$ with label c_i-d_i, it is clear that x must execute. It remains to show that the data dependence due to y is satisfied, this is done by showing that y will not execute.

Assume that y is executed. Since x is also executed under the control condition c_i-d_i there exist two paths in CDG, $P_1 = \langle g_0,\cdots,g_p = y\rangle$ and $P_2 = \langle h_0,\cdots,h_{q-1} = c_i, h_q = x\rangle$. Clearly x cannot lie on P_1 as $x\delta_c^* y$ and $x\Delta_p y$ would contradict the acyclicity of CFG. Also, y cannot lie on P_2, as $y\delta_c^* x$ contradicts $x\Delta_p y$. This implies that there exists an l such that

1) $0 \le l \le \min(p,q)$,
2) $g_l \ne h_l$, and
3) $g_r = h_r$ for all $0 \le r < l$.

Clearly, $h_l\bar{\delta}_c^* x$ and $g_l\bar{\delta}_c^* y$. Using a simlar argument as in the proof of Lemma 8, we can show that either $h_l\Delta_p g_l$ or $g_l\Delta_p h_l$.

If $g_l\Delta_p h_l$, then since $h_l\bar{\delta}_c^* x$, by Lemma 3, $g_l\Delta_p x$. Since $g_l\bar{\delta}_c^* y$, we get a path from x to y via g_l which along with $x\Delta_p y$ contradicts the acyclicity of CFG.

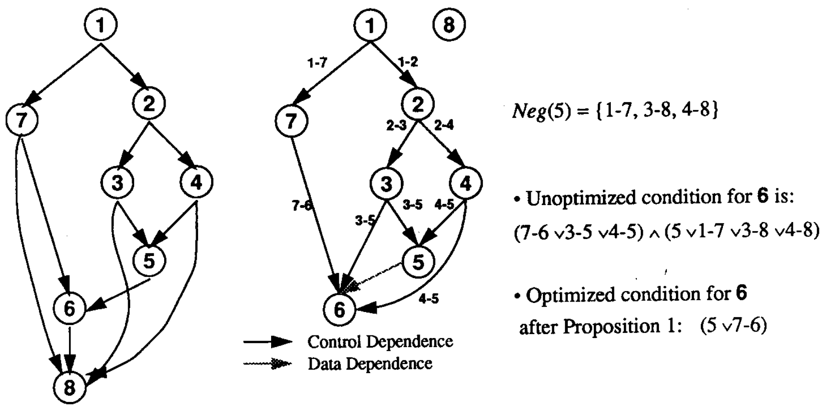
If $h_l\Delta_p g_l$, then since $g_l\bar{\delta}_c^* y$, by Lemma 3, $h_l\Delta_p y$. If x and h_l are distinct, then since $x\Delta_p y$, by Lemma 3, either $x\Delta_p h_l$, contradicting $h_l\bar{\delta}_c^* x$, or $h_l\Delta_p x$ which along with $h_l\bar{\delta}_c^* x$ contradicts the acyclicity of CFG. Hence, $x = h_l$. This gives $c_i = h_{l-1} = g_{l-1}\delta_c^* y$. Since $c_i \in REAC(a_k-b_k)$, $y \in REAC(a_k-b_k)$. This contradicts $y \in BranNeg(a_k-b_k)$.

In either case we get a contradiction, proving that y will not be executed.

($\Leftarrow$) Next we show that when x is ready to execute, the new condition will become true. When x is ready to execute, the data dependence due to y is satisfied and hence either y has finished execution or y will not be executed. If y has finished execution, then the literal y will become true and hence the new condition for x will be true. If y is not executed, then exactly one of the labels, a_k-b_k in $Neg(y)$ will be true for some k, $1 \le k \le n$ by Lemma 10. Since $a_k\delta_c^* y$ and $x\Delta_p y$, by Lemma 4 $x\Delta_p a_k$ or $a_k\delta_c^* x$. If $x\Delta_p a_k$ then a_k-b_k is in the new condition by Case 1, hence the new condition for x will become true. Hence, let $a_k\delta_c^* x$. Since the label a_k-b_k is true, a_k is also executed. Let the two paths in CDG leading to the execution of a_k and x be $\langle g_0,\cdots,g_p = a_k\rangle$ and $\langle h_0,\cdots,h_{q-1} = c_j, h_q = x\rangle$ where $c_j\delta_c x$ with true label c_j-d_j for some j, $1 \le j \le m$. We apply Lemma 8. Then one of the following is true.

