OPTIMIZATION OF BUSY WAITING IN CONDITIONAL CRITICAL REGIONS†

by

Lishing Liu*

Edmund M. Clarke**

TR-16-79

†This research was supported in part by National Science Foundation Grant No. MCS-79-08365.

*Author's address: Mitre Corporation, Bedford, Mass. 01730

**Author's address: Aiken Computation Lab., Harvard University, Cambridge, Mass. 02138

OPTIMIZATION OF BUSY WAITING IN CONDITIONAL CRITICAL REGIONS

Lishing Liu
Mitre Corporation
Bedford, Mass. 01730

Edmund M. Clarke*
Harvard University
Cambridge, Mass. 02138

Abstract

Standard implementations of conditional critical regions and monitors can lead to "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). This information may be used to obtain more efficient scheduling algorithms.

1. INTRODUCTION

Many different synchronization primitives have been proposed for controlling access to shared resources in parallel programs. A particularly powerful example is the conditional critical region ([HO72], [BH73]) which permits processes to wait on arbitrary boolean conditions. With this primitive variables which must be accessed by more than one process are grouped together as resources. Individual processes access resources in critical regions of the form "with R when b do A od", where R is a resource name, b is a boolean expression, and A is a block of statements. When a conditional critical region is reached during execution of a process, the process is delayed until no other process is using R and the condition b is true. The block A is then executed with exclusive use of the resource.

The standard implementation ([BH73]) of conditional critical regions uses two queues for each resource R: a "main queue" R_m and a "wait queue" R_w. Each process wishing to enter a critical region for R is first placed on R_m. Processes on R_m are allowed to enter their critical regions one at a time and examine whether the entry condition b is satisfied. If so, the process completes execution of the block A; otherwise, the process releases the resource and is put on R_w. Each time a process completes the execution of a critical region for R all processes waiting on R_w must be moved to R_m, since the execution of the body of a critical region may change the entry conditions of some of the waiting processes from false to true. Note that in this implementation a process may be transferred back and forth many times between R_m and R_w before its entry condition finally becomes true.

* This work was partially supported by NSF Grant MCS-7908365.

This busy waiting is "the price we pay for the conceptual simplicity achieved by using arbitrary boolean expressions as synchronizing conditions [BH73]". Essentially the same type of busy waiting can occur with monitors [HO73] if monitor procedures are allowed to wait on arbitrary boolean conditions.

More efficient implementations of conditional critical regions are possible if programs are preprocessed to obtain scheduling information. Consider, for example, the unbounded buffer problem. In this problem a class of producer processes and a class of consumer processes communicate by means of an unbounded buffer. Synchronization is necessary to avoid buffer underflow and to insure that only one process from each class is working on the buffer at a given time. A well-known solution is as follows: the shared resource UB contains three variables np, nc, and p indicating respectively the number of active producers, the number of active consumers, and the number of full slots currently in the buffer; initially all are zero. Each producer is of the form

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

and each consumer process has the form

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

Suppose that a consumer process "consumer1" is waiting on R_w and another consumer process "consumer2" is executing the body of the critical region CS1. After the execution of CS1 by consumer2 it is unnecessary to move consumer1 from R_w to R_m since nc>0 at this point. On the other hand, the execution of PR2 always makes np = 0, and hence a producer waiting to execute PR1 can be activated without testing the entry condition.

A number of heuristics have been developed by Schmid [SC76] for determining which conditional critical regions are enabled by the execution of a given conditional critical region. The heuristics assume that each conditional critical region has the form

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

and use only local information. Thus, Schmid's technique does not easily generalize to more complicated critical regions where, for example, the entry condition is an equality $a_1x_1 + \ldots a_nx_n + a_{n+1} = 0$. Also since only local information is used, his technique does not reveal in the case of the unbounded buffer problem that execution of PR2 <u>always</u> enables PR1 and that execution of CS1 <u>always</u> disables CS1.

In this paper we show how global data flow analysis can be used to obtain useful scheduling information for parallel programs where processes wait on arbitrary boolean conditions. Our technique does not depend on a particular syntax for the boolean conditions and is able to exploit the global structure of the parallel program. As with most optimization techniques we cannot claim that our scheduling algorithm is optimal. However, it can be shown that our algorithm is <u>conservative</u>, i.e. our algorithm never fails to determine that an enabled conditional critical region is really enabled.

