APPROXIMATE ALGORITHMS FOR OPTIMIZATION OF BUSY WAITING IN PARALLEL PROGRAMS

by

Edmund M. Clarke*
Lishing Liu**

TR-12-79

* Harvard University
Aiken Computation Laboratory
Cambridge, MA 02138

** Mitre Corporation
Bedford, MA 01730

ABSTRACT

Traditional implementations of conditional critical regions and monitors can lead to unproductive "busy waiting" if processes are allowed to wait on arbitrary boolean expressions. Techniques from global flow analysis may be employed at compile time to obtain information about which critical regions (monitor calls) are enabled by the execution of a given critical region (monitor call). We investigate the complexity of computing this information and show how it can be used to obtain efficient scheduling algorithms with less busy waiting.

TABLE OF CONTENTS

SECTION 1

INTRODUCTION

BACKGROUND

Hoare [HO72] and Brinch Hansen [BH73] have proposed conditional critical regions as a synchronization primitive for parallel programs. With this primitive, logically related variables which must be accessed by more than one process are grouped together as resources. Individual processes are allowed access to a resource R only in a critical region of the form "with R when b do A od" where b is a boolean expression and A is a statement whose execution may change the values of the shared variables in R. When execution of a process reaches the conditional critical region, the process is delayed until no other process is using resource R and condition b is satisfied. The statement A is then executed as an indivisible operation.

Unfortunately, the standard implementation ([HO72], [BH73]) of conditional critical regions may result in busy waiting where the scheduler repeatedly attempts to activate a process that is waiting on a false condition. The standard implementation uses two queues for each resource R: a main queue R_m and a wait queue R_w. When a process wishes to enter a conditional critical region for resource R, it enters R_m. Processes on R_m are allowed to enter their critical regions one at a time and inspect the variables of R to see if the entry condition b is satisfied. If so, the process completes its critical region by executing statement A. Otherwise, the process leaves its critical region and is put on R_w. When a process successfully

executes the body of its conditional critical region and changes the values of the shared variables in R, it may cause some of the conditions on which processes in R_w are waiting to become true. Thus, all processes in R_w must be transferred to the main queue and allowed to reevaluate their conditions. (See figure 1.)

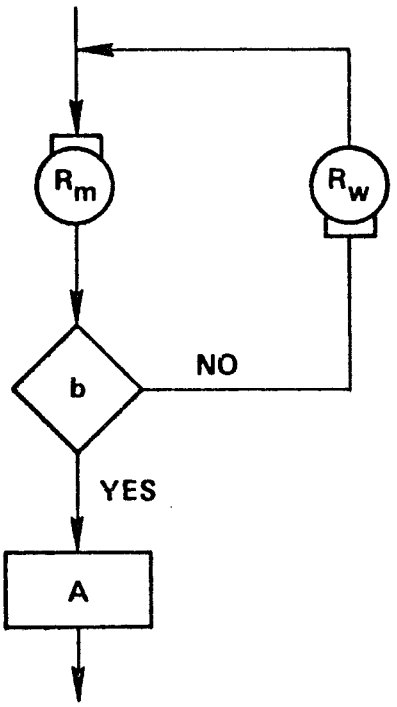

Figure 1

Note that a process may be transferred from R_m to R_w and back several times before it finally executes its critical region. According to Brinch Hansen [BH73] this busy waiting "is the price we pay for the conceptual simplicity achieved by using arbitrary boolean expressions as synchronizing conditions." Note that essentially the same type of busy waiting can occur with monitors [HO73] provided that monitor procedures are allowed to contain wait statements on arbitrary boolean expressions.

NEW RESULTS OF THIS PAPER

More efficient implementations of conditional critical regions may be obtained if programs are preprocessed to obtain information about which conditional critical regions are enabled by the execution of a given conditional critical region. Consider the standard solution to the readers and writers problem with writer priority [BH73]. Here, resource R consists of three shared variables: aw (the number of writer processes waiting to execute a write statement), rw (the number of writer processes currently executing write statements), and rr (the number of readers currently executing read statements). Each reader process has the form

```
Reader: repeat
            R1: with R when aw=0 do rr:=rr+1 od
            read;
            R2: with R when true do rr:=rr-1  od
        forever
```

and each writer process has the form

```
Writer: repeat
            W1: with R when true do aw:=aw+1  od
            W2: with R when rr=0 and rw=0 do
                        rw:=1  od
            write;
            W3: with R when true do
                        rw:=0; aw:=aw-1  od
        forever
```

Suppose that at some point during a computation $reader_1$ has been placed on the wait queue R_w because it is attempting to execute critical

region R1 and aw > 0. If $reader_2$ successfully executes critical region R2, then it is unnecessary to transfer $reader_1$ to the main queue R_m since the condition on which it is waiting has not changed.

A first step towards obtaining such information at compile time has been made by Schmid [SC76]. By restricting conditional critical regions to have the form

$$\begin{aligned} &\text{CCR: } \underline{\text{with}}\ R(\bar{x})\ \underline{\text{when}}\ a_1x_1 + \ldots + a_nx_n + a_{n+1} \geq 0\ \underline{\text{do}} \\ &\qquad x_1 := x_1 + b_1; \\ &\qquad\quad \vdots \\ &\qquad x_n := x_n + b_n\ \underline{\text{od}} \end{aligned}$$

he is able to give heuristics for determining when execution of CCR_i can enable the condition of CCR_j (the enable relation ea). Schmid's technique, however, does not easily generalize to more complicated conditional critical regions which do not obey his restrictions on conditions and assignment statements.

We investigate the complexity of precisely computing the relation ea. We show that even for very restricted critical regions it is impossible to devise feasible algorithms for computing this relation. By using techniques from global flow analysis, however, we show that useful approximations to ea may be obtained for a wide class of conditional critical regions.

We also present a scheduling algorithm for conditional critical regions which uses the information provided by our flow analysis algorithm to eliminate reevaluation of conditions which cannot be enabled. As with most optimization techniques, we cannot claim that our algorithm is optimal. However, it is possible to prove that our algorithm is conservative, i.e., our algorithm will never fail to determine that an enabled process is really enabled. The

proof that our algorithm is conservative uses a fixpoint characterization of the data generated by the flow analysis.

Although global analysis has been applied to many optimization problems for sequential programs ([AL70], [CO76], [KE71], [HU75], [KI73]) and to deadlock detection and constant propagation for parallel programs [RE79], we believe that this is the first attempt to use such techniques for the optimization of busy waiting in parallel programs.

OUTLINE OF PAPER

In Section 2 we describe a simple parallel programming language in which processes access shared data via conditional critical regions. Complexity issues involved in optimization of busy waiting are discussed in Section 3. Section 4 presents the basic ideas of global flow analysis which are needed to understand the remainder of the paper. In Section 5 we show how Kildall's algorithm [KI73] can be modified to obtain information about which conditional critical regions may be enabled by the execution of a given conditional critical region, and in Section 6 we show how this information can be used to obtain more efficient scheduling algorithms. The paper concludes with a discussion of the results and some remaining open problems.

SECTION 2

A SIMPLE PARALLEL PROGRAMMING LANGUAGE

A parallel program will consist of two parts: an initialization part "$\bar{x}:=\bar{e}$" in which values are assigned to the program variables $\bar{x}$, and a parallel execution part "resource $R(\bar{x})$ cobegin $P_1//P_2// \ldots //P_t$ coend" which permits the simultaneous or interleaved execution of the statements in the processes $P_1, \ldots, P_t$. All variables accessed by more than one process must appear in the prefix $R(\bar{x})$ of the parallel execution part. Each process P_i is a sequential program composed of simple Algol statements (assignment, conditional, while, etc.) and conditional critical regions. Conditional critical regions have the form "with R when b do A od" where b is a boolean expression and A is a sequence of assignment statements. Shared variables can only be referenced within critical regions, and critical regions can only reference shared variables and constants. Thus, we ignore the flow of information between program variables and synchronization variables. The programs that we consider can be regarded as the synchronization skeletons of actual parallel programs. In a future paper, we plan to discuss in detail the interaction between program variables and synchronization variables.