1) $p \le q$ and $g_i = hi$ for $0 \le i \le p$.
2) $q \le p$ and $h_i = gi$ for $0 \le i \le q$.
3) $\nexists$ a z such that $a_k\bar{\delta}_c^* z$ and $x\bar{\delta}_c^* z$.

Clearly, Statement 3 is not true as we can choose $z = x$ and we have $a_k\delta_c^* x$ and $x\bar{\delta}_c^* x$. Statement 2 is not true because



(a) (b) (c)

Fig. 8. Further optimization of execution condition. (a) CFG. (b) CDG + DDG. (c) Optimization of execution condition.

that would imply $x = h_q = g_q\bar{\delta}_c^* a_k$ which along with $a_k\delta_c^* x$ contradicts acyclicity of *CFG*. Hence, Statement 1 is true. Since a_k and x are distinct, $p < q$ and $a_k = g_p = h_p$ and $a_k\delta_c h_{p+1}$ with label a_k–b_k. If $h_{p+1} = x$, then by Case 2, the literal a_k–b_k is in the new condition for x forcing it to be true. Otherwise, $h_{p+1}\bar{\delta}_c^* h_{q-1} = c_j$ and hence $c_j \in REAC(a_k$–$b_k)$. But then by Case 3, the literal c_j–d_j is in the new condition for x forcing it to be true. Thus, we see that when x is ready to execute, the new condition for x will become true. □

Although the proof of Proposition 1 is intricate, the actual computation can be done easily, as all that needs to be done is to find the immediate predecessors of x in *CFG* which are also in $REAC(a$–$b)$ for each a–$b \in Neg(y) - Neg(x)$. Consider the example control flow graph shown in Fig. 8. *Neg*(5) is {1–7, 3–8, 4–8} and *Neg*(6) is {7–8, 3–8, 4–8}. The condition for 6 is (7–6 ∨ 3–5 ∨ 4–5) ∧ (5 ∨ 1–7 ∨ 3–8 ∨ 4–8). Since $6\Delta_p 5$ we can use the above proposition to simplify the condition. The branch 1–7 falls under Case 3. It is now sufficient to look at predecessors of 6 which are descendants of 1–7. Thus, due to the arc 1–7 we include only 7–6 in the condition. Branch 3–8 also falls under Case 3; however, 6 has no predecessors which are also descendants of 3–8 and hence no literals are added to the condition for 6. This can also be seen from 3–8 being in *Neg*(6). Similarly, no new terms are added due to 4–8, and the final condition for 6 is evaluated to (5 ∨ 7–6).

The conditions for our example flow graph of Fig. 4 after elimination of redundant dependences and simplification are shown in Table II. The condition for 8 can be further simplified to (6 ∨ 7 ∨ 1–8), however, Proposition 1 can be applied to only a single data dependence at a time and hence this is not done. The complexity issues of the optimization algorithms are considered in [14].

IX. Parallel Source Code Generation

Let us consider the problem of parallel source code generation based on the execution conditions derived and optimized as discussed in the previous sections. We use the **cobegin** ⋯ **coend** parallel construct with its ordinary semantics, and

TABLE II
Optimized Conditions for all Nodes in Fig. 4

Node	Condition
START	–
1	–
2	1–2
3	2
4	3–4
5	4
6	5–6
7	5–7
8	(6 ∨ 5–7 ∨ 1–8) ∧ (7 ∨ 5–6 ∨ 1–8)
9	–
STOP	–

the synchronization primitives **wait, post, clear**. The synchronization primitives operate on *events*. The **wait**(a) statement induces a wait on the event "a" until a corresponding **post**(a) is done by some other task. A **clear**(a) clears all prior posts. Multiple posts with no wait or clear operations in between are equivalent to a single post. Fig. 9 shows the parallel code for our example. Curly brackets are used to group one or more program statements separated by semicolons. Such statements execute sequentially in the obvious lexicographic order.

Assuming all events have been cleared initially, code for any node has the following appearance.

1) **wait** (own event).
2) Do own work.
3) Update the conditions which are dependent on it and if any evaluate to true, then do the corresponding posts.

We assume that the updates to the condition for any node are done through a critical section as different nodes could be updating the condition for a node simultaneously.

The starting point for code generation is the *CDG*, and identically control dependent nodes are executed in parallel barring data dependences. There are two kinds of optimizations which can be done immediately.

1) If it is known at compile time that an update is going to change a condition to evaluate to true, one can

```
for all task_semaphore do
  clear(sem(task_semaphore))
endfor;
cobegin
{ 1;
  if 1-2 then
    cobegin
    { 2; post(sem(3)) }
    { wait(sem(3)); 3;
      if 3-4 then
        cobegin
        { 4; post(sem(5)) }
        { wait(sem(5)); 5;
          if 5-6 then
            update condition for 8;
            if condition for 8 is true then
            post(sem(8)) endif;
            6;
            update condition for 8;
            if condition for 8 is true then
            post(sem(8)) endif;
          else
            update condition for 8;
            if condition for 8 is true then
            post(sem(8)) endif;
            7;
            update condition for 8;
            if condition for 8 is true then
            post(sem(8)) endif;
          endif }
        { wait(sem(8)); 8 }
        coend
      endif }
    coend
  else
    { 8 }
  endif }
{ 9 }
coend
```

● The statements of a block {stmt_1; stmt_2; ...; stmt_n} execute in sequential order.

Fig. 9. Parallel code for our example flow graph.

replace Step 3 above by just the corresponding **post**. For instance, in our example, as the condition for 3 is just 2, node 2 instead of updating the condition for 3 proceeds directly with **post**($sem(3)$).

2) While generating code, the placement of a node in the program will cause certain conditions to be true at that point in the program, and hence the condition for a node can be further simplified. For example, when 8 is executed under the branch 1–8, 1–8 is true at that point in the program and hence the condition for 8 is true (both clauses of the conjunction will be true, see Table II) and hence 8 need not wait for any event. However, when 8 is placed under the 3–4 branch, no such inference can be made and there must be a synchronization instruction for 8.

It will be observed that the code for node 8 has been duplicated. Duplication of code is a known problem while generating code from the control dependence graph and can be avoided, sometimes at the expense of parallelism [11].

Another optimization which can be done is to note that since 2 is the only node updating the condition for 3, and 3 and 2 are identically control dependent, the synchronization can be removed by executing 2 and 3 in sequence. The same applies for nodes 4 and 5. For more details on parallel source generation, the reader is referred to [13].

X. Conclusion

The capacity of a compiler to encapsulate task-level parallelism from a sequential or parallel program is critically dependent on accurate estimation of data and control dependences, and more importantly on the derivation of minimal execution constraints for each task in a program. In this paper we presented a framework for the construction of a program's task graph based on data and control dependences, the derivation of execution conditions for each task node, and the optimization of these conditions. Moreover, through an example we showed how *HTG*, our intermediate representation, can be used to generate parallel source code from a sequential program.

Brute-force derivation of control and data constraints would result in little or no parallelism in the resulting code. Finding the minimal set of such constraints necessary to preserve program correctness during parallel execution is therefore instrumental in extracting and exploiting parallelism from sequential and parallel programs.

Acknowledgment

The authors would like to thank referee B for his many useful comments and suggestions, which helped improve the presentation of this work.

References

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

[2] F.E. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An overview of the PTRAN analysis system for multiprocessing," *J. Parallel Distributed Comput.*, vol. 5, no. 5, pp. 617–640, Oct. 1988.

[3] J.R. Allen, "Dependence analysis for subscripted variables and its application to program transformations," Ph.D. dissertation, Dep. Math. Sci., Rice Univ., Houston, TX, Apr. 1983.

[4] R. Allen and K. Kennedy, "Automatic translation of FORTRAN programs to vector form," *ACM Trans. Programming Languages Syst.*, vol. 9, no. 4, Oct. 1987.

[5] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein, "The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages," in *Proc. ACM SIGPLAN'90 Conf. Programming Language Design and Implementation*, June 1990, pp. 257–271.
[6] U. Banerjee, "Speedup of ordinary programs," Ph.D. dissertation, Dep. Comput. Sci., Univ. of Illinois at Urbana–Champaign, Oct. 1979.
[7] ——, *Dependence Analysis for Supercomputing*. Norwell, MA: Kluwer Academic, 1988.
[8] E. Coffman, Jr., Ed., *Computer and Job-shop Scheduling Theory*. New York: Wiley, 1976.
[9] R. Cytron, "Compile-time scheduling and optimization for asynchronous machines," Ph.D. dissertation, Dep. Comput. Sci., Univ. of Illinois at Urbana–Champaign, Urbana, IL, Oct. 1984.
[10] R. Cytron, J. Ferrante, and V. Sarkar, "Experiences using control dependence in ptran," *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau, and D. A. Padua, Eds. Cambridge, MA: MIT Press, 1990, pp. 186–212.
[11] R. Cytron, M. Hind, and W. Hsieh, "Automatic generation of DAG parallelism," in *Proc. 1989 SIGPLAN Conf. Programming Language Design and Implementation*, July 1989, pp. 54–68.
[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Programming Languages Syst.*, vol. 9, no. 3, pp. 319–349, July 1987.
[13] M. Girkar, "Automatic detection and management of parallelism in programs," Ph.D. dissertation, Center for Supercomput. Res. and Develop, Univ. of Illinois at Urbana–Champaign, Aug. 1991, in preparation.
[14] M. Girkar and C. D. Polychronopoulos, "A universal intermediate representation for parallel programs based on control and data dependences," Tech. Rep. 1046, Center for Supercomput. Res. and Develop., Univ. of Illinois at Urbana–Champaign, 1990.
[15] H. Kasahara, H. Honda, M. Iwata, and M. Hirota, "A compilation scheme for macro-dataflow compuatation on hierarchical multiprocessor systems," unpublished manuscript, 1989.
[16] D. J. Kuck, *The Structure of Computers and Computations, Vol. I*. New York: Wiley, 1978.
[17] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. J. Wolfe, "Dependence graphs and compiler optimizations," in *Proc. 8th Annu. ACM Symp. Principles Programming Languages*, ACM, Jan. 1981, pp. 207–218.
[18] C. Lee, "On run-time systems for parallel supercomputers," Master's thesis, Univ. of Illinois at Urbana–Champaign, May 1990.
[19] S. P. Midkiff and D. A. Padua, "Compiler algorithms for synchronization," *IEEE Trans. Comput.*, vol. C-36, pp. 1485–1495, Dec. 1987.
[20] C. D. Polychronopoulos, "Toward auto-scheduling compilers," *J. Supercomput.*, pp. 297–330, 1988.
[21] ——, "Auto scheduling: Control flow and data flow come together," Tech. Rep. 1058, Center for Supercomput. Res. and Develop., Univ. of Illinois at Urbana–Champaign, 1990.
[22] 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. 1989 Int. Conf. Parallel Processing*, St. Charles, IL, Aug. 1989.
[23] M. Weiser, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, no. 7, pp. 446–452, July 1982.
[24] M. J. Wolfe, "Optimizing supercompilers for supercomputers," Ph.D. dissertaton, Dep. Comput. Sci., Univ. of Illinois at Urbana–Champaign, 1982.

Milind Girkar received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Bombay, and the M.S. degree in computer science from Vanderbilt University, Nashville, TN.

He is currently a Ph.D. candidate in computer science at the Center for Supercomputing Research and Development at the University of Illinois at Urbana–Champaign. His research interests are in parallelizing compilers and program restructuring.

Constantine D. Polychronopoulos (S'85–M'86) received the Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign

He has been a faculty member of the Department of Electrical and Computer Engineering and the Center for Supercomputing Research and Development of the University of Illinois since 1986. He serves on the editorial board of the JPDC and the *International Journal on High-Speed Computing*, and is the author of a book and several technical journal and conference papers. His research interests are on parallel computer architectures, their compilers, and operating systems.

Dr. Polychronopoulos was a recipient of the Presidential Young Investigator Award in 1989.