In Section 2 we describe a simple parallel programming langauge in which processes access shared data via conditional critical regions. In Section 3 we present the basic ideas of global data flow analysis and show how flow graphs can be constructed for parallel programs. We also show in Section 3 how Kildall's algorithm can be modified to obtain information about which conditional critical regions may be enabled by the execution of a given conditional critical region. We show in Section 4 how this information can be used to obtain more efficient scheduling algorithms. In Section 5 we briefly discuss some of the problems involved in extending our results to monitors and in making the flow analysis more efficient.

2. PARALLEL PROGRAMS

A parallel program will consist to two parts: an <u>initialization part</u> "$\bar{x}{:=}\bar{e}$" in which values are assigned to the program variables $\bar{x}$, and a <u>parallel execution part</u> "<u>resource</u> $R(\bar{x})$:<u>cobegin</u> P1//P2//...Pt <u>coend</u>" which permits the simultaneous or interleaved execution of the statements in the <u>processes</u> P1,...,Pt. Each process Pi is a sequential program composed of simple Algol statements (assignment, conditional, while, etc.) and conditional critical regions. The conditional critical regions have the form "<u>with</u> R <u>when</u> b <u>do</u> A <u>od</u>", where b is a boolean expression and A is a sequence of assignment statements. Shared variables can only be accessed within critical regions and must be listed in the prefix $R(\bar{x})$ of the parallel execution part. This syntax is illustrated by the unbounded buffer program shown below. Here, each of the m producer processes and n consumer processes has the syntax described in Section 1.

```
np,nc,b:=0,0,0;
resource R(np,nc,b):
cobegin
    producer_1//producer_2//...//producer_m//
    consumer_1//consumer_2//...//consumer_n
coend
```

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. A <u>state</u> of P is 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 <u>program counter</u> for process Pi. Thus, $\sigma_o = (\bar{e},1,\ldots,1)$ denotes the state of the program after the initialization part $\bar{x}{:=}\bar{e}$ has been executed. The function $pc_i(\sigma)$ gives the value of the ith program counter in state σ, and the function $val(x,\sigma)$ gives the value of shared variable x in state σ. Both functions are extended in the natural manner to apply to sets of states. We write $\sigma \overset{C_i}{\Rightarrow} \sigma'$ if critical region C_i can be executed in state σ to produce state σ'. Likewise, $\sigma_o \overset{P}{\Rightarrow} \sigma$ indicates that state σ can occur during execution of program P on initial state σ_o. We use Σ to denote the set of all possible states for the program P.

Let CCR be the set of conditional critical regions in program P. An <u>enable relation</u> 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 $\sigma_o \overset{P}{\Rightarrow} \sigma_k \overset{C_i}{\Rightarrow} \sigma_{k+1} \overset{C_j}{\Rightarrow} \sigma_{k+2}$ and condition b_j of region C_j is false in state σ_k. Similarly, we may define a <u>disable relation</u> da for P such that $(C_i,C_j) \in$ da whenever there exists a computation $\sigma_o \overset{P}{\Rightarrow} \sigma_k \overset{C_i}{\Rightarrow} \sigma_{k+1}$ with b_j true in σ_k but not in σ_{k+1}.

For the unbounded buffer program we see by inspection that any enable relation must be a superset of $ea_{min} = \{(PR1,PR1), (PR2,CO1), (CO2,CO2)\}$. Similarly, any disable relation must be a superset of $da_{min} = \{(PR1,PR1), (CO1,CO1)\}$. In [CL79b] we show that, in general, it is extremely difficult to exactly compute the minimal enable and disable relations for a given parallel program P. By using techniques from global flow analysis, however, it is possible to find good approximations for these relations.

3. GLOBAL DATA FLOW ANALYSIS FOR PARALLEL PROGRAMS

In data flow analysis, programs are modeled by directed graphs where the nodes corespond 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 <u>successor</u> of e and n a <u>predecessor</u> 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 <u>path from</u> n_0 to n_k of length k. A path of length o is called a <u>null path</u>. Path(n,n') will denote the set of all paths from n to n'.

3.1 DEFINITION: A <u>flow graph</u> $G=(N,E,n_0)$ is a finite directed graph (N,E) together with a distinguished <u>entry node</u> n_0 such that n_0 has no predecessor and $\text{path}(n_0,n)\neq\phi$ for all $n\in N$. □

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

3.2 DEFINITION: An <u>information propagation problem</u> is a triple $<G,F,x_0>$ where

(1) $G=(N,E,n_0)$ is a flow graph,

(2) $F=\{f_e:\mathscr{P}(\Sigma)\to\mathscr{P}(\Sigma)\mid e\in E\}$ is a set of transition functions, and

(3) $x_0\in\Sigma$ is the initial information attached at node n_0. □

A transition function $f_e:\mathscr{P}(\Sigma)\to\mathscr{P}(\Sigma)$, 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.

Let P be a parallel program with processes $P_1, P_2,\ldots,P_t$. Assume that $C_{i_1},C_{i_2},\ldots,C_{i_{k_i}}$ are the critical regions which occur in process P_i and that each C_{ij} has the form "<u>with</u> R <u>when</u> b_{ij} <u>do</u> A_{ij} <u>od</u>", where A_{ij} consists of assignment statements. Let $ST(i,j)=\{\sigma\mid\sigma_0\overset{P}{\Rightarrow}\sigma,\ pc_i(\sigma)=j\}$ be the set of program states which may preceed 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. $E=\bigcup_{i=1}^{t} E_i\cup\{(n_0,n_{i0})\mid 1\leq i\leq t\}\cup E'$ where

   $E'=\{(n_{iq},n_{jr})\mid i\neq j$ and n_{iq},n_{jr} represent conditional critical regions$\}$.

The flow graph for the unbounded buffer program with one producer process and one consumer process is shown in Figure 1.

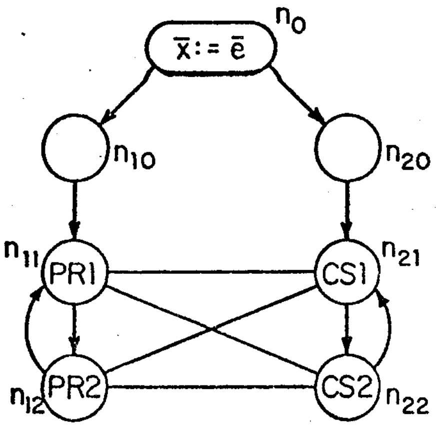

Figure 1

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.

Transition functions $f_e:\mathscr{P}(\Sigma)\to\mathscr{P}(\Sigma)$ for a flow graph G may be defined as follows: with each node n of G we associate a function $f_n:\mathscr{P}(\Sigma)\to\mathscr{P}(\Sigma)$ such that $f_n(X)=\{post(n,\sigma)\mid\sigma\in X$ and the instruction at n is executable under $\sigma\}$. Here, $post(n,\sigma)$ is the state which results when the instruction at node n is executed in state σ. Thus, if n represents a conditional critical region C_{ij}, $f_n(X)=\{A_{ij}(\sigma)\mid\sigma\in X$ and $b_{ij}(\sigma)=\text{true}\}$. For an edge $e=(n,n')$ where $n'=n_{rq}$ the function $f_e:\mathscr{P}(\Sigma)\to\mathscr{P}(\Sigma)$ is defined by $f_e(X)=\{\sigma\in f_n(X)\mid pc_r(\sigma)=q\}$.

The flow analysis algorithm that we use (see Fig. 2) is a modification of <u>Kildalls' algorithm</u> [KI73]. To insure that our algorithm terminates we use a <u>widening operator</u> W [CO76]. For each $n\in N$, the successive values assigned to COV(n) at statement 9

Algorithm F

input: An information propagation problem $I = \langle G, \{f_e\}, \phi \rangle$

output: A mapping $COV: N \to \mathscr{P}(\Sigma)$

method:

```
    begin
1.      for each node $n \in N$ do $COV(n) := \phi$ od;
2.      modify_list:=list of all nodes in N;
3.      while modify_list $\neq \phi$ do
4.          pop n from modify_list;
5.          for i = 1 to t do
6.              for $q \in pc_i(COV(n))$ do
7.                  let n' be the node $n_{iq}$;
8.                  save:=COV(n');
9.                  $COV(n') := W(COV(n') \cup f_{(n,n')}(COV(n)))$;
10.                 If save $\neq$ COV(n') and n' $\notin$ modify_list
11.                     then add n' to modify_list;
12.             od
13.         od
14.     od
15. end
```

FIGURE 2: Flow Analysis Algorithm

form an ascending chain in $\mathscr{P}(\Sigma)$. The purpose of the widening operator is to guarentee a finite bound on the length of such chains.

3.3 DEFINITION: A mapping $W: \mathscr{P}(\Sigma) \to \mathscr{P}(\Sigma)$ is a widening operator if it satisfies the following properties for all X and Y in $\mathscr{P}(\Sigma)$:

(1) W is monotone, i.e. $X \subseteq Y$ implies $W(X) \subseteq W(Y)$

(2) W is increasing, i.e. $X \subseteq W(X)$. □

We say that W is finitely convergent if for any ascending chain $U_1 \subset U_2 \subset \ldots$ in $\mathscr{P}(\Sigma)$, the chain defined by $V_i = W(U_i)$ for $i \geq 1$ is eventually stable (i.e., $\exists K > 0[V_i = V_K$ for all $i \geq K])$.

In [CL79b] we show that $ST(i,j) \subseteq COV(n_{ij})$ for critical regions n_{ij} of the parallel program. This means that our algorithm always produces "conservative approximations" for the state sets ST(i,j). Thus, any property of the parallel program that holds for all states in COV(n) must also hold at node n during the actual execution of the program. Because of this fact our scheduling algorithm will never fail to determine that an enabled process is really enabled.

The usefulness of algorithm F strongly depends on the widening operator that is used. If the widening operator is very difficult to compute, our optimization techniques may become prohibitively expensive. On the other hand, a widening operator that throws away too much information will cause the scheduler to make many redundant queue operations. The widening operator that we describe below uses intervals to represent sets of states and appears to avoid both of these problems.

We first consider a widening operator W_I for $\mathscr{P}(Z)$, where Z is the set of integers and $I = [a,b]$ is a finite interval in Z. For each $X \in \mathscr{P}(Z)$ 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 $W_{I_1,\ldots,I_m}: \mathscr{P}(\Sigma) \to \mathscr{P}(\Sigma)$, where $I_1,\ldots,I_m$ are given finite intervals in Z, such that

$$W_{I_1,\ldots,I_m}(X) = \mathop{X}_{i=1}^{m} W_{I_i}(val_i(X)) \times \mathop{X}_{j=1}^{t} pc_j(X) \quad .$$

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 (mimimum, 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:

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

Note that the flow analysis algorithm determines exact values for the variables np and nc.

In the above example we have implicitly assumed an efficient interval calculus [CO76]. The interval calculus is usually easy to implement: $[a,b] + [c,d] = [a+c, b+d]$, $[a,b](-1) = [-b,-a]$, etc. We will not discuss interval calculus further in this paper, however. By using more sophisticated widening operators it should be possible to get faster convergence and to obtain even better approximation solutions. Various propagating sequences [RE79] can also be constructed to further limit the number of interprocess edges considered in the flow analysis.

4. CONSTRUCTION OF THE SCHEDULER

Our scheduler is an extension of the one described by Schmid [SC76]. In Schmid's implementation (see Fig.3) the conditional critical regions in a program are divided into "equivalence classes" so that all regions with the same condition are placed in the same class. The ea and da relations among these equivalent classes are constructed using heuristics about how the execution of one critical region affects the condition of another. 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,

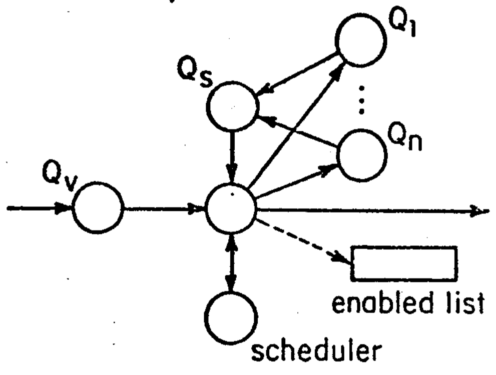

Figure 3

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 "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.

We will use the same diagram that Schmid used (Fig. 3) to describe our implementation. In our implementation an 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 $\sigma_0 \overset{P}{\Rightarrow} \sigma_k \overset{CCR_i}{\Rightarrow} \sigma_{k+1}$ 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 $\sigma_0 \overset{P}{\Rightarrow} \sigma \overset{CCR_i}{\Rightarrow} \sigma_{k+1}$ 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 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 3. Assume that after executing Algorithm F we obtain the approximation $COV:N \to \mathcal{P}(\Sigma)$ for the solution to the information propagation problem $<G,\{f_e\},\phi>$. 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. For each conditional critical region C, we use $\tilde{C}$ to denote the equivalence class of C.

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

D. Put $(\tilde{C}_{iq},\tilde{C}_{jr})$ 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 $(\tilde{C}_{iq},\tilde{C}_{jr})$ in sda if there does not exist a state $\sigma \in COV(\sigma) \cap b_{iq} \cap b_{jr}$ such that $\sigma' = fe(\sigma) \in b_{jr}$ and $pc_j(\sigma') = r$.

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

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].

To illustrate how the scheduling algorithm works we consider the conditional critical region solution to the <u>dining philosophers' problem</u> [BH73]. In this problem five philosophers, numbered 0-4, sit around a circular dining table (see Fig. 4). In front of each philosopher there is a plate of spaghetti. When a philosopher wishes to eat, he picks up the two forks next to his plate. Since there are five forks on the table, a philosopher can only eat when none of his neighbors are eating. In the conditional critical region solution each

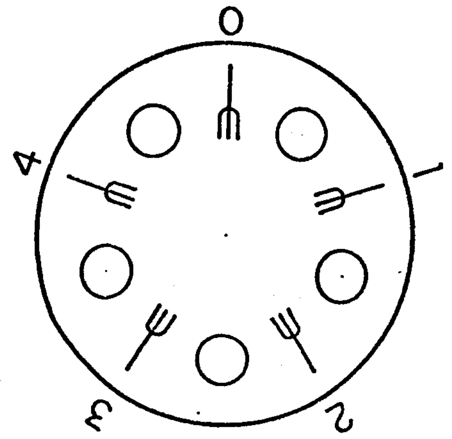

Figure 4

dining philosopher is represented by a process, and a shared array fork(0:4) is used to indicate the status of the forks:

```
Resource R(fork):
Cobegin D1//...//D5 coend
```

When philosopher i is ready to eat, he must simultaneously pick up the two forks immediately adjacent to his plate:

```
Di:repeat
   Di1:with R when fork(i) = 0 ∧ fork(i+) = 0 do
                fork(i):=1; fork(i+):=1 od
       eat;
   Di2:with R when true do
                fork(i):=0; fork(i+):=0 od
   forever
```

(i^+ and i^- denote the mod 5 successor of i and the mod 5 predecessor of i, respectively).

The flow graph for this parallel program is shown in Figure 5. In order to simplify the diagram we have not included the edges connecting critical regions in different processes.

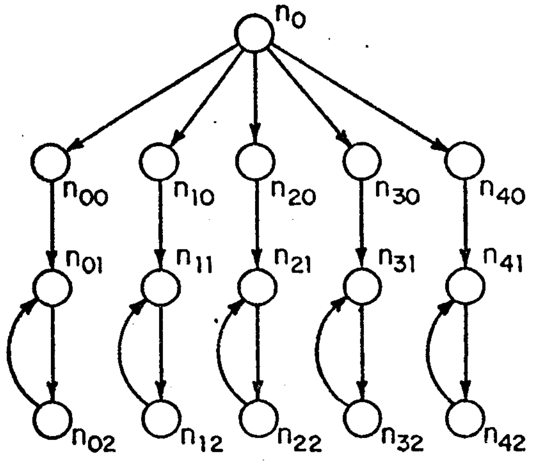

Figure 5

Each state σ of the program is a 10-tuple of the form $\sigma = (f_0,\ldots,f_4,p_0,\ldots,p_4)$ where $1 \le p_i \le 2$ is the program counter for process Di. The set Σ_D of all such states is clearly finite. Thus, a simpler widening operator can be used in this example than was used in the unbounded buffer example. We use:

$$W(X) = \mathop{X}_{j=1}^{10} \pi_j(X) \quad \text{where } \pi_j \text{ is the projection of X}$$

to the ith component. When algorithm F is applied with this widening operator we obtain the covering shown below:

$$COV(n_0) = \phi$$
$$COV(n_i) = \{(0,0,0,0,0,1,1,1,1,1,)\}$$
$$COV(n_{i1}) = \{\sigma \in \Sigma_D | \sigma = (f_0,\ldots,f_4,p_0,\ldots,p_4) \text{ and } p_i=1\}$$
$$COV(n_{i2}) = \{\sigma \in \Sigma_D | \sigma = (f_0,\ldots,f_4,p_0,\ldots,p_4),\ p_i = 2,\ p_{i^-} = p_{i^+} = 1, \text{ and } f_i = f_{i^+} = 1\}$$

The method described earlier in this section can then be used to compute the enable and disable relations for the program. The resulting relations are summarized in Figure 6 where sda, wea, and sea are represented by using <———>, ----->, and ····> respectively.

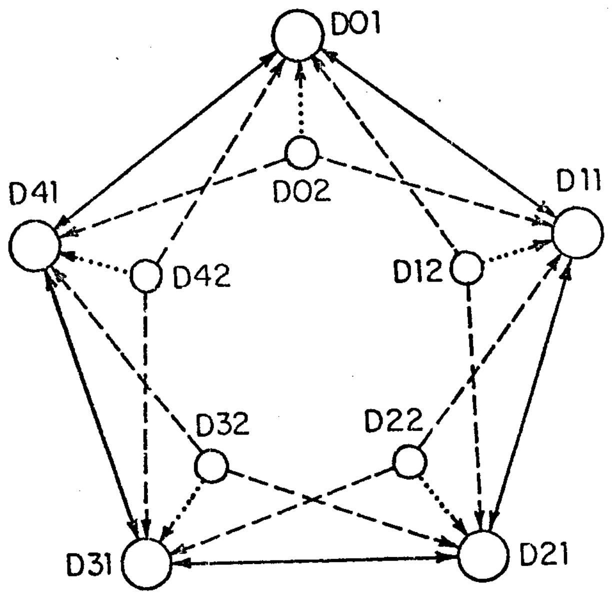

Figure 6

This information, in turn, can be used to construct the efficient implementation for the dining philosopher's problem shown in Figure 7. The procedures enter and remove are used to access the various queues used by the implementation. The scheduler will only check the conditions of critical regions on the enabled list when determining which process to run next.

```
Di:repeat
   Di1:with R do
      if not $(\text{fork}(i) = 0 \wedge \text{fork}(i^+) = 0)$
      then enter Di1 on wait queue;
      $\text{fork}(i) := 1$; $\text{fork}(i^+) := 1$;
      remove $Di^{+}1$ and $Di^{-}1$ from the enabled_list
      od;
   eat;
   $D_{i2}$:with R do
      $\text{fork}(i) := 0$; $\text{fork}(i) := 0$;
      enter $Di^{-}1$, Di1, and $Di^{+}1$ in the enabled_list
      od;
   forever
```

FIGURE 7

5. CONCLUSION

We have shown how global flow analysis can be used to reduce busy waiting in implementations of conditional critical regions. Currently, we are trying to extend our flow analysis techniques to monitors. As noted in Section 1, the same type of busy waiting can occur in monitor implementations if monitor procedures are allowed to wait on arbitrary boolean expressions. In this case, the flow analysis is complicated by the transmission of parameters that is associated with a monitor call. It appears that some type of interprocedural data flow analysis may be necessary to correctly handle monitors.

We are also investigating ways of increasing the efficiency of our flow analysis algorithm. An obvious problem is determining which widening operators should be used with a particular parallel program. This problem is important since the widening operator determines how quickly the flow analysis algorithm converges. The efficiency can also be improved by developing methods to limit the number of interprocess edges that are considered during the flow analysis.

Finally, we believe that flow analysis techniques may be applicable to other important problems in parallel computation such as determining when two statements are mutually exclusive. Results in [CL77] suggest that flow analysis might 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, NJ.

[CL79] Clarke, E. M. Synthesis of Resource Invariants for Concurrent Programs. 6th POPL Conference, January 1979.

[CL79b] Clarke, E. M. and L. Liu. Approximate Algorithms for Optimization of Busy Waiting in Parallel Programs. 20th FOCS Conference, October 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 is 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. Acata Informatica 6, 227-249, 1976.