Let P be a parallel program of the form described above where $\bar{x} = (x_1, \ldots, x_m)$ is the set of shared variables, and the statements of each process Pi are $S_1, S_2, \ldots, S_{k_i}$. A state of P in an (m+t)-tuple $\sigma = (v_1, \ldots, v_m, p_1, \ldots, p_t)$ where v_j is the integer value of variable

x_j and p_i is the program counter for process P_i. σ_0 will denote the state $(\bar{e}, 1, \ldots, 1)$ of the program after initialization $\bar{x} := \bar{e}$. The function $pc_i(\sigma)$ gives the value of the i^{th} program counter in state σ, and the function $val(x, \sigma)$ gives the value of shared variable x in state σ. Both functions may be extended in the natural manner to apply to sets of states. We write $\sigma \overset{C_i}{\Longrightarrow} \sigma'$ if critical region C_i can be executed in state σ to produce state σ'. Likewise $\sigma_0 \overset{P}{\Longrightarrow} \sigma$ indicates that state σ can occur during execution of program P on initial state σ_0.

Let CCR be the set of conditional critical regions in program P. An enable relation for P is a binary relation ea on CCR with the property that $(C_i, C_j) \in ea$ whenever there is a computation of P of the form $\sigma_0 \overset{P}{\Longrightarrow} \sigma_k \overset{C_i}{\Longrightarrow} \sigma_{k+1} \overset{C_j}{\Longrightarrow} \sigma_{k+2}$ and condition b_j of region C_j is false in state σ_k. Similarly, we may define a disable relation da for P such that $(C_i, C_j) \in da$ whenever there exists a computation $\sigma_0 \overset{P}{\Longrightarrow} \sigma_k \overset{C_i}{\Longrightarrow} \sigma_{k+1}$ with b_j true in σ_k but not in σ_{k+1}. Note that any super-set of an enable (disable) relation is also an enable (disable) relation according to this definition.

SECTION 3

COMPLEXITY ISSUES

We investigate the complexity of computing the minimal ea and da relations for four different sets of restrictions on the programming language of Section 2. With each of these restrictions processes are nonterminating loops of the form repeat C_i : with R when b_i do A_i od forever. In describing the restrictions Z will denote the set of integers, N the set of non-negative integers, and N_d the set of non-negative integers smaller than d.

The first language L_1 is essentially the language considered by Schmid [SC76]; L_1 programs are required to satisfy the following three conditions:

1. All shared variables range over the integers Z.

2. The conditions of conditional critical regions are monotone boolean combinations of linear inequalities of the form

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n + a_{n+1} \geq 0,$$

where $a_i \in N$ for $1 \leq i \leq n$ and $a_{n+1} \in Z$.

3. The bodies of conditional critical regions are sequences of assignment statements of the form x:=x+d where x is a shared variable in R and $d \in Z$.

Theorem. The problem of computing the minimal ea (da) relation for L_1 programs is recursively unsolvable.

Proof: Recall that the halting problem for counter machines [MI67] is recursively unsolvable. Given an arbitrary counter machine M, we show how to construct an L_1 program P with the property that a particular critical region pair (C_1, C_2) will be in the minimal ea (da) relation for P iff M halts. Each variable a of the counter machine M will be represented by a pair of variables a^+ and a^- such that $a^+ = a$ and $a^- = -a$. In addition, P will use the variable pc as the program counter for M. There are four types of counter machine instructions; we show how each of these instructions can be simulated using restricted conditional critical regions. We assume that each statement in the program for the counter program has a unique label and that L+1 is the label of the statement which follows the statement labeled L.

1. "L: a:=a+1" is simulated by:

repeat

with R when $(pc^+ - L \geq 0) \wedge (pc^- + L \geq 0)$ do

$a^+ := a^+ + 1$; $a^- := a^- - 1$;

$pc^+ := pc^+ + 1$; $pc^- := pc^- - 1$; od

forever

2. "L: a:= 0" is simulated by:

repeat

with R when $(pc^+ - L \geq 0) \wedge (pc^- + L \geq 0) \wedge (a^+ - 1 \geq 0)$ do

$a^+ := a^+ - 1$; $a^- := a^- + 1$ od

forever

//

repeat

with R when $(pc^+ - L \geq 0) \wedge (pc^- - L \geq 0) \wedge (a^- \geq 0)$ do

$pc^+ := pc^+ + 1$; $pc^- := pc^- - 1$ od

forever

3. "L: if a > 0 then a := a - 1 else go to N" is simulated by:
(We assume that $N \leq L$. The case in which $N \geq L$ is similar.)

repeat

with R when $(pc^+ - L \geq 0) \wedge (pc^- + L \geq 0) \wedge (a^+ - 1 \geq 0)$ do

$a^+ := a^+ - 1$; $a^- := a^- + 1$;

$pc^+ := pc^+ + 1$; $pc^- := pc^- - 1$ od

forever

//

repeat

with R when $(pc^+ - L \geq 0) \wedge (pc^- + L \geq 0) \wedge (a^- \geq 0)$ do

$pc^+ := pc^+ + 1$; $pc^+ := pc^+ + 1$

$pc^- := pc^- - 1$; ... $pc^- := pc^- - 1$

(N-L times)

od

forever

4. "L: Halt" is simulated by:

C_1: repeat

with R when $(pc^+ - L \geq 0) \wedge (pc^- + L \geq 0)$ do

flag: = flag + 1 od

forever

//

C_2: repeat

with R when (flag - 1 ≥ 0) do od

forever

The variable flag is initialized to 0. Thus, $(C_1, C_2) \in ea$ iff M halts. The case of disables relation da is similar and will be left to the reader. □

Note that the construction of Theorem 3.1 would be impossible if shared variables were required to be non-negative. In Language L_2 we investigate the effect of requiring that shared variables be non-negative. L_2 programs are exactly like L_1 programs except that condition (1) is modified as follows:

(1a) After initialization, all shared variables are non-negative, i.e.,

$$val(\bar{x}, \sigma_0) \geq 0.$$

(1b) Each critical region

"with R when $B(\bar{x})$ do $\bar{x}$: = $F(\bar{x})$ od"

must have the property that

$$\forall \bar{x} \in Z^n [(B(\bar{x}) \wedge \bar{x} \geq 0) \Rightarrow F(\bar{x}) \geq 0] .$$

From (1a) and (1b) it follows that the shared variables $\bar{x}$ will always be non-negative during the execution of an L_2 program. For L_2 programs the

problem of determining the ea (da) relations is closely related to the interval reachability problem for vector replacement systems [KL77].

Definition: An n-dimensional vector replacement system is a finite set of pairs of vectors $\{(\bar{U}_1, \bar{V}_1), (\bar{U}_2, \bar{V}_2), \ldots, (\bar{U}_m, \bar{V}_m)\}$ such that $\bar{U}_i \in N^n$, $\bar{V}_i \in Z^n$ for $1 \leq i \leq m$. □

Let Γ be an n-dimensional vector replacement system and let $\bar{x}_1, \bar{x}_2$ be two vectors in N^n. We say that $\bar{x}_2$ is directly reachable from $\bar{x}_1$ if there is a pair of vectors $(\bar{U}, \bar{V})$ in Γ such that $\bar{x}_1 - \bar{U} \geq 0$ and $\bar{x}_2 = \bar{x}_1 - \bar{U} + \bar{V}$. We say that $\bar{x}_2$ is reachable from $\bar{x}_1$ if there is a sequence of vectors $\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_k$ in N^n such that $\bar{x}_1 = \bar{y}_1$, $\bar{x}_2 = \bar{y}_k$, and $\bar{y}_{j+1}$ is directly reachable from $\bar{y}_j$ for $1 \leq j \leq k - 1$. Let $I_1, \ldots, I_n$ be a collection of intervals of the form $[a_i, b_i)$ where $a_i \in N$ and either $a_i < b_i$ or $b_i = +\infty$. The interval reachability problem is the problem of determining if there exists a point $r \in I_1 \times I_2 \times \ldots \times I_n$ such that r is reachable by Γ from the origin $\bar{O}$.

