International Journal of Parallel Programming, Vol. 19, No. 5, 1990

Transformations for Optimizing Interprocess Communication and Synchronization Mechanisms[1]

Carole M. McNamee[2] and Ronald A. Olsson[2]

Received January 1990; revised December 1990

This paper presents source-level transformations that improve the performance of programs using synchronous and asynchronous message passing primitives, including remote call to an active process (rendezvous). It also discusses the applicability of these transformations to shared memory and distributed environments. The transformations presented reduce the need for context switching, simplify the specific form of interprocess communication, and/or reduce the complexity of the given form of communication. One additional transformation actually increases the number of processes as well as the number of context switches to improve program performance. These transformations are shown to be generalizable. Results of hand-applying the transformations to SR programs indicate reductions in execution time exceeding 90% in many cases. The transformations also apply to many commonly occurring synchronization patterns and to other concurrent programming languages such as Ada and Concurrent C. The long term goal of this effort is to include such transformations as an optimization step, performed automatically by a compiler.

KEY WORDS: Parallel programming; interprocess communication; synchronization mechanism; program transformation; compiler optimization.

1. INTRODUCTION

Considerable effort has gone into identifying program transformations that improve the run-time behavior of programs written in sequential program-

[1] This work was supported by NSF under Grant Number CCR88-10617.

[2] Division of Computer Science, University of California, Davis, Davis, California 95616. mcnamee@cs.ucdavis.edu, olsson@cs.ucdavis.edu.

828/19/5-1

ming languages running on uniprocessors. The range of such transformations includes optimizations that are machine independent, are machine dependent, or result from inter-procedural analysis.

The development of high-speed parallel processors and networking systems has created a need for programming languages that efficiently utilize such hardware. In many cases, a parallel language is, both syntactically and semantically, just an existing sequential language; the implementations of these implicitly parallel languages, however, identify potentially parallel operations and generate parallel target code. In other cases, a parallel language includes language features that provide explicit control over processes and their communication and synchronization. Two fundamental models for explicit interprocess communication and synchronization have been used: those based upon shared memory and those based upon message passing.[1] The shared memory model uses shared variables with exclusive access provided by mechanisms such as semaphores, monitors, and conditional critical regions. The message passing model uses messages to provide communication between processes. It may be used in both a distributed and a multiprocessor environment, thereby making it more versatile than the shared memory model.

The optimization of implicitly parallel languages has been the subject of much research and many techniques have been developed.[2–4] However, only minimal progress has been made in identifying and implementing optimizations of parallel programming languages providing explicit control over processes and their synchronization.[5] Because the implementation of explicit interprocess communication and synchronization primitives can be expensive for both shared memory and message passing, the optimization of these primitives is important. Such optimizations are difficult due to the run-time machine environment in which a program executes, the interaction between the generated code and the run-time system supporting the language, and the semantic nuances of the interprocess communication mechanisms.

This paper presents new source-level transformations that improve the performance of programs using guarded synchronous and asynchronous message passing primitives. It also discusses the applicability of these transformations to shared memory and distributed environments. The transformations presented reduce the need for context switching, simplify the specific form of interprocess communication, and/or reduce the complexity of the given form of communication. One, perhaps counter-intuitive, transformation actually increases the number of processes as well as the number of context switches to improve overall program performance. These optimizations have been hand-simulated at the source-code level and timed on several classical client/server problems. Generally, the execution

times for the optimized programs were significantly lower than those for equivalent unoptimized programs. In many cases, the execution time was reduced by more than 90%. The long-term goal of this effort is to include such transformations as an optimization step, performed automatically by a compiler.

The work presented in this paper extends earlier efforts of Habermann and Nassi,[6] Roberts *et al.*,[7] and Schauer[8] by introducing a general technique for the transformation of guarded synchronization. The technique is based on a new application of the baton passing method, a method originally developed by Andrews[9] for deriving concurrent programs.

The SR concurrent programming language[10,11] is used as the base language for our transformations since it provides all the desired forms of interprocess communication including asynchronous message passing, and synchronous calls to active bodies of code (rendezvous) and to inactive bodies of code (remote procedure call, i.e., RPC). Ada,[12] with rendezvous only, and Concurrent C,[13] with asynchronous message passing and rendezvous, provide only subsets of the desired forms of communication. [Although no transformations for RPC are presented in this paper, they are currently under investigation.[14]] However, most of the optimizations developed for SR are applicable to Ada and/or Concurrent C.

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 presents a brief overview of SR and relates it to Ada and Concurrent C. Section 4 presents the new source-level transformations and a model for their generalization. Section 5 compares the transformations with previous ones, indicates how they apply to other languages, and discusses how they can be realized automatically as a compiler optimization. Finally, Section 6 contains some concluding remarks.

2. RELATED WORK

The work described in this paper builds on several earlier efforts. Habermann and Nassi[6] proposed transformations for some cases of Ada's rendezvous. This notion is extended by Roberts *et al.*,[7] and by Schauer[8] to provide transformations for a broader range of special cases. All of these transformations replace rendezvous by less expensive mechanisms, namely, procedure calls to inactive procedures and semaphores to preserve the original order of execution. Hikita and Ishihata[15] describe high-level program transformations removing the procedure calls altogether; however, the synchronization mechanisms in the transformed programs are not low-level enough to make implementation easy, particularly in the case of guarded synchronization.

Optimizations of both asynchronous message passing and RPC have

been proposed and implemented for SR.[11] Simple messages with no parameters are replaced by semaphores. Remote calls to certain operations are replaced, at compile time or at run-time, by conventional procedure calls.

Other approaches to optimizing interprocess communication have focussed on run-time support system enhancements.[16–18] Both Hilfinger and Vestal propose using coroutines to implement tasks that employ specific forms of rendezvous, namely monitors identified by programmer supplied compiler directives and some additional compiler analysis. The program transformations presented in this paper handle a wider range of synchronization problems because mutual exclusion is transformed explicitly making no assumptions about the need for concurrency. These high-level program transformations require no compiler directives. In addition, these transformations could be applied as an initial step, thus providing additional opportunities for the application of Hilfinger's and Vestal's optimizations. Stevenson[19] proposes another approach to run-time support system enhancement with the use of an intermediate language to facilitate the development of several different implementations of Ada rendezvous.

3. LANGUAGE BACKGROUND

This section introduces SR's interprocess communication mechanisms and describes the differences between such mechanisms in Ada, Concurrent C, and SR.

3.1. SR

To introduce SR's interprocess communication mechanisms, consider an SR solution to the familiar *readers/writers* problem.[20] This problem is chosen because it is representative of a large class of client/server synchronization problems. It is also used to illustrate the new optimizing transformations presented in this paper. Details of SR and more examples can be found in Refs. 10 and 11.

Figure 1 outlines a single resource program for readers/writers. A resource is SR's unit of encapsulation and compilation. For convenience, only single resource programs are used—thus details related to resources are not given.

One instance of the *server* process, *noreaders* instances of the *reader* process, and *nowriters* instances of the *writer* process are created when the program begins execution. The server uses an input statement (*in*) to handle requests and releases from the readers and writers. The variables *nr*

and *nw* record the number of readers and writers, respectively, currently accessing the database. The guards on the request operations contain *synchronization expressions* that restrict the order, as dictated by the problem, in which processes access the database.

```
resource rw()
    op read_request();      op write_request()
    op read_release();      op write_release()
        ...

    process server
        var nr:=0, nw:=0
        do true  ->
            in read_request() & nw=0 ->
                nr++
            []read_release() ->
                nr--
            []write_request() & (nr=0 and nw=0) ->
                nw++
            []write_release() ->
                nw--
            ni
        od
    end server

    process reader(n:=1 to noreaders)
                ...
            read_request()
                ...
            read_release()
                ...
    end reader

    process writer(n:=1 to nowriters)
                ...
            write_request()
                ...
            write_release()
                ...
    end writer
end rw
```

Fig. 1. Conventional SR solution to *readers/writers*.

The operations *read_request*, etc. are invoked synchronously by the readers and writers. Synchronous invocation is the default, although it can also be specified by the optional keyword *call*. Asynchronous invocation of the release operations is appropriate in the example and can be specified by, e.g.,

```
send read_release()
```

The operations *read_request*, etc. are serviced by an input statement. Requests for all operations serviced by the same *in* statement are serviced first-come-first-served (FCFS) except as constrained by the *in* statements guards. (See Refs. 11 and 21 for details.) Guards in input statements can also contain *scheduling expressions*, which can alter the FCFS servicing order of eligible pending invocations. Such scheduling expressions are useful for writing, e.g., disk servers that order their requests based on the requests' cylinder numbers. Although not shown in Fig. 1, synchronization and scheduling expressions may reference specific invocation parameters.

Another way to service an operation is by a *proc*. A *proc* is declared like a procedure. However, a separate process is created to execute the *proc* code for each invocation. Table I summarizes the effects of the various combinations of SR's interprocess communication primitives.

SR provides semaphores with *P* and *V* commands. However, they are simply syntactic abbreviations for parameterless operations and simple *in* and *send* statements, i.e., $P(x)$ is

```
in x() → ni
```

and $V(x)$ is

```
send x().
```

SR's implementation optimizes such semaphores by using semaphores provided in the kernel of its run-time support system when the *P*'s and *V*'s occur in the same address space, as they do for single resource programs.

Table I. Summary of SR's Interprocess Communication Primitives

Invocation	Service	Effect
call	proc	procedure call
call	in	rendezvous
send	proc	dynamic process creation
send	in	message passing

3.2. Ada

The Ada programming language provides a rendezvous mechanism similar to SR's; it does not provide asynchronous message passing or an RPC mechanism.[12] Ada's rendezvous mechanism is invoked by a blocking call to an Ada "entry" (similar to an SR "operation"). The entries are serviced by Ada *accept* statements. A *select* statement is provided to enable selection of one of a number of entries and is analogous to the SR *in* statement with multiple arms. Guards are provided in the form of a *when* clause. The Ada equivalent of the SR *in* statement in Fig. 1 is presented in Fig. 2. One significant difference between SR's and Ada's rendezvous mechanisms is the order in which pending invocations are serviced. Ada places invocations for entries on entry-specific queues and services each entry queue in FCFS order. Selection of an entry when more than one guard is true and invocations are pending is nondeterministic, not FCFS as in SR.

3.3. Concurrent C

Concurrent C provides both a rendezvous mechanism similar to both SR's and Ada's, and an asynchronous message passing mechanism.[13] As in SR and Ada, rendezvous is invoked by a call to a "transaction" (similar to SR's "operation" and Ada's "entry"). Transactions are serviced by *accept*

```
select
   when nw=0 =>
        accept read_request;
        nr++;
or
   when true =>
        accept read_release;
        nr--;
or
   when nr=0 and nw=0 =>
        accept write_request;
        nw++;
or
   when true =>
         accept write_release;
         nw--;
end select;
```

Fig. 2. Ada *select* statement for the *readers/writers* problem.

statements similar to Ada's *accept* and SR's *in* statement. A *select* statement similar to Ada's is also provided. Ada-like guards are used to control selection of specific transactions. When more than one transaction-specific guard is true and those transactions have pending invocations, the selection semantics is nondeterministic, as in Ada. Pending invocations are placed on transaction-specific queues and similar to SR and Ada, invocations for the same transaction are serviced in FCFS order. As in SR, this FCFS servicing order can be controlled through the use of synchronization and scheduling expressions. The Concurrent C equivalent of the SR *in* statement in Fig. 1 is given in Fig. 3.

Concurrent C's asynchronous message passing mechanism is similar to SR's, although more restrictive. Transactions can be declared to be *async*. Invocations of *async* transactions are implicitly nonblocking and are serviced just like synchronous transactions. In SR, operations may be invoked either synchronously or asynchronously unless specifically restricted in the operation's declaration.

3.4. Effect of Semantic Differences

The semantic differences between Ada, Concurrent C, and SR described earlier, although subtle, are significant when devising semantic-preserving transformations and when considering the applicability of the optimizations developed for SR to Ada and Concurrent C. Section 4.3.2 discusses this issue in greater detail. In addition, the implementations of

```
select
   (nw=0):
      accept read_request()
         nr++;
or
      accept read_release()
         nr--;
or
    (nr=0 and nw=0):
       accept write_request()
          nw++;
or
       accept write_release()
          nw--;
```

Fig. 3. Concurrent C *select* statement for the *readers/writers* problem.

these different selection strategies also have an effect on the performance of specific transformations. The implementation of SR's FCFS servicing policy places requests for service for operations serviced by the same *in* statement on the same queue.[11] Ada and Concurrent C implementations typically use separate queues for each operation.[13,22] Ensuring the FCFS behavior of SR's multi-operation queues in our transformations results in a performance penalty; however, the run-time reduction is still significant (Section 4.3.2).

A factor affecting the development of specific transformations and the measurement of their performance is the availability of efficiently implemented semaphore operations. As described earlier, SR provides such a facility. Ada also provides efficiently implemented semaphores as a special package. Concurrent C can simulate semaphore operations.[13] However, each semaphore requires a manager process to execute *P* and *V* transactions. The cost of invoking the *P* and *V* transactions plus the incumbent context switching at least partially negates the effect of using the semaphores.

4. GLOBAL PROGRAM TRANSFORMATIONS

Several approaches to program transformation have produced significant improvement in the run-time performance of programs with a substantial need for interprocess communication. These include transformations that reduce the need for context switching, simplify the specific form of communication, and/or reduce the complexity of the given form of communication. Languages with high-level constructs for handling the selection of messages and their resultant effects (e.g., Ada *select* statement, SR *in* statement, Concurrent C *select* statement) provide expressiveness at the cost of run-time efficiency. These constructs are often difficult to implement and costly to run. Many concurrent programs using these high-level constructs can be transformed into programs using lower-level forms of communication (e.g., semaphores) that run more efficiently.

The remainder of this section first discusses tools used to measure the performance of the transformations. It then describes optimizing transformations presented by Roberts *et al.*[7] based on the work of Habermann and Nassi[6] and presents some new transformations that significantly extend the work of Habermann and Nassi, and Roberts *et al.*

4.1. Measurement Tools

Because SR provides an efficiently implemented semaphore operation, it is possible to effect at the source level many of the transformations of

SR's high-level interprocess communication mechanisms. The optimizing transformations were applied by hand to SR programs with the resultant programs using the efficiently implemented semaphores. The execution times for the translated programs were compared to the execution times for the unoptimized programs. We were interested in the execution time associated with the implementation of high-level language features versus simple low-level features independent of the underlying costs of sending messages between processors. Therefore, the optimizations were applied to single resource programs only (although they can be applied appropriately to multiple resource programs as well).

The terms "run-time performance" and "cost" are used throughout this paper to mean the user execution time on a lightly loaded system. "Reduction in execution time" and "run-time reduction" both refer to the difference between the user execution times for the transformed and original programs as a percentage of the original program's user execution time. All user execution times have been adjusted to eliminate constant costs not associated with the cost of interprocess communication and synchronization (see Section 4.4).

The UNIX *time* command[3] was used to determine the user times for both optimized and unoptimized versions of several classical problems including *readers/writers* and *bounded-buffer*. Timing was done on a Sun 3/160, a Sun SPARCstation, an Encore Multimax, a Sequent Symmetry, and several VAXes including a 750 and an 8600. Programs run on the Encore Multimax and the Sequent Symmetry used only one processor. All systems were lightly-loaded when timing tests were run.

The run-time performance statistics presented in this section reflect run-time reductions for the readers/writers problem with from zero to one hundred readers, zero to one hundred writers, and 10,000 requests and releases per reader/writer when run on a Sun 3/160. Tests were run multiple times and with varying amounts of computation and explicit induced context switching between reader/writer requests. The numbers provided are representative of all of the results. They are also representative of the results obtained when the transformations were applied to other problems and run on the other systems. Section 4.4 contains more complete results and additional interpretation.

4.2. Roberts' Transformations

A series of program transformations is presented in Ref. 7 based upon Haberman and Nassi's optimizations of Ada rendezvous.[6] All of the trans-

[3] Results using the *time* command and results using *getrusage*, a UNIX system call, were compared and found to be nearly identical.

formations in Ref. 7 are applied to client/server models with centralized servers, i.e., one server task accepts entries from any number of client tasks. The transformations replace server *select/accept* statements and their bodies by passive procedures containing the bodies. Client tasks remain unchanged; however, their calls to the Ada entries reduce to simple procedure calls when clients and server procedures share the same address space. Low-level semaphore operations are used to control execution of the procedures and ensure the semantics of Ada's *select* and *accept* statements. An SR version of this simple client/server transformation appears in Fig. 4.

The new transformations presented in this paper focus on a client/server model presented in Example 4 in Ref. 7. This model provides mutually exclusive servicing of client requests for one of a number of operations. An optimizing transformation for this model is presented in Example 4 in Ref. 7 where a task contains a guarded select statement. Roberts *et al.* present a sequence of transformations that ultimately results in a server task controlling semaphores used to control entry into the procedures used to implement the *accept* statements. Because the server procedures must ensure that their guards are satisfied as well as provide mutually exclusive access to their code, the procedures must wait on two semaphores. In addition, the value of other operations' guards may be altered by the servicing of an operation necessitating the use of a joint semaphore operation for the two semaphores. The joint semaphore operation, JOINTP(x, y), simply waits until a P operation can be performed on each of its semaphore arguments, x and y, before allowing the server procedure to continue processing. An SR version of this client/server model and

```
process server
   do true ->
        in oper1() -> <body1> ni
        in oper2() -> <body2> ni
        ...
        in opern() -> <bodyn> ni
    od
end server

process client
        ...
        call operi()
        ...
end client
```

(a): Simple client/server model

```
proc oper1()
        P(sem1)
          <body1>
        V(sem2)
end oper1

...

proc opern()
        P(semn)
          <bodyn>
        V(sem1)
end opern

process client
        ...
        call operi()
        ...
end client
```

(b): Transformed server

Fig. 4. SR version of simple client/server model and its transformation.

```
process server
   do true ->
      select
         when cond1 ->
            in oper1(...)  ->
                    <body1>
            ni
         [] cond2 ->
            in oper2(...)  ->
                    <body2>
            ni
         ...
         [] condn ->
            in opern(...)  ->
                    <bodyn>
            ni
      end select
   od
end server
```

(a): Server in Robert's Example 4

```
process server
    do true ->
        P(statement_sem)
        c1 := cond1
        c2 := cond2
        ...
        cn := condn
        if (~c1 and ~c2 and ... ~cn) ->
              raise error
        fi
        if c1 ->   V(op_sem1)    fi
        if c2 ->   V(op_sem2)    fi
        ...
        if cn ->   V(op_semn)    fi
    od
end server

proc oper1(...)
    JOINTP(op_sem1, mutex)
      <gobble other released op_sems>
      <body1>
    V(statement_sem)
    V(mutex)
end oper1

...

proc opern(...)
    JOINTP(op_semn, mutex)
      <gobble other released op_sems>
      <bodyn>
    V(statement_sem)
    V(mutex)
end opern
```

(b): Optimized server

Fig. 5. Pseudo-SR version of client/server model in Roberts' Example 4 and its transformation.

the transformed server appears in Fig. 5. An Ada-like *select* statement with a *when* clause as well as a JOINTP have been introduced to preserve the original semantics of Roberts *et al.*'s transformation. It is significant that this transformation does not completely eliminate the server task as the transformation in the simple client/server model does. Ada semantics specifies that all of the *select* statement guards be evaluated before any attempt at rendezvous is made and that a *PROGRAM_ERROR* must be raised if no guard is true.[7,12] As a result, Habermann and Nassi's and Roberts' transformations are not semantic preserving; specifically, there is no task to raise the *PROGRAM_ERROR*.[4]

[4] It can be argued that, at some level of system specification, the presence of the server task is not required. Both the original version and the transformed version that eliminates the task will satisfy the specification even though the two programs may not be semantically equivalent.

This subtle semantic difference between Ada, and SR and Concurrent C, allows Roberts' transformations to be applied automatically only to programs written in SR or Concurrent C.

4.3. New Transformations

The following transformations significantly extend those described previously and are demonstrated using the familiar *readers/writers* problem, an instance of the client/server model presented in Fig. 5. A conventional SR solution to this problem was presented in Fig. 1.

4.3.1. Transformation Tools—Passing the Baton

Several of the optimizing transformations of programs using complex forms of interprocess communication, including guarded or selective synchronization, involve the application of a *baton passing* technique, a technique originally developed by Andrews[9] as part of a method for deriving correct concurrent programs. Figure 6 illustrates a semaphore solution to the *readers/writers* problem obtained using this technique.[9] The *mutex* semaphore is used to provide mutually exclusive access to shared variables, and the *read_signal* and *write_signal* semaphores are used to awaken delayed readers and writers, respectively. The name baton passing is due to the fact that when a delayed reader or writer is signaled the entry semaphore is not released; instead mutually exclusive access (i.e., the "baton") transfers directly to the signaled reader or writer. It then becomes the responsibility of the signaled process to either signal a still delayed process, if possible, or release exclusion.

In our transformations, the baton passing technique is used for an entirely different purpose than program derivation. Specifically, it is used to ensure the integrity of guarded synchronization when optimizing transformations are applied.

4.3.2. Rendezvous Transformations

As described in Section 3, SR and Concurrent C provide more powerful rendezvous selection mechanisms than Ada. Synchronization expressions in SR and Concurrent C may contain references to actual operation parameters and must be evaluated for each message that the receiving statement attempts to service. The evaluation of the synchronization expressions is a function of the message selection process. In contrast, Ada select guards are evaluated before any messages are considered for servicing. As a result, they may not contain references to operation parameters in specific messages. Ada guards are not considered part of the message selection process but part of the code for the task itself making it

```
initially:      mutex:=1;
                nr:=0; nw:=0;
                dr:=0; dw:=0;
                read_signal:=0;
                write_signal:=0;

Reader process: P(mutex)                    Writer process: P(mutex)
                if nw = 0 ->                                if nr = 0 and nw = 0 ->
                   skip                                        skip
                [] nw > 0 ->                                [] nr > 0 or nw > 0 ->
                   dr++                                        dw++
                   V(mutex)                                    V(mutex)
                   P(read_signal)                              P(write_signal)
                fi                                          fi
                nr++                                        nw++
                SIGNAL                                      SIGNAL
                  read  whatever                              write whatever
                P(mutex)                                    P(mutex)
                nr--                                        nw--
                SIGNAL                                      SIGNAL

where SIGNAL is

            if nw = 0 and dr > 0 ->
               dr--
               V(read_signal)
            [] nr=0 and nw=0 and dw>0  ->
               dw--
               V(write_signal)
            [] else ->
               V(mutex)
            fi
```

Fig. 6. Semaphore solution to *readers/writers* using the baton passing technique.

difficult to eliminate the server task.[7] The synchronization expressions in SR and Concurrent C are a component of the communication mechanism itself. Therefore, their evaluation may be moved to separate procedures and processes can be eliminated from programs, making this more powerful mechanism unexpectedly easier to optimize.

- Process Elimination/Simplification of Form of Communication
  Extending the technique of replacing rendezvous by a call to an inactive procedure, a loop containing an SR *in* statement with synchronization expressions can be replaced by passive procedures using several variants of the baton passing mechanism described in Section 4.3.1. These transformations significantly increase the number of communications and the complexity of the code but the communications are very low-level (semaphores) and thus much less costly. Figure 7 presents an SR solution to the *readers/writers* problem using the baton passing technique. Because the server process is eliminated, the joint semaphore operation present in Roberts' transformation is also eliminated. A dramatic reduction in execution time, 91–99%, is achieved.

Unfortunately, the signaling mechanism at the end of the procedures introduces a degree of nondeterminism (due to the nondeterministic semantics of SR's if statement). The nondeterministic servicing of delayed processes violates SR's FCFS servicing of invocations to operations serviced by the same *in* statement but is perfectly acceptable in Ada and Concurrent C where the semantics of the select statement specifies nondeterminism. The transformed procedures guarantee FCFS servicing upon entry assuming that the entry semaphore has a FCFS servicing policy. However, processes that enter a procedure and find that they are blocked waiting for a signal are not guaranteed to be awakened in the necessary FCFS order due to the nondeterminism inherent in the *if* statement. To solve this problem, delayed readers and writers must be signaled in the order in which they are delayed (to the extent possible given the synchronization constraints of the *readers/writers* problem). To guarantee the FCFS order specified by SR semantics, a list of blocked processes is required so that delayed processes may be awakened and allowed to continue when conditions permit. Mutually exclusive access to the list is required along with routines to insert into the list and delete from the list in FCFS order as constrained by the guards in the signaling mechanism. Since the SR *in* and *send* statements already perform these operations, it is expedient to use them to maintain such a list.[23] This appears to be a step backward and in fact, it is, but the *in* statement used in the signaling mechanism with only two operations is simpler than the original *in* statement in Fig. 1, which services requests for four different operations. In addition, the *in* statement services not all requests, but only delayed requests. For the *readers/writers* problem the difference between the total number of requests and the number of delayed requests is especially significant when the number of readers is much greater than the number of writers. Because the *in* statement is a higher-level communication mechanism than the semaphores used in the transformation in Fig. 7, there is a performance penalty but the run-time reduction is still significant, ranging from 51–97%. This solution appears in Fig. 8.

- Simplification of Communication Form
  Using a somewhat counter-intuitive approach—i.e., increasing the number of processes but simplifying the communication mechanism—also produces a significant improvement in performance. Replacing each arm of an *in* statement with an active

```
resource rw()
        ...
    proc read_request()
        P(mutex)
        if nw > 0 ->
            dr++
            V(mutex)
            P(read_signal)
        fi
        nr++
        SIGNAL
    end read_request
        ...
    proc read_release()
        P(mutex)
        nr--
        SIGNAL
    end read_release
        ...
    process reader(n:=1 to noreaders)
                ...
            read_request()
                ...
            read_release()
                ...
    end reader
                ...
end rw

where SIGNAL is:
        if nw=0 & dr>0 ->
            dr--
            V(read_signal)
        [] nr=0 & nw=0 & dw>0 ->
            dw--
            V(write_signal)
        [] else ->
            V(mutex)
        fi
```

Fig. 7. *Readers/writers* with rendezvous replaced by calls to inactive procedures. Procedures *write_request* and *write_release* are similar to those for the read operations.

```
resource rw()
            ...
    op delay_read_signal()
    op delay_write_signal()
            ...
    proc read_request()
        P(mutex)
        if nw > 0 ->
            dr++
            V(mutex)
            send delay_read_signal()
            P(read_signal)
        fi
        nr++
        SIGNAL
    end read_request

    proc read_release()
        P(mutex)
        nr--
        SIGNAL
    end read_release
        ...
    process reader(i:= 1 to noreaders)
                ...
            read_request()
                ...
            read_release()
                ...
    end reader
                ...
end rw

where SIGNAL is:
        in delay_read_signal()  & nw=0  ->
            dr--
            V(read_signal)
        [] delay_write_signal()  & nr=0 & nw=0  ->
            dw--
            V(write_signal)
        [] else ->
            V(mutex)
        ni
```

Fig. 8. Transformation that preserves the FCFS semantics of SR. Procedures *write_request* and *write_release* are similar to those for the read operations.

process and using semaphores to ensure mutual exclusion gives a run-time reduction of 72–94% if nondeterminism is acceptable, and 64–87% if FCFS semantics is required. Figure 9 presents this transformation using a nondeterministic signaling mechanism.

```
resource rw()
                ...
    process read_begin
        do true ->
            P(read_request)
            P(mutex)
            if nw > 0 ->
                dr++
                V(mutex)
                P(read_signal)
            fi
            nr++
            V(oktoread)
            SIGNAL
        od
    end read_begin

    process read_end
        do true ->
            P(read_release)
            P(mutex)
            nr--
            V(read_release_ok)
            SIGNAL
        od
    end read_end
        ...
    process reader(i:=1 to noreaders)
                ...
            V(read_request);   P(oktoread)
                ...
            V(read_release);   P(read_release_ok)
                ...
    end reader
                ...
end rw
```

Fig. 9. Transformation of operations into active processes with semaphores to control progress of processes. SIGNAL is as it is in Figs. 7 and 8.

Calls to operations are transformed into *V/P* pairs. Because the calling processes access the semaphores directly, all tasks must share the same memory. If the operations are guarded, then a baton passing mechanism or another synchronization technique is required to ensure fairness. Interestingly, this increase in the number of processes results in only a relatively small increase in the total number of *forced* context switches. A forced context switch is a context switch necessary to complete a communication or synchronization operation. For rendezvous, the number of forced context switches is either two or three depending upon whether the invoker requests service before the server attempts to service or the server attempts to service before the invoker requests service.[6] Although the number of processes increases in this transformation, the number of context switches needed to perform each interprocess synchronization does not increase. In the original SR code, readers and writers were forced to synchronize with the same server process. In the transformed code, readers and writers synchronize with different processes, but do so the same number of times. The small increase in the total number of forced context switches that was observed is due to that fact that *read_requests* and *read_releases* are serviced by the same process in the original program and by separate processes in the transformed program, necessitating the additional context switching. Obviously, the performance of this transformation is dependent upon the relative cost of rendezvous versus the cost of context switching. Context switching has been shown to be considerably less expensive than rendezvous in both SR and Ada.[24,25] As in the previous case, the signaling mechanism of Fig. 8 may be used to preserve FCFS semantics.

- Simplification of Communication Complexity
  Yet another approach to optimizing interprocess communication mechanisms involves simple reductions in the complexity of a mechanism. For example, eliminating synchronization expressions and/or reducing the number of arms in an *in* statement can provide some improvement in performance. It is possible to do so using problem-specific information not apparent from the syntactic form of the code. Figure 10 gives a transformation of the readers/writers problem that replaces the *in* statements of Fig. 1 by several less complex *in* statements. The *in* statements have fewer arms and all operation requests for servicing are not placed on the same wait queues; thus they run more efficiently. Run-time reductions of

0–82% were obtained on the Sun 3/160; slight performance losses, however, were observed on some machines for the case with many readers and no writers. When the number of readers greatly exceeds the number of writers the number of queued requests for service is small, as is the number of blocked processes. This results in efficient processing of the *in* statement despite the number of arms.

This transformation is independent of environment. It can be applied to programs running in either distributed or multiprocessor environments making automation of the transformation feasible. A variety of other transformations similar to the transformation in Fig. 10 is currently under investigation. For example, it is possible to nest the *in* statements thereby giving enough context to eliminate the synchronization expressions. However, most of the transformations of this type suffer from the fact that they require problem-specific semantic information to apply the transformation, making automatic transformation unreasonable.

4.3.3. Asynchronous Message Passing Transformations

- Simplification of Communication Form
  A process containing an *in* statement with multiple arms that service asynchronous sends can be replaced by multiple processes, one for each arm, with semaphores controlling entry and mutual exclusion to improve performance. This transformation is similar to the multiple process transformation of synchronous communication primitives. To illustrate this approach, consider again the original readers/writers problem from Fig. 1, but with synchronous calls to *read_release* and *write_release* replaced by asynchronous sends. The readers/writers solution in Fig. 11 illustrates the transformation of asynchronous primitives in conjunction with the transformation of synchronous calls to inactive procedures. Asynchronous sends to *read_release* and *write_release* are transformed to semaphore communications with active processes, providing a 91–97% reduction in execution time. Again, true FCFS semantics can be reproduced but with some performance degradation. A 44–86% reduction in execution time is obtained.

4.4. Results and Interpretation

The performance of the transformations presented in Sections 4.2 and 4.3 is summarized below. The transformations were applied to the

bounded-buffer problem with results similar to those obtained for the readers/writers problem.

Recall from Section 4.1, the readers/writers problem and its transformed variants were run on six different systems varying the number of

```
resource rw()
        ...
    op read_request();          op write_request()
    op read_release();          op write_release()
        ...
    process server
        do nr = 0 & nw = 0 ->
            in read_request() -> nr++
            [] write_request() -> nw++
            ni
        [] nr>0 ->
            in read_request() -> nr++
            [] read_release() -> nr--
            ni
        [] nw > 0 ->
            in write_release() -> ni
            nw--
        od
    end server

    process reader(n:=1 to noreaders)
              ...
            read_request()
              ...
            read_release()
              ...
    end reader

    process writer(n:=1 to nowriters)
              ...
            write_request()
              ...
            write_release()
              ...
    end writer
end rw
```

Fig. 10. Transformation into multiple *in* statements eliminating the need for synchronization expressions.

readers, the number of writers, and the amount of work and context switching done by each reader/writer process between requests to read and release. The specific work done between reader/writer requests for service varied from nothing, to a random number of explicit induced context switches, to a random number of numerical computations along with a random number of explicit induced context switches. Base cases reflecting constant costs unrelated to the costs of the interprocess communication and synchronization of the reader/writer processes were timed and their results were subtracted from the times for the conventional and trans-

```
resource rw()
        ...
    proc read_request()
        P(mutex)
        if nw > 0   ->
            dr++
            V(mutex)
            P(read_signal)
        fi
        nr++
        SIGNAL
    end read_request
        ...
    process read_end
        do true ->
            P(read_release)
            P(mutex)
            nr--
            SIGNAL
        od
    end read_end
        ...
    process reader(i:=1 to noreaders)
                ...
            read_request()
                ...
            V(read_release)
                ...
    end reader
        ...
end rw
```

Fig. 11. Transformations of synchronous *read_request* and asynchronous *read_release* communication primitives.

formed programs before computing run-time reduction percentages. The range of run-time reduction percentages is shown in Table II.

The slightly smaller percentages observed on the SPARC are due to the use of Sun's lightweight process library package in that particular implementation of SR. Being a general package, the handling of processes is less efficient than that provided by the specialized routines in SR's run-time support on the other systems.

The run-time reductions vary considerably depending upon the number of readers versus writers. The effect of varying the number of readers and writers can be seen in Fig. 12 where the total number of reader/writer processes was held constant at one hundred but the number of readers and writers each varied from zero to one hundred. The results shown were obtained on a Sun 3/160 but are again representative of results obtained on the other systems. The differences are largely due to the increased costs associated with high-level synchronization mechanisms. Specifically, when the number of writers is much greater than the number of readers and many requests for service are queued and their processes blocked, the costs associated with servicing requests from these queues is significant.

Another factor affecting the results is the somewhat naive approach to determining the base case where the number of unblocked processes, i.e., those that are ready to run, is always equal to the number of readers plus the number of writers. With the presence of interprocess synchronization in the actual reader/writer solution, the number of unblocked processes varies depending upon the state of other reader/writer processes. Surprisingly, run-time reductions exceeding 100 percent were observed for some of our

Table II. Run-time Reductions for Optimizing Transformations Applied to the *Readers/Writers* Problem

	Percent Reduction in User Execution Time					
Transformation	Vax 750 (vax)	Vax 8600 (vax)	Sun 3/160 (mc68020)	Encore[a] Multimax (ns32332)	SPARC-station (sparc)	Sequent[a] Symmetry (i80386)
Baton Passing	89–99	93–105	91–99	78–102	78–102	91–100
Baton Passing (FCFS)	45–98	43–98	51–97	47–98	50–99	48–98
Multiple process	66–95	69–103	72–94	65–96	45–76	67–97
Multiple process (FCFS)	59–89	59–95	64–87	47–91	38–71	56–93
Asynch release	83–98	90–106	91–97	73–100	74–96	88–100
Asynch release (FCFS)	40–76	43–91	44–86	44–78	37–89	43–78
No synch expression	–3–85	1–92	0–82	–20–88	–4–69	0–89

[a] Program execution used one processor.

optimizations when they were timed with large numbers of writer processes. This anomalous behavior was observed only on the VAXes, the Encore Mutimax, and the Sequent Symmetry. It can be explained as follows. When write permission is granted to a writer process, in some cases, all other processes are blocked and our timing tests perform a significant number of context switches from a writer process to itself. Theoretically, a context switch should take constant time. However, after much gnashing of teeth, additional testing, and consultation with architecture experts, it was determined that the effect of data caching present in these machines speeds up the context switch from a process to itself, thus explaining the seemingly impossible results. In general, the presence of instruction and data caches as well as instruction pipelining in modern architectures make concise timing of programs impossible. However, the

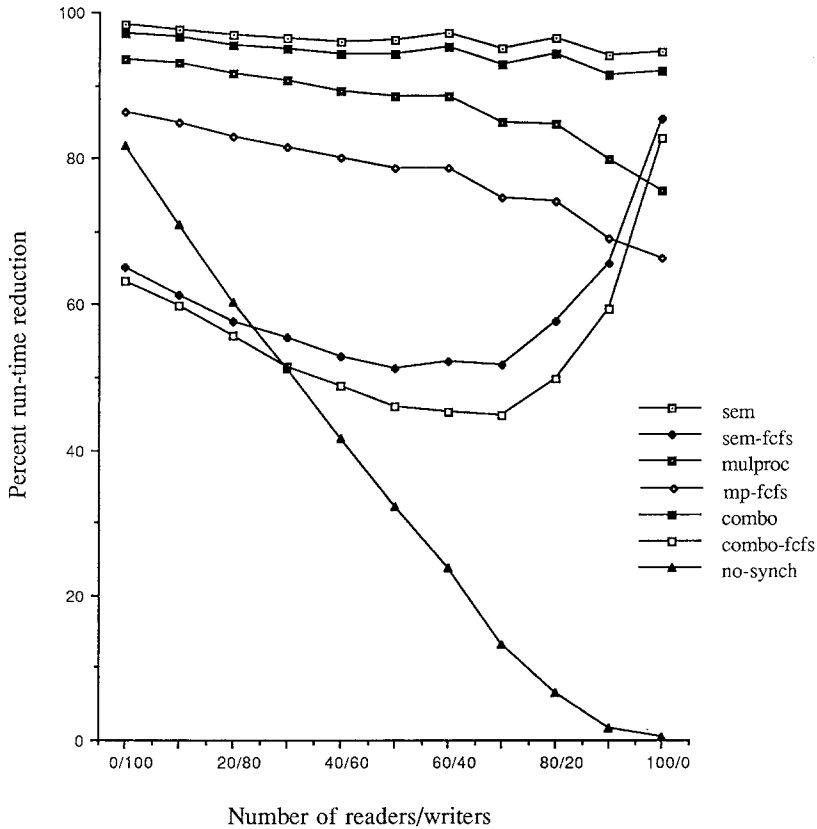

Fig. 12. Effect of varying the number of *readers/writers* on the performance of the optimizing transformations.

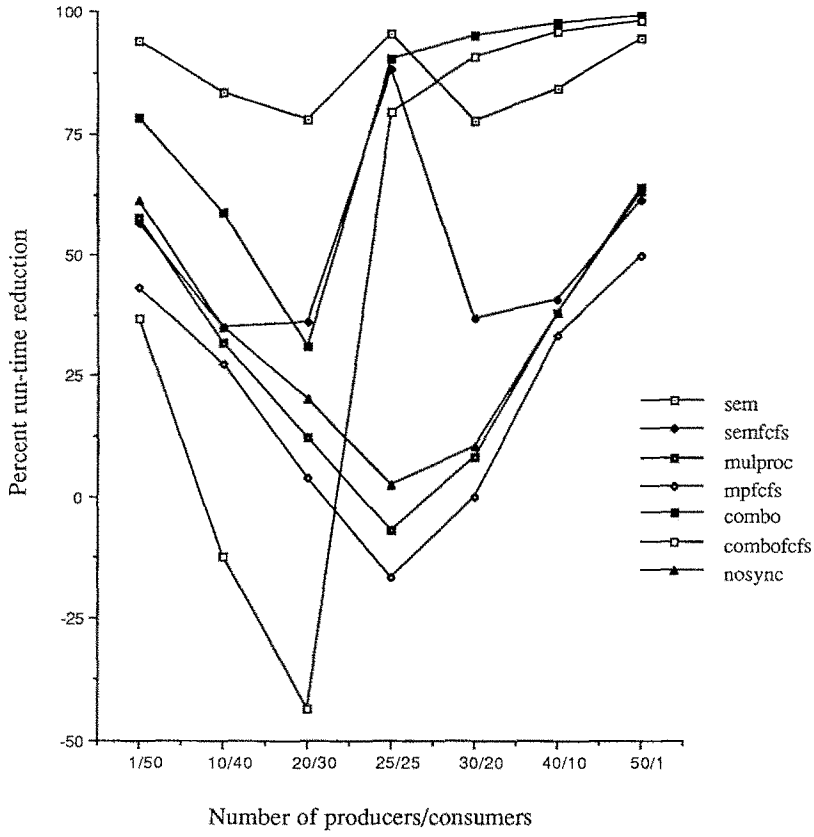

Fig. 13. Effect of varying the number of producers/consumers on the performance of the optimizing transformations.

magnitude of our run-time reductions on a variety of architectures is much greater than any possible variation due to a specific architecture.

The performance of our optimizing transformations when applied to the *bounded-buffer* problem is slightly less dramatic than for the *readers/writers* problem. Figure 13 illustrates the performance of our transformations when the number of producers and consumers varies from one to fifty and the total number of producers and consumers is either fifty or fifty-one. The buffer size is held constant at one hundred[5] and the number of items passed into and out of the buffer is held constant at 30,000.

The *bounded-buffer* problem requires more than the simple synchronization present in the *readers/writers* problem: the operations use

[5] Run-times for programs with varying buffer sizes significantly less than the total number of items passed into and out of the buffer are similar.

parameters to deposit data into and fetch data out of the buffer. While performance of the transformation with inactive procedures and baton passing should still be effective, the presence of the operation parameters makes the transformation to active processes less desireable than it is for the *readers/writers* problem. Examination of Fig. 13 confirms this. The transformation to inactive procedures for synchronous communication (sem in Fig. 13) gives good performance. The peak performance observed when the number of producers and consumers is both twenty-five is due to the fact that with nearly equivalent supply and demand for the buffer, processes are delayed less often and the transformed programs execute more quickly. This behavior is also observed in the transformation to both inactive procedures, used for a synchronous fetch, and active processes, used for an asynchronous deposit (combo in Fig. 13). The performance of the transformations using active processes only (mulproc and mulprocfcfs) is, as expected, not as good as it is when applied to programs doing synchronization only. However, its performance in a true multiprocessor environment may yield better results. The lack of symmetry in the performance of the transformations using both inactive procedures for synchronous communication and active processes for asynchronous communication (combo and combofcfs) is a result of a performance degradation in the original program caused by unconstrained asynchronous requests made by the producers. Unconstrained asynchronous requests can result in unreasonable memory requirements and long pending invocation queues, i.e., a flow-control problem.[13] The performance of the transformation that eliminates synchronization expressions is similar to that observed for the *readers/writers* problem.

5. DISCUSSION

The source-level transformations presented in Section 4 have been applied to SR programs. Although the examples presented use only a single resource, they can applied to multiple resource programs as well. In the case of the client/server model, clients may reside in different resources, and assuming that the multiple resources reside in the same shared address space the performance of the transformations would be the same as for those applied to single resource programs. Additionally, although these transformations are intended for use in a shared memory environment, they are also semantically valid in a distributed environment when all of the inactive procedures or active processes used to implement the server are located (as is reasonable) on one processor. The calls to the procedures from the client tasks simply become remote procedure calls. Whether or not these transformations to inactive procedures provides any reduction in

execution time in a distributed environment depends upon the relative cost of rendezvous and remote procedure call. Certainly no significant reduction in overall execution time will be achieved as the dominant cost is associated with the communication between machines rather than with the calling and servicing mechanisms at either end of the communication.

As stated earlier, these transformations are applicable to SR, to Concurrent C, and in some cases, to Ada. Ada's semantics for guarded selection prohibits elimination of the server process, making these transformations inappropriate as an automatic compiler optimization. However, they are still appropriate, in some cases, as high-level source-to-source transformations for Ada. Concurrent C, being similar to SR, is also a candidate for automatic optimization. Ada and Concurrent C both have a nondeterministic selection policy making their transformations more efficient than those implementing SR's FCFS selection policy.

```
process server
    do true ->
        in oper1(...) & cond1 ->
               <body1>
        [] oper2(...) & cond2 ->
               <body2>
        ...
        [] opern(...) & condn ->
               <bodyn>
        ni
    od
end server


process clienti
    ...
    operj(...)
    ...
end clienti
```

(a): Generalized server and client

```
# initially
    mutex:=1
    signal1:=0
      ...
    signaln:=0
    d1:=0
      ...
    dn:=0


proc oper1(...)
    P(mutex)
    # delay awaiting cond1
    if not(cond1) ->
        d1++
        V(mutex)
        P(signal1)
    fi
    <body1>
    # signal delayed processes
    if cond1 & d1>0 ->
        d1--
        V(signal1)
    [] cond2 & d2>0 ->
        d2--
        V(signal2)
    []  ...

    [] condn & dn>0 ->
        dn--
        V(signaln)
    [] else ->
        V(mutex)
    fi
end oper1

    ...
```

(b): Optimized server

Fig. 14. Generalized transformation using the baton passing technique.

Guarded operations with parameters may be transformed as long as their synchronization expressions are independent of operation parameters. This restriction is relevant for SR and Concurrent C, but not Ada where guards may not reference operation parameters.

Automation of these transformations is possible for the "baton passing" transformations and the general form is given in Fig. 14. An *n*-operation *in* statement embedded within an infinite loