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.

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 [H073] 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,
```

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 ([BH73], [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" Rm and a "wait queue" Rw. Each process wishing to enter a critical region for R is first placed on Rm. Processes on Rm 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 Rw. Each time a process completes the execution of a critical region for R all processes waiting on Rw must be moved to Rm, 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 Rm and Rw before its entry condition finally becomes true.

* This work was partially supported by NSF Grant MCS-7908365.
Suppose that a consumer process "consumer1" is waiting on \( R_w \) and another consumer process "consumer2" is executing the body of the critical region \( CSR1 \). After the execution of \( CSR1 \) 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

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

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 always enables PR1 and that execution of CS1 always 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 conservative, 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 language 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 of two parts: an initialization part "\( R := E \)" in which values are assigned to the program variables \( R \), and a parallel execution part "resource \( R(x) \) cobegin \( P1/\ldots/Pt \) coend" which permits the simultaneous or interleaved execution of the statements in the processes \( P1, \ldots, Pt \). 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 "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 accessed within critical regions and must be listed in the prefix \( R(X) \) 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.

\[
np, nc, b := 0, 0, 0; \quad \text{resource } R(np, nc, b):\quad \text{cobegin}
\]

\[
\text{producer_1/producer_2/\ldots/producer_m/}
\]

\[
\text{consumer_1/consumer_2/\ldots/consumer_n}
\]

\[
\text{coend}
\]

Let \( P \) be a parallel program of the form described above where \( x = (x_1, \ldots, x_p) \) is the set of shared variables. A state of \( P \) is an \( (m+n) \)-tuple \( \sigma = (v_1, \ldots, v_m, p_1, \ldots, p_n) \) where \( v_j \) is the integer value of variable \( x_j \), and \( p_j \) is the program counter for process \( P_i \). Thus, \( \sigma_o = (x_1, \ldots, 1) \) denotes the state of the program after the initialization part \( R := E \) has been executed. The function \( p_\sigma(x) \) gives the value of the \( i \)th program counter in state \( \sigma \), and the function \( val(x, \sigma) \) gives the value of shared variable \( x \) in state \( \sigma \). Both functions are extended in the natural manner to apply to sets of states.

We write \( \sigma \Rightarrow \sigma' \) if critical region \( C_i \) can be executed in state \( \sigma \) to produce state \( \sigma' \). Likewise, \( \sigma \Rightarrow \sigma \) indicates that state \( \sigma \) can occur during execution of program \( P \) on initial state \( \sigma_o \). We use \( \Sigma \) 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 enable relation for \( P \) is a binary relation \( e \) on CCR with the property that \( (C_i, C_j) \in e \) whenever there is a computation of \( P \) of the form

\[
\sigma_o \Rightarrow \sigma_k \Rightarrow \sigma_{k+1} \Rightarrow \ldots \Rightarrow \sigma_{k+2} \text{ and condition } b_j \text{ of region } C_j \text{ is false in state } \sigma_k \text{. Similarly, we may define a disable relation } d \text{ for } P \text{ such that } (C_4, C_5) \in d \text{ whenever there exists a computation }
\]

\[
\sigma_o \Rightarrow \sigma_k \Rightarrow \sigma_{k+1} \Rightarrow \ldots \Rightarrow \sigma_{k+2} \text{ with } b_j \text{ true in } \sigma_k \text{ but not in } \sigma_{k+1} \text{.}
\]

For the unbounded buffer program we see by inspection that any enable relation must be a superset of \( e_{min} = \{(PR1, PR1), (PR2, CO1), (CO2, CO2)\} \). Similarly, any disable relation must be a superset of \( d_{min} = \{(PR1, PR1), (CO1, CO1)\} \). 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.

2
3. GLOBAL DATA FLOW ANALYSIS FOR PARALLEL PROGRAMS

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 \( n' \). \( \text{Succ}(n) \) will denote the set of all successors of the node \( n \). A sequence of edges \( \rho = (n_0, n_1), (n_1, n_2), \ldots, (n_{k-1}, n_k) \) 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 \( n' \).

3.1 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]).

3.2 DEFINITION: An information propagation problem is a triple \( <G, F, n_0> \) where

1. \( G = (N, E, n_0) \) is a flow graph,
2. \( F = \{f_e: \mathcal{E}(\mathcal{L}) \rightarrow \mathcal{E}(\mathcal{L}) | e \in E\} \) is a set of transition functions, and
3. \( x_0 \in \mathcal{E} \) is the initial information attached at node \( n_0 \).

A transition function \( f_e: \mathcal{E}(\mathcal{L}) \rightarrow \mathcal{E}(\mathcal{L}) \), 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 \( n \).

Let \( P \) be a parallel program with processes \( P_1, P_2, \ldots, P_t \). Assume that \( C_{i1}, C_{i2}, \ldots, C_{ij} \) 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 \( \text{ST}(i, j) = \{g | g \in \mathcal{P}, pc(g) = j\} \) be the set of program states which may precede the execution of \( C_{ij} \). In general it is impossible to calculate \( \text{ST}(i, j) \) 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 \( \text{ST}(i, j) \).

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:

\[ N = \bigcup_{i=1}^{t} N_i \cup \{n_0\} \]

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

\[ E' = \{(n_{i0}, n_{j0}) \mid i \neq j \text{ and } n_{i0}, n_{j0} \text{ represent conditional critical regions}\}. \]

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 inessential edges will not be used at all.

Transition functions \( f_e: \mathcal{E}(\mathcal{L}) \rightarrow \mathcal{E}(\mathcal{L}) \) for a flow graph \( G \) may be defined as follows: with each node \( n \) of \( G \) we associate a function \( f_n: \mathcal{E}(\mathcal{L}) \rightarrow \mathcal{E}(\mathcal{L}) \) such that \( f_n(X) = (\text{post}(n, \sigma)) | X \in \mathcal{E} \) and the instruction at \( n \) is executable under \( \sigma \). Here, \( \text{post}(n, \sigma) \) is the state which results when the instruction at node \( n \) is executed in state \( \sigma \). Thus, if \( n \) represents a conditional critical region \( C_{ij} \), \( f_n(X) = (A_{ij} | \sigma \in X \text{ and } b_{ij}(\sigma) = \text{true}) \). For an edge \( e = (n, n') \) where \( n'' = n_{i0} \) the function \( f_e: \mathcal{E}(\mathcal{L}) \rightarrow \mathcal{E}(\mathcal{L}) \) is defined by \( f_e(X) = (\sigma \in f_n(X) \mid pc_e(\sigma) = q) \).

The flow analysis algorithm that we use (see Fig. 2) is a modification of Kildall's algorithm [K73]. To ensure that our algorithm terminates we use a widening operator \( W \) [C76]. For each \( n \in N \), the successive values assigned to \( \text{COV}(n) \) at statement 9
We first consider a widening operator $W_1$ for $\mathcal{P}(Z)$, where $Z$ is the set of integers and $I = [a, b]$ is a finite interval in $Z$. For each $X \in \mathcal{P}(Z)$ we define $W_1(X) = U V_1(r)$ where $V_1(r)$ is $(-\infty, a)$ if $r < a$, $rX$ $(b, +\infty)$ if $r > b$, and $r$ otherwise.

For a general parallel program with $m$ variables and $t$ processes, we define a widening operator $W_1', \ldots, W_m$ for $\mathcal{P}(Z)$, where $I_1, \ldots, I_m$ are given finite intervals in $Z$, such that

$$W_1', \ldots, W_m(X) = \prod_{i=1}^{m} W_1\left(val_i(X) \times W_1\left(val_i(X) \times \text{pc}(X)\right)\right).$$

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 $\left[\min(a_i, a_i + d_i), \max(b_i, b_i + c_i)\right]$.

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 $W_1'$ we obtain the approximation:

$$COV(n_0) = \phi, \quad COV(n_{10}) = COV(n_{20}) = \{0\}$$

$$COV(n_{11}) = \{0\} \times [0, 1] \times \{0, \infty\} \times \{1\} \times \{1, 2\}$$

$$COV(n_{12}) = \{1\} \times [0, 1] \times \{0, \infty\} \times \{2\} \times \{1, 2\}$$

$$COV(n_{21}) = \{0, 1\} \times [0] \times \{0, \infty\} \times \{1\} \times \{1, 2\}$$

$$COV(n_{22}) = \{0, 1\} \times \{1\} \times \{0, \infty\} \times \{1, 2\} \times \{2\}$$

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 [C76]. The interval calculus is usually easy to implement: $[a, b] + [c, d] = [a + c, b + d]$, $[a, b](-) = [b, a]$, 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 approximations. Various propagating sequences [R79] 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 equivalence classes are constructed using heuristics about how the execution of one critical
The information obtained by the flow analysis can be used in a straightforward manner to approximate the six enable and disable relations. Let \( C \) 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 \( \text{COV}(n) \cup \mathcal{F}(C) \) for the solution to the information propagation problem \( \mathcal{G}_e(f, \phi^\ast) \). Let \( n \) and \( n' \) be essential nodes representing two critical regions \( C_i \) and \( C_j \) 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 \((\tilde{C}_i, \tilde{C}_j) \) in \( ea \) if there exists a state \( \sigma \in \text{COV}(n) \cap \widetilde{b}_{i} \cap \widetilde{b}_{j} \) such that \( \sigma' = f_e(\sigma) \in \widetilde{b}_{j} \) and \( p_{c_j}(\sigma') = r \).

B. Put \((\tilde{C}_i, \tilde{C}_j) \) in \( sea \) if there does not exist a state \( \sigma \in \text{COV}(n) \cap \widetilde{b}_{i} \cap \widetilde{b}_{j} \) such that \( \sigma' = f_e(\sigma) \in \widetilde{b}_{j} \) and \( p_{c_j}(\sigma') = r \).

C. Put \((\tilde{C}_i, \tilde{C}_j) \) in \( wea \) if \((\tilde{C}_i, \tilde{C}_j) \) \( \epsilon \) \( ea \) \(-\) \( sea \).

D. Put \((\tilde{C}_i, \tilde{C}_j) \) in \( da \) if there exists a state \( \sigma \in \text{COV}(n) \cap \widetilde{b}_{i} \cap \widetilde{b}_{j} \) such that \( \sigma' = f_e(\sigma) \in \widetilde{b}_{j} \) and \( p_{c_j}(\sigma') = r \).

E. Put \((\tilde{C}_i, \tilde{C}_j) \) in \( sda \) if there does not exist a state \( \sigma \in \text{COV}(n) \cap \widetilde{b}_{i} \cap \widetilde{b}_{j} \) such that \( \sigma' = f_e(\sigma) \in \widetilde{b}_{j} \) and \( p_{c_j}(\sigma') = r \).

F. Put \((\tilde{C}_i, \tilde{C}_j) \) in \( wda \) if \((\tilde{C}_i, \tilde{C}_j) \) \( \epsilon \) \( da \) \(-\) \( sda \).

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 \( Q_0 \). When a process has entered a region \( CRR_j \) and found out that the condition for \( CRR_j \) does not hold, the process is placed in \( Q_1 \). When a process has executed a region \( CRR_j \) and leaves it, the system will move a process \( P_k \) to \( Q_0 \) if there is a process \( P_k \) waiting in \( Q_1 \) such that \((CRR_k, CRR_j) \) \( \epsilon \) \( sea \). Then all other critical regions \( CRR_h \) with \((CRR_j, CRR_h) \) \( \epsilon \) \( ea \) are pushed on enabled list. At the same time all critical regions \( CRR_i \) on enabled list with \((CRR_i, CRR_j) \) \( \epsilon \) \( sda \) 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 dining philosophers' problem [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
Each state \( \sigma \) of the program is a 10-tuple of the form \( \sigma = (f_0, \ldots, f_4, p_0, \ldots, p_4) \) where \( 1 \leq p_i \leq 2 \) is the program counter for process \( D_i \). The set \( \Sigma_D \) 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:

\[
W(X) = X \prod_{j=1}^{10} \pi_j(X) \text{ where } \pi_j \text{ is the projection of } X \text{ on the } j\text{th component.}
\]

When algorithm \( F \) is applied with this widening operator we obtain the covering shown below:

\[
\text{COV}(n_0) = \emptyset
\]

\[
\text{COV}(n_1) = \{(0,0,0,0,0,1,1,1,1,1)\}
\]

\[
\text{COV}(n_{11}) = \{\sigma \in \Sigma_D | \sigma = (f_0, \ldots, f_4, p_0, \ldots, p_4) \text{ and } p_1 = 1\}
\]

\[
\text{COV}(n_{12}) = \{\sigma \in \Sigma_D | \sigma = (f_0, \ldots, f_4, p_0, \ldots, p_4), p_1 = 2, p_4 = 1, \text{ and } f_1 = f_4 = 1\}
\]

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 \( \longrightarrow \), \( \rightarrow \rightarrow \), and \( \cdots \) respectively.

Figure 4

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

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.
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


[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.