Note that the ordinary reachability problem for vector replacement systems is a special case of the interval reachability problem. From the work of Lipton [LI77] on the ordinary reachability problem, it follows that the interval reachability problem is EXSPACE hard. We prove:

Theorem. The interval reachability problem for vector replacement systems is polynomially reducible to the problem of determining the minimal ea (da) relation for L_2 programs.

Proof: Let $\Gamma = \{(\bar{U}_1, \bar{V}_1), \ldots (\bar{U}_m, \bar{V}_m)\}$ be an n-dimensional vector replacement system and let $[a_1, b_1), \ldots, [a_n, b_n)$ be a collection of n intervals which determine an interval reachability problem. We construct

an L_2 program P with m+2 processes where m is the number of vector pairs in Γ. P will contain a process of the form

repeat

with R when $(\overline{X}-\overline{U}\geq 0)\wedge(\text{flag1}-1\geq 0)\wedge(\text{flag2}-1\geq 0)$ do

$\overline{x} := \overline{X} - \overline{U} + \overline{V}$ od

forever

for each pair $(\overline{U},\overline{V})$ in Γ. In addition P will contain the two processes C_1, C_2 shown below:

C_1: repeat

with R when $\bigwedge_{i=1}^{n}(x_i-a_i\geq 0)\wedge(\text{flag1}-1\geq 0)\wedge(\text{flag2}-1\geq 0)$ do

$\overline{x} := \overline{x} - \overline{a} + \overline{b}$;

flag1 := flag1 - 1 od

forever

C_2: repeat

with R when $\bigvee_{i=1}^{n}(x_i-b_i\geq 0)\wedge(\text{flag2}-1\geq 0)$ do

flag2 := flag2 - 1 od

forever

For infinite intervals, e.g., $b_k = +\infty$ the construction of C_1 and C_2 must be modified slightly. In this case the assignment $x_k := x_k - a_k + b_k$ is omitted from the body of C_1 and the disjunct $x_k - b_k \geq 0$ from the condition of C_2. The variables flag1 and flag2 are used to prevent processes C_1 and C_2

from being executed more than once and are initialized to 1; all other shared variables are initialized to 0.

Note that $(C_1, C_2) \in ea$ iff there is a computation of P such that at some point the predicate

$$\bigwedge_{i=1}^{n} \left(x_i - a_i \geq 0\right) \wedge \neg \left[\bigvee_{i=1}^{n} \left(x_i - b_i \geq 0\right) \right] \wedge (\bar{x} + \bar{b} - \bar{a} \geq \bar{b})$$

is true. Rearranging the predicate we obtain $\bigwedge_{i=1}^{n} (x_i \geq a_i) \wedge \bigwedge_{i=1}^{n} (x_i \leq b_i)$ or simply $\bigwedge_{i=1}^{n} (a_i \leq x_i \leq b_i)$. Thus $(C_1, C_2) \in ea$ iff there exists a point $r \in I_1 \times I_2 \times \ldots \times I_n$ which is reachable from the origin $\bar{0}$ by a computation of Γ. As in Theorem 1 the case of the disables relation is similar and will be left to the reader.

In language L_3 shared variables range over N_d. Boolean expressions of conditional critical regions are boolean combinations of atomic formulas of the form $x_i = k$ where x_i is a shared variable and $k \in N_d$. Bodies of conditional critical regions are sequences of assignment statements of the form $x_i := k$ or $x_i := x_j$ where x_i, x_j, and k are as described above.

Theorem. The problem of computing the minimal ea (da) relation for L_3 programs is PSPACE complete.

Proof: We first show that the problem of determining the minimal ea (da) relation for L_3 programs is PSPACE hard. Recall that the problem of determining if $L(M) = \Sigma^*$ is PSPACE hard when M is a nondeterministic finite automaton with input alphabet Σ. Let $M = (Q, \Sigma, \delta, q, F)$ be a nondeterministic finite automaton. We show how to construct an L_3 program P with the property that a particular critical region pair (C_1, C_2) will be in the minimal ea (da) relation for P iff $L(M) \neq \Sigma^*$. The parallel program P will

guess inputs for M and simulate M on these inputs. The construction is complicated by the fact that the shortest string $\sigma \notin L(M)$ might be quite long in comparison to M. Thus, it is necessary to guess the string σ one symbol at a time. Note also that the program P must deterministically simulate the nondeterministic behavior of M, since an input string σ could be accepted by one computation of M and not accepted by another.

In order to best explain how the program P works, we group the processes of P into three distinct parts. Part 1 guesses inputs for M one symbol at a time by setting up race conditions in the parallel program P. Part 2 deterministically simulates nondeterministic behavior of M. Part 3 determines if a string has been found which is not accepted by M. We assume that the input alphabet of M is $\Sigma = \{ \sigma_1, \ldots, \sigma_k \}$ and that the state set is $Q = \{ q_1, \ldots, q_s \}$. We also assume (without loss of generality) that q_s is the only final state of M. All variables of the program P will belong to the same resource R.

Part 1: This part of the program P will be executed each time that it is necessary to guess a new input symbol for M. There is one process for each input symbol in Σ, i.e.,

$$\text{Guess\_}\sigma_1 \,//\, \text{Guess\_}\sigma_2 \,//\, \ldots \,//\, \text{Guess\_}\sigma_k \ ,$$

and each of these processes has the form

```
Guess_σ_i: repeat
               with R when guess = 1 do
                   guess := 0; input := σ_i;
                   sim := 1 od
               forever
```

If the variable "guess" is initialized to 1, the first process to set "guess" to 0 will be the one which selects the new input symbol. The variable "sim," initially zero, is used to flag the simulation part (part 2) when it can proceed.

Part 2: In this part we deterministically simulate one step in the nondeterministic computation of M. The simulation uses two sets of 0, 1 — valued variables

$$q'_1, q'_2, \ldots q'_s \qquad\qquad q^*_1, q^*_2, \ldots, q^*_s$$

to represent the possible states that M could be in before and after the new input symbol is processed. A maximum of k * s + 1 processes are needed to simulate one nondeterministic move of M:

state #1_sym #1 // ... // state #1_sym #k //
state #2_sym #1 // ... // state #2_sym #k //
. . .
state #s_sym #1 // ... // state #s_sym #k //
recopy

The first k*s processes are used to encode the transition function of M and have the form

State #i_sym #j: repeat

with R when (sim = i) $\wedge$ (q'_i = 1) $\wedge$ (input = σ_j) do

/* $\delta(q_i, \sigma_j) = \{q_{t_1}, q_{t_2}, \ldots q_{t_r}\}$ */

$q^*_{t_1} := 1; q^*_{t_2} := 1; \ldots q^*_{t_r} := 1;$

sim: = i + 1 od

forever

One additional process is needed to recopy state information from the array q* into the array q'.

recopy: repeat

with R when (sim = s + 1) do

$q'_1 := q^*_1; \ldots q'_s := q^*_s;$

$q^*_1 := 0; \ldots q^*_s := 0;$

sim := 0; test := 1 od

forever

Part 3: Here we determine if the string guessed so far should be rejected. If so, then $L(M) \neq \Sigma^*$ and we insure that the critical region pair (C_1, C_2) is included in the minimal enable relation ea. (The case of the disables relation da is similar.) Otherwise, we arrange for execution to return to Part 1 of the program and a new input symbol to be guessed.

LOOP: repeat

with R when (test = 1) $\wedge$ (q'_s = 1) do

test := 0; guess := 1 od

forever

//

C_1: repeat

with R when (test = 1) $\wedge$ ($q'_s \neq 1$) do

test := 0; flag := 1 od

forever

//

C_2: repeat

with R when flag := 1 do

flag := 0 od

forever

Note that the size of program P is polynomial (linear, in fact) in the size of the nondeterministic automaton M. Note also that P can be constructed from M by a LOGSPACE bounded Turing Machine.

We must prove that the minimal ea (da) relation can be computed in PSPACE. We first show how to construct a nondeterministic PSPACE bounded Turing machine M to recognize strings of the form $P\#(C_1, C_2)$ where P is an L_3 program and (C_1, C_2) is a pair of conditional critical regions in the minimal ea (da) relation for P. The machine M keeps a list of the variables in P and their current values on one of its work tapes. The list is initialized to reflect the assignments made in the initialization part of P. M then repeatedly examines the conditional critical regions in the processes of P and nondeterministically selects one whose condition is true. M then executes the body of the critical region and updates its copy of P's variables. If the selected critical region was C_1, M also checks to see if execution of C_1 enables (disables) C_2. If so, then M accepts its input; otherwise M selects another critical region and the sequence is repeated. By Savitch's theorem there is a deterministic PSPACE bounded Turing machine M' which accepts exactly the same set of input tapes as M. By using M' repeatedly on all possible pairs of critical regions in P, the minimal ea (da) relation for P can be computed in PSPACE. □

Language L_4 is like L_3 except that processes are not allowed to contain loops; i.e., processes are restricted to be single critical regions of the form

Proc N: with R when b do S od.

Theorem. The problem of determining the minimal ea (da) relations for L_4 programs is NP-complete.

Proof: The construction is similar to that used in Theorem 3.4 and will be left to the reader. □

Because of these negative results, we devote the remainder of the paper to finding good (i.e., non-minimal) approximations for the enable and disable relations.

SECTION 4

BASIC DEFINITIONS OF GLOBAL DATA FLOW ANALYSIS

Research on sequential program optimization ([CO76], [GW76], [HU75], [KI73], [KU76], [LD79]) has demonstrated the importance of modeling information flow in programs with lattice-theoretic concepts.

Definition. An (upper) semi-lattice is a pair $(L, \vee)$ where L is a set of elements and $\vee$ is a binary meet operator on L such that $x \vee x = x$, $x \vee y = y \vee x$, and $x \vee (y \vee z) = (x \vee y) \vee z$ for all x, y, and z in L. □

Let $(L, \vee)$ be a semi-lattice. We say that $x \leq y$ if $x \vee y = y$ and that $x < y$ if $x \leq y$ and $x \neq y$. The semi-lattice is complete if for each subset s of L there is a least upper bound $\vee S$ of S in L with respect to the partial order $\leq$ (i.e., $\vee S \in L$; $x \leq \vee S$ for all $x \in S$; and if $y \in L$ and $x \leq y$ for all $x \in S$, then $\vee S \leq y$). A sequence of lattice elements $x_1, x_2, \ldots, x_k$ is an ascending chain of length k if $x_i \leq x_{i+1}$ for all $1 \leq i < k$.

In data flow analysis, programs are modeled by directed graphs where the nodes correspond to individual statements and the arcs represent the flow of control. We restate below the formal definition of a flow graph together with the necessary related terminology from graph theory.

Let $G = (N, E)$ be a directed graph. If $e = (n, n') \in E$, we call n' a successor of e and n a predecessor of n'. Succ(n) will denote the set of all successors of the node n. A sequence of edges $\pi = (n_0, n_1), (n_1, n_2), \ldots, (n_{k-1}, n_k)$ is called a path from n_0 to n_k of length k. A path of length 0

is called a null path. Path (n,n') will denote the set of all paths from n to n'.

Definition. A flow graph $G = (N,E,n_0)$ is a finite directed graph (N,E) together with a distinguished entry node n_0 such that n_0 has no predecessor and $\text{Path}(n_0,n) \neq \emptyset$ for all $n \in N$. □

Many flow analysis problems can be formulated as information propagation problems in flow graphs ([GW76], [LD79]).

Definition. An information propagation problem is a 5-tuple $<G,L,\vee,F,x_0>$ where $G = (W,E,n_0)$ is a flow graph,

$(L,\vee)$ is a complete upper semi-lattice with join $\vee$,

$F = \{ f_e : L \to L \mid e \in E \}$ is a set of transition functions, and

$x_0 \in L$ is the initial information attached at node n_0. □

A transition function $f_e : L \to L$, where $e \in E$, specifies how information changes when it flows through edge e. To solve an information propagation problem is to merge, for each node n of the flow graph, the set of information which can be propagated from x_0 through a path from n_0 to n.

Definition. Let $I = <G,L,\vee,F,x_0>$ be an information propagation problem. The join of paths solution (JOP) to I is the mapping $JP:N \to L$ such that, for each $n \in N$,

$$JP(n) = \bigvee_{\pi \in \text{Path}(n_0,n)} f_\pi(x_0)$$

where $f_\pi = f_{e_k} \circ f_{e_{k-1}} \circ \cdots \circ f_{e_1}$ if $\pi = e_1, e_2, \cdots, e_k$. □

In our definition of an information propagation problem we require that the semi-lattice be complete so that the JOP solution is always well defined. However, it is often difficult to solve for the JOP ([KU76], [LD76]). Various algorithms ([GW76], [KI73], [KU76], [HU75], [LD79]) have been proposed for approximating join of paths solutions by fixpoint techniques.

Definition. Let $I = <G, L, \vee, F, x_0>$ be an information propagation problem. A fixpoint of I is a mapping such that $FP(n_0) = x_0$, and

$$FP(n) = \bigvee_{(n',n)\in E} f_{(n',n)}(FP(n')),\ n \neq n_0 \quad . \qquad \square$$

It can be shown that if each f_e is continuous (i.e., $f_e(\vee U) = \vee_{x\in U} f_e(x)$), then I has a least fixed point.

SECTION 5

COLLECTION OF INFORMATION IN PARALLEL PROGRAMS

Let P be a parallel program with processes $P_1, P_2, \ldots, P_t$. Assume that $C_{i1}, C_{i2}, \ldots, C_{ik_i}$ are the critical regions which occur in process P_i and that each C_{ij} has the form "with R when b_{ij} do A_{ij} od," where A_{ij} consists of assignment statements. Let $ST(i,j) = \{\sigma \mid \sigma_0 \xrightarrow{P} \sigma, pc_i(\sigma) = j\}$ be the set of program states which may precede the execution of C_{ij}. In general it is impossible to calculate ST(i,j) since this would require simulation of all possible executions of the program P. However, by using techniques from global flow analysis it is possible to obtain good approximations for ST(i,j). Before discussing such techniques we describe how flow graphs are constructed for parallel programs.

We assume that for each process P_i a flow graph $G_i = (N_i, E_i, n_{i0})$ is constructed such that every conditional critical region C_{ij} is represented by a single node. Each process also has a special start node n_{i0}. The composite flow graph $G = (N, E, n_0)$ for P is constructed as follows:

1. $N = \bigcup_{i=1}^{t} N_i \cup \{n_0\}$

2. $E = \bigcup_{i=1}^{t} E_i \cup \{(n_0, n_{i0}) \mid i \leq i \leq t\} \cup E'$, where

$$E' = \{(n_{iq}, n_{jr}) \mid q \neq 0,\ r \neq 0, \text{ and } i \neq j\}$$

The flow graph for solution to the readers and writers problem is shown in figure 2. For simplicity we consider only the case of one reader process and one writer process. In the flow graph an undirected edge (n,n') between critical regions in different processes represents two directed edges (n,n') and (n',n). Note that many of the interprocess edges are unnecessary because they are not part of possible execution paths. During the execution of our flow analysis algorithms, the interprocess edges need never be explicitly constructed, and many inactive edges will not be used at all.

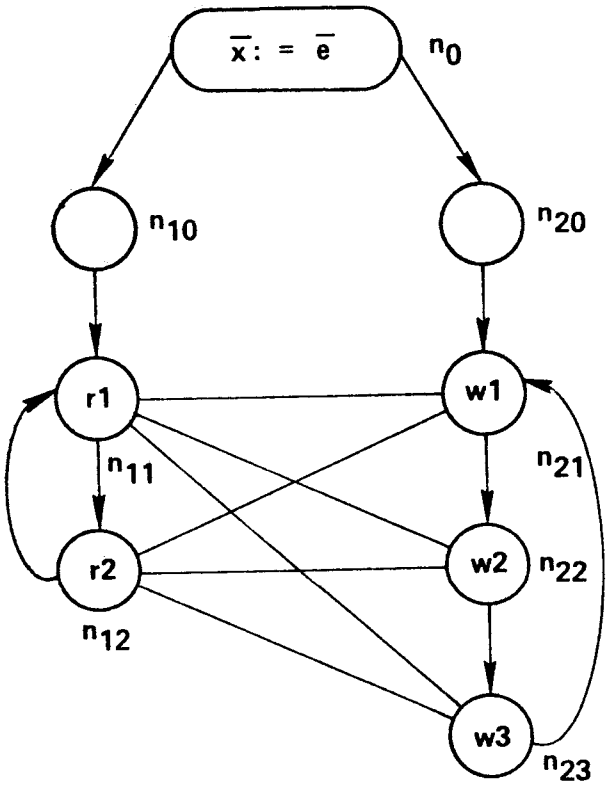

Figure 2

Let $LS = \mathcal{P}(\Sigma)$ where Σ is the set of program states for P. Note that $(LS, \cup)$ is a complete upper semi-lattice. If flow graph G is constructed as described above, then transition functions $f_e : LS \rightarrow LS$ for G may be defined as follows: We first define the function $f_n : LS \rightarrow LS$, for each node nn, such that

$$f_n(X) = \{ \text{post}(n, \sigma) \mid \sigma \in X \text{ and the instruction at } n \text{ is executable under } \sigma \},$$

where $\text{post}(n, \sigma)$ denotes the state resulted by executing the instruction at n under the state σ. For instance, if n represents a conditional critical region C_{ij}, then $f_n(X) = \{ A_{ij}(\sigma) \mid \sigma \in X \ \& \ b_{ij}(\sigma) = \text{true} \}$. For an edge $e = (n, n')$, where $n' = n_{rq}$, the function $f_e : LS \rightarrow LS$ is defined as

$$f_e(X) = \{ \sigma \in f_n(X) \mid pc_r(\sigma) = q \} \quad .$$

Theorem. Consider the information propagation problem $I = \langle G, LS, \cup, \{f_e\}, \emptyset \rangle$, then

a. Each transition function f_e is continuous with respect to the lattice $(LS, \cup)$.

b. The least fixpoint of I is the JOP solution to I.

c. If $JP : N \rightarrow LS$ is the JOP solution to I and n is a node representing critical region C_{ij}, then $JP(n) = ST(i, j)$.

Proof: omitted. □

Theorem 5.1 shows that it is sufficient to approximate the JOP solution in order to obtain an approximation for $ST(i, j)$. The flow analysis algorithm (see figure 3) that we use to approximate the JOP solution is a

Algorithm F:

input: An information propagation problem

$$I = \langle G, LS, \vee, \{f_e\}, \emptyset \rangle$$

output: A mapping $COV: N \rightarrow LS$.

method:

```
begin
 1.  for each node n∈N
 2.  do COV(n):=∅ od
 3.  modify_list:= list of all nodes in N;
 4.  while modify_list ≠ ∅ do
 5.        pop n from modify_list;
 6.        for n'∈Succ(n) do
 7.            save:= COV(n');
 8.            COV(n'):= W(COV(n') ∪ f(n,n')(COV(n)))
 9.            if save ≠ COV(n') and n'∈modify_list
10.                then add n' to modify_list
11.            od
12.       od
13. end
```

Figure 3

modification of Kildall's algorithm [KI73]; it uses the characterization of the least fixpoint of a continuous function as the limit of a sequence of finite approximations in order to compute ST(i, j).

To insure that our algorithm terminates we use a widening operator W [CO76]. For each $n \in N$, the successive values assigned to COV(n) at statement 8 in our algorithm form an ascending chain in LS. The purpose

of the widening operator is to guarantee a finite bound on the length of such chains.

Definition: Let $(L, \vee)$ be an upper semi-lattice. A mapping $W: L \rightarrow L$ is a widening operator if it satisfies the following properties for all x and y of L:

(1) W is monotone, i.e. $x \leq y$ implies $W(x) \leq W(y)$

(2) W is increasing, i.e. $x \leq W(x)$. □

We say that W is finitely convergent if for any ascending chain $u_1 \leq u_2 \leq \ldots$ in L, the ascending chain defined by $v_i = W(u_i)$ for $i \geq 1$ is eventually stable (i.e., $\exists k \geq 0\ [v_i = v_k$ for all $i \geq k]$).

Lemma: When Algorithm F terminates with output $COV: N \rightarrow LS$ we have $COV(n') = W(COV(n') \cup f_{(n,n')}(COV(n)))$ for each edge (n,n') in E.

Proof: Omitted. □

The following theorem shows that, as long as Algorithm F converges, the output mapping $COV: N \rightarrow LS$ always forms a covering of the JOP solution to the information propagation problem I. Such a covering is a "conservative approximation" of the join of paths solution; any property of the parallel program that holds in all states of COV(n) will also hold at node n during program execution.

Theorem: Let $JP: N \rightarrow LS$ be the JOP solution to the information propagation problem $I = \langle G, LS, \cup, \{f_e\}, \emptyset \rangle$. Assume that Algorithm F terminates on I with output $COV: N \rightarrow LS$. Then $JP(n) \subseteq COV(n)$ for all n in N.

Proof: We prove the theorem by contradiction. Assume that the assertion of the theorem is false. By the definition of the function JP, we have

$$JP(n) = \bigcup_{\pi \in Path(n_0, n)} f_\pi(\emptyset) \quad \text{for all} \quad n \in N \ .$$

Thus, there exists a node n and path $\pi \in Path(n_0, n)$ such that $f_\pi(\emptyset) \not\subseteq COV(n)$. Consider such a node-path pair in which the path π has minimal length. It is clear that the path π is not the null path, since $JP(n_0) = \emptyset = COV(n_0)$ always holds. Hence, let $\pi = e_1, e_2, \ldots, e_k$ where $e_k = (n', n)$. Let π' be the subpath of π of length k-1, i.e., $\pi' = e_1, e_2, \ldots, e_{k-1}$. We have $f_{\pi'}(\emptyset) \subseteq COV(n')$ since π' is shorter than π. Since Algorithm F terminates on I, we have $COV(n) = W(COV(n) \cup f_{e_k}(COV(n')))$ by the lemma. Hence $COV(n) \cup f_{e_k}(COV(n')) \subseteq COV(n)$, which in turn implies that $f_{e_k}(COV(n')) \subseteq COV(n)$. Therefore, $f_\pi(\emptyset) = f_{e_k}(f_{\pi'}(\emptyset)) \subseteq f_{e_k}(COV(n')) \subseteq COV(n)$ leads to a contradiction to our choice of π. This proves the theorem. □

To illustrate our flow analysis technique, we examine the behavior of Algorithm F with several different widening operators. The first two examples that we consider are based on the solution to the readers and writers problem shown in figure 2. $I1 = \langle G1, LS, \cup, \{f_e\}, \emptyset \rangle$ will be the corresponding information propagation problem, where G1 is the flow graph in figure 2. Recall that the vector of shared variables is $\bar{x} = (aw, rw, rr)$, and the initial state σ_0 is (0,0,0,1,1).

Example: We first consider the identity widening operator W_0, i.e., $W_0(X) = X$ for all $X \in LS$. In this case, Algorithm F behaves like Kildall's algorithm.

When Algorithm F is applied to I1 with the widening operator W_0, the algorithm terminates, and the output $COV: N \rightarrow LS$ is the join of paths solution to I1:

$$COV(n_{10}) = COV(n_{20}) = \{\sigma_0\},\ COV(n_{11}) = \{\sigma_0, (1,0,0,1,2)\},$$
$$COV(n_{12}) = \{(0,0,1,2,1), (1,0,1,2,2)\},\ COV(n_{21}) = \{\sigma_0, (0,0,1,2,)\},$$
$$COV(n_{22}) = \{(1,0,0,1,2), (1,0,1,2,2)\},\ COV(n_{23}) = \{(1,1,0,1,3)\}.$$ □

Example: Let $W_1 : LS \rightarrow LS$ be the widening operator such that, for $X \in LS$, $W_1(X) = \overset{m+t}{\underset{i=1}{X}} \pi_i(X)$, where $\pi_i(X)$ is the projection of X to the ith component. Hence W_1 collapses both values of shared variables and the program counters. With W_1 Algorithm F produces the following suboptimal approximation of the JOP solution to I1:

$$COV(n_{10}) = COV(n_{20}) = \{\sigma_0\},\ COV(n_{11}) = \{0,1\} \times \{(0,0,1)\} \times \{1,2\},$$
$$COV(n_{12}) = \{0,1\} \times (0,1,2) \times \{1,2\},$$
$$COV(n_{21}) = \{(0,0)\} \times \{0,1\} \times \{1,2\} \times \{1\},$$
$$COV(n_{22}) = \{(1,0)\} \times \{0,1\} \times \{1,2\} \times \{2\}.$$ □

Algorithm F converges on I1 in the above two examples. However, neither W_0 nor W_1 is finitely convergent; with these widening operators Algorithm F may fail to terminate on information propagation problems with unbounded semi-lattices. To demonstrate how unbounded semi-lattices can be handled, we consider the unbounded buffer problem. In this problem two "classes" of processes, called "producer" and "consumer," work on an unbounded buffer. At any moment there can be at most one producer process and at most one consumer process working on the buffer. In order to avoid buffer underflow these processes need to be synchronized. A well-known solution to this problem is as follows: There are three shared variables in the resource UB: np, nc, and b denoting the number of producers, the number of consumers, and the number of portions contained in the buffer respectively. Initially, $\bar{x} = (np, nc, b) = (0,0,0)$.

```
producer:
    repeat
        PR1: with UB when np = 0 do np:= np+1 od;
        produce;
        PR2: with UB when true do np:= np-1; b:= b+1 od
    forever

consumer:
    repeat
        CS1: with UB when nc = 0 & b > 0 do
                    nc:=nc+1; b:=b-1 od
        consume;
        CS2: with UB when true do nc:=nc-1 od
        forever
```

Consider the information propagation $I = < G2, LS, \cup, \{f_e\}, \emptyset >$, where G_2 is described in figure 4. Note that Algorithm F fails to terminate on I2 if either W_0 or W_1 is used. What we need is a widening operator that produces finitely representable coverings for infinite sets.

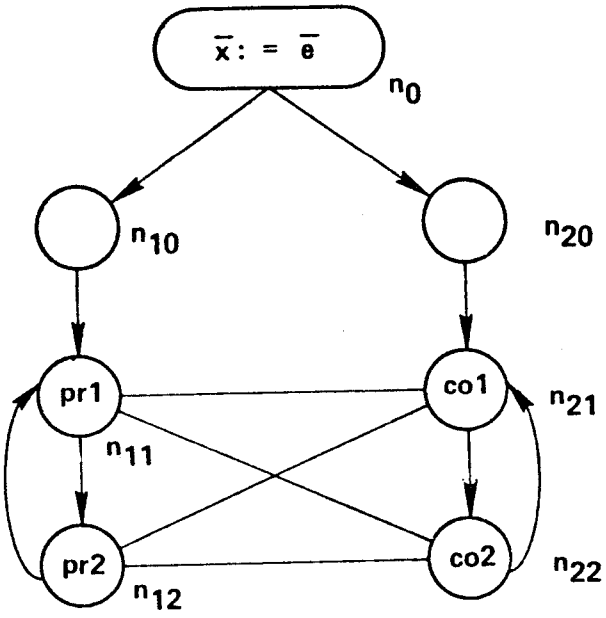

Figure 4

Example: We first consider a widening operator W_I on the semi-lattice $(\mathcal{P}(Z), \cup)$, where $I = [a,b]$ is a finite interval in Z. For each $X \in \mathcal{P}(Z)$ we define $W_I(X) = \bigcup_{r \in X} V_I(r)$, where $V_I(r)$ is $(-\infty, a)$ if $r < a$, $(b, +\infty)$ if $r > b$, and r otherwise.

For a general parallel program with m variables and t processes, we define a widening operator $W_{I_1, \ldots, I_m}: LS \to LS$, where $I_1, \ldots, I_m$ are given finite intervals in Z, such that

$$W_{I_1, \ldots, I_m}(X) = \left\{ \bigtimes_{i=1}^{m} W_{I_i}\left[val_i(X)\right] \right\} \times \left[\bigtimes_{j=1}^{t} pc_j(X) \right].$$

This widening operator not only collapses the values of each shared variable, but also widens the collapsed value sets into intervals. The intervals $I_1, \ldots, I_m$ may be chosen by examining the program text. One heuristic for this purpose is as follows:

Let a_i and b_i be the maximum and minimum constants which are assigned to or compared with x_i in the program text. Let c_i (d_i, resp.) be the maximum (minimum, resp.) integer k such that $x_i := x_i + k$ appears in the program text. Then we choose I_i as the interval $[\min(a_i, a_i + d_i), \max(b_i, b_i + c_i)]$.

If we apply the above heuristic to the parallel program for the unbounded buffer problem, we get $I_1 = I_2 = I_3 = [-1, 1]$. Using the widening operator W_{I_1, I_2, I_3} we obtain the approximation:

$$COV(n_0) = \emptyset,\ COV(n_{10}) = COV(n_{20}) = \{\sigma_0\},$$
$$COV(n_{11}) = \{0\} \times [0,1] \times [0, \infty) \times \{1\} \times \{1,2\},$$
$$COV(n_{12}) = \{1\} \times [0,1] \times [0, \infty) \times \{2\} \times \{1,2\},$$
$$COV(n_{21}) = [0,1] \times \{0\} \times [0, \infty) \times \{1,2\} \times \{1\},$$
$$COV(n_{22}) = [0,1] \times \{1\} \times [0, \infty) \times \{1,2\} \times \{2\}.$$

□

In the above example we have implicitly assumed an efficient interval calculus [CO76]. The interval calculus is usually easy to implement: $[a,b] + [c,d] = [a+c, b+d]$, $[a,b]\cdot(-1) = [-b,-a]$, etc. We will not discuss interval calculus further in this paper, however.

In defining the flow graph for a parallel program we consider all possible interprocess edges. However, many of these interprocess edges never occur in any execution path of the program. In an actual implementation of the flow analysis algorithm such interprocess edges need never be explicitly constructed. Currently, the only place in Algorithm F where the edges are used is in the for-loop "for $n' \in$ Succ(n) do" which begins at statement 6. Consider, for example, the assignment statement at statement 8. When n represents a conditional critical region, say C_{ij}, the only states that can appear in $f_{(n,n')}(COV(n))$ must result from a state $\sigma \in COV(n)$ such that $pc_i(\sigma) = j$. Thus, the for-loop "for $n' \in$ Succ(n) do ... od" may be implemented as

```
for i=1 to t do /* t is # of processes */
    for j∈pc_i(COV(n)) do
        Let n' be the node n_ij;
        ... od
    od
```

This implementation eliminates many of the references to redundant edges and increases the efficiency of the algorithm.

For each node n the successive values assigned to COV(n) is an ascending chain. Assume that there is a finite upper bound d on the lengths of such chains. Then a worst case bound on the time complexity of Algorithm F is $0(|G| * d)$ extended steps, where an extended step is the work required to calculate a function application $f_e(X)$ or to do a widening

operation. As the examples demonstrate, the extended steps can be efficiently implemented if an appropriate interval calculus is used. The bound d on the lengths of ascending chains can also be kept low if proper widening operators are chosen. For instance, suppose the widening operator $W_{I_1, \ldots, I_m}$ is used in Algorithm F, then d is bounded by $(|I_1| + \ldots + |I_m| + k_1 + \ldots + k_t)$ (recall that k_j is the number of conditional critical regions in the process P_j).

By using more sophisticated widening operators it should be possible to get faster convergence and to obtain even better approximate solutions. Various propagating sequences [RE79] can also be constructed to further limit the number of interprocess edges considered in the flow analysis.

SECTION 6

CONSTRUCTION OF THE SCHEDULER

In this section we show how the information generated by the flow analysis in Section 5 can be used to construct a good scheduler for conditional critical regions. Our scheduler is an extension of the one described by Schmid [SC76].

In Schmid's implementation (see figure 5) the conditional critical regions in a program are divided into "equivalence classes" so that all regions with the same condition and instruction part are placed in the same class. For each class the ea and da relations are constructed using heuristics about how the execution of one critical region affects the condition of another. This part of Schmid's method heavily depends on the

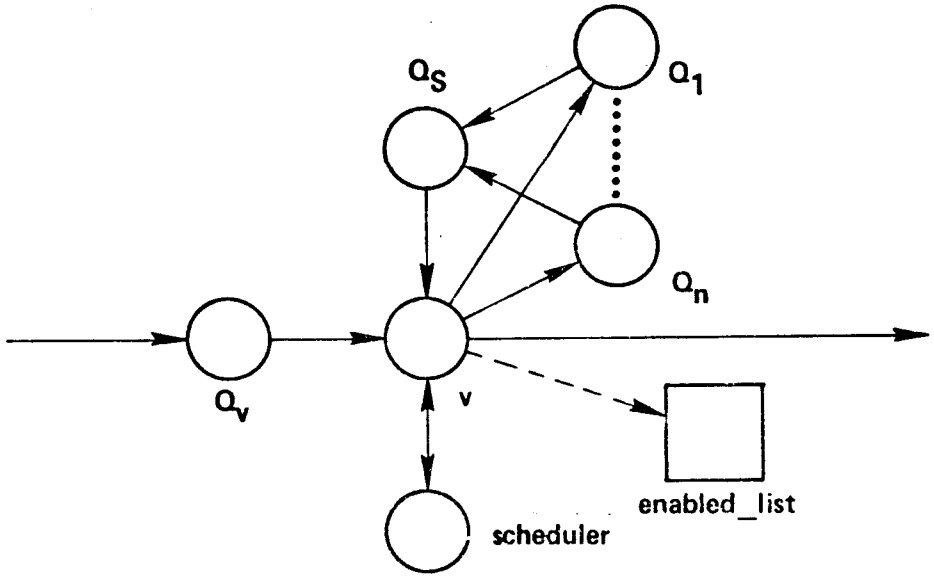

Figure 5

restriction that conditions are linear inequalities and that assignment statements only increment (or decrement) shared variables. During execution of a program, all processes that want to enter a conditional critical region are entered in the queue Q_v. When a process has entered a region CCR_i and found that the condition is not satisfied, the process is placed in the queue Q_i, together with all other processes that want to execute a region of the same equivalence class. When a process has executed a region CCR_i, it enters into the enabled_list all (classes of) "regions" CCR_j for which $(CCR_i, CCR_j) \in ea$ holds.

The scheduling of conditional critical executions is controlled by a special process called the "scheduler" in Schmid's implementation. The "scheduler" examines whether the condition for a class in enabled_ list is true. If this is the case, it transfers the processes that are waiting in Q_i for that class to the queue Q_s. If $(CCR_i, CCR_i) \in da$, then only one of the waiting processes is transferred; otherwise they all are. The queue Q_s has the highest priority; a process can only enter a critical region if there is no process ahead of it on Q_s. Note that it is unnecessary to retest the condition of a process on Q_s before starting execution.

We will use the same system diagram that Schmid used (figure 5) to describe our implementation. For simplicity we will only consider the case in which every conditional critical region belongs to a distinct equivalence class. The modification of our scheduling algorithm to handle the case in which equivalence classes may contain multiple critical regions will be discussed in a future paper.

In our implementation the minimal enable relation ea (disable relation da) for a program P is decomposed into two disjoint relations sea and wea (sda and wda, resp.). The prefix letter "s" ("w") stands for "strong"

("weak") and indicates that the corresponding property occurs in all possible (in only some) executions of the program. The four relations sea, wea, sda, and wda can be formally defined as follows:

1. $(CCR_i, CCR_j) \in$ sea iff there is no computation of the form $\sigma_0 \overset{P}{\Longrightarrow} \sigma_k \overset{CCR_i}{\Longrightarrow} \sigma_{k+1}$ such that condition b_j of critical region CCR_j is false in state σ_k and also in σ_{k+1}.

2. wea = ea - sea.

3. $(CCR_i, CCR_j) \in$ sda iff there is no computation of the form $\sigma_0 \overset{P}{\Longrightarrow} \sigma_k \overset{CCR_i}{\Longrightarrow} \sigma_{k+1}$ such that condition b_j of critical region CCR_j is true in state σ_k and also in σ_{k+1}.

4. wda = da - sda.

The information obtained by the flow analysis in Section 5 can be used in a straightforward manner to approximate the six enable and disable relations. Let G be a flow graph for a parallel program in which conditional critical regions are indexed as described in Section 5. Assume that after executing Algorithm F we obtain the approximation COV: $N \rightarrow LS$ for the JOP solution to the information propagation $I = \langle G, LS, \cup, \{f_e\}, \emptyset \rangle$. Let n and n' be essential nodes representing two critical regions C_{iq} and C_{jr} which occur in different processes and are therefore connected by edge $e = (n, n')$ in the flow graph. To simplify our notation, we identify a predicate with the set of states which make it true.

A. Put (C_{iq}, C_{jr}) in ea if there exists a state $\sigma \in COV(n) \cap b_{iq} \cap \bar{b}_{jr}$ such that $\sigma' = f_e(\sigma) \in b_{jr}$ and $pc_j(\sigma') = r$.

B. Put (C_{iq}, C_{jr}) in sea if there does not exist a state $\sigma \in COV(n) \cap b_{iq} \cap \bar{b}_{jr}$ such that $\sigma' = f_e(\sigma) \in \bar{b}_{jr}$ and $pc_j(\sigma') = r$.

C. Put (C_{iq}, C_{jr}) in wea if $(C_{iq}, C_{jr}) \in$ ea - sea.

D. Put (C_{iq}, C_{jr}) in da if there exists a state $\sigma \in COV(n) \cap b_{iq} \cap b_{jr}$ such that $\sigma' = f_e(\sigma) \in \bar{b}_{jr}$ and $pc_j(\sigma') = r$.

E. Put (C_{iq}, C_{jr}) in sda if there does not exist a state $\sigma \in COV(\sigma) \cap b_{iq} \cap b_{jr}$ such that $\sigma' = f_e(\sigma) \in b_{jr}$ and $pc_j(\sigma') = r$.

F. Put (C_{iq}, C_{jr}) in wda if $(C_{iq}, C_{jr}) \in$ da - sda.

Note that efficient computation of the four enable and disable relations also requires the use of interval arithmetic.

During execution the system works as follows: A process that wants to enter a critical region is placed in Q_v. When a process has entered a region CCR_i and found out that the condition for CCR_i does not hold, the process is placed in Q_i. When a process has executed a region CCR_i and leaves it, the system will move a process P_k to Q_s if there is a process P_k waiting in Q_j such that $(CCR_i, CCR_j) \in$ sea. Then all other critical regions CCR_h with $(CCR_i, CCR_h) \in$ ea are pushed on enabled_ list. At the same time all critical regions CCR_j on enabled_list with $(CCR_i, CCR_j) \in$ sda are removed from enabled_list. The scheduler still works the same way as described in [SC76].

Consider, for example, a solution to the readers and writers problem with two reader processes and one writer process:

```
x̄: = (0,0,0);   /* x̄ = (aw, rw, rr) */
resource R(x̄)
cobegin
      Reader_1
    //
      Reader_2
    //
      Writer_1
coend
```

If we apply Algorithm F to this problem with the widening operator W_0, we get the JOP solution:

$$COV(n_0) = \emptyset,\ COV(n_{10}) = COV(n_{20}) = COV(n_{30}) = \{\sigma_0\},$$

$$COV(n_{11}) = \{\sigma_0, \sigma_1, \sigma_2, \sigma_3, \sigma_4\},\ COV(n_{12}) = \{\sigma_5, \sigma_6, \sigma_7, \sigma_8\},$$

$$COV(n_{21}) = \{\sigma_0, \sigma_1, \sigma_2, \sigma_5, \sigma_7\},\ COV(n_{22}) = \{\sigma_3, \sigma_4, \sigma_6, \sigma_8\},$$

$$COV(n_{31}) = \{\sigma_0, \sigma_3, \sigma_5, \sigma_6\},\ COV(n_{32}) = \{\sigma_1, \sigma_4, \sigma_7, \sigma_8\},$$

where

$$\sigma_0 = (0,0,0,1,1,1),\ \sigma_1 = (1,0,0,1,1,2),$$

$$\sigma_2 = (1,1,0,1,1,3),\ \sigma_3 = (0,0,1,1,2,1),$$

$$\sigma_4 = (1,0,1,1,2,2),\ \sigma_5 = (0,0,1,2,1,1),$$

$$\sigma_6 = (0,0,2,2,2,1),\ \sigma_7 = (1,0,1,2,1,2),$$

$$\sigma_8 = (1,0,2,2,2,2).$$

The method described earlier in this section may be applied to the JOP solution to determine the enable and disable relations:

$ea = \{ (W3,R1), (R2,W2), (W3,W2) \}$

$sea = \{ (W3,R1), (W3,W2) \}$

$wea = \{ (R2,W2) \}$

$da = \{ (W1,R1), (R1,W2), (W2,W2) \}$

$sda = da, \quad wda = \emptyset$.

This information, in turn, can be used to construct the efficient implementation for the program shown in figure 6. Q_1 (Q_2, resp.) is the waiting queue for the conditional critical region R1 (W2, resp.). The procedures enter, unblock_one, and unblock_all are used to access the two waiting queues.

```
reader:   repeat
              R1: with  R  do
                        if not  aw = 0  then enter(Q1);
                        rr: = rr+1
                        od;
              read;
              R2: with  R  do
                        rr: = rr-1;
                        if  rw = 0  &  rr = 0  then unblock_one(Q2)
                        od
          forever
```

Figure 6 (continued)

```
writer:   repeat
              W1: with R do
                  aw: = aw+1;
                  remove R1 from enabled_list
                  od
              W2: with R do
                  if not (rr=0 & rw=0) then enter(Q2)
                  rw: = rw+1;
                  remove W2 from enabled_list
                  od
              write;
              W3: with R do
                  rw: = rw-1; aw: = aw-1;
                  if aw = 0 then unblock_all(Q1)
                  else unblock_one(Q2)
                  od
          forever
```

Figure 6 (concluded)

SECTION 7

CONCLUSIONS

We have demonstrated how global flow analysis techniques can be used to reduce the incidence of busy waiting in conditional critical regions. We believe that similar techniques may be applicable to other important problems in parallel computation. In particular, it should be fairly easy to modify the flow analysis algorithm presented in sections 4 and 5 so that it can be used to detect deadlock and to prove mutual exclusion of statements in parallel processes. If stronger widening operators are used [CL77], our algorithm should also be useful in the automatic verification of parallel programs.

REFERENCES

[AC76] Allen, F.E., J. Cocke. A Program Data Flow Analysis Procedure. CACM 19:3, 137-147.

[AL70] Allen, F.E. Control Flow Analysis. SIGPLAN Notices 5:7, 1-19.

[AU73] Aho, A.V., J.D. Ullman. The Theory of Parsing, Translation and Compiling, Vol. II: Compiling. Prentice Hall, Englewood Cliffs, NJ.

[BH73] Brinch Hansen, P. Operating System Principles. Prentice Hall, Englewood Cliffs, N.J.

[CL77] Clarke, E.M. Synthesis of Resource Invariants for Concurrent Programs. 6th POPL Conference, January 1979.

[CO76] Cousot, P., R. Cousot. Static Determination of Dynamic Properties of Programs. Proc. 2nd International Symposium on Programming, B. Robinet, Ed., Dunod, Paris, April 1976.

[GW76] Graham, S.L., M. Wegman. A Fast and Usually Linear Algorithm for Global Flow Analysis. JACM 23:1, January 1976, 172-202.

[HO72] Hoare, C.A.R. Towards a Theory of Parallel Programming. In: Hoare, C.A.R., R.H. Perrot, Eds., Operating System Techniques. London, Academic Press, 1972, 61-71.

[HO73] Hoare, C.A.R. Monitors: An Operating System Structuring Concept. IFIP-WG 2.3, Munich 1973.

[HU75] Hecht, M.S., J.D. Ullman. A Simple Algorithm for Global Data Flow Analysis Programs. SIAM J. Computing 4:4, 519-532.

[KE71] Kennedy, K. A Global Flow Analysis Algorithm. Int'l. J. Computer Math. 3, 5-15.

[KI73] Kildall, G.A. A Unified Approach to Program Optimization. Proc. ACM Symp. on Principles of Programming Languages, 1973.

[KL77] Keller, R. M. Generalized Petri Nets as Models for System Verification, Computer Science Dept. Technical Report, University of Utah, 1977.

[KU76] Kam, J. B., J. D. Ullman. Monotone Data Flow Analysis Frameworks, Acta Informatica 7:3, 305-318.

[LD79] Liu, Lishing and A. Demers. Unpublished manuscript.

[LI76] Lipton, R. The Reachability Problem and Boundedness Problem for Petri Nets in Exponential-space Hard. Conf. on Petri Nets and Related Methods, MIT, July 1975.

[MI67] Minsky, M. L. Computation: Finite and Infinite Machines. Prentice Hall, 1967.

[OW76] Owicki, S., D. Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach. CACM 19:5, 279-284, 1976.

[RE79] Reif, J. Data Flow Analysis of Communicating Processes. 6th POPL, January 1979.

[SC76] Schmid, H. A. On the Efficient Implementation of Conditional Critical Regions and Construction of Monitors. Acta Informatica, 6, 227-249, 1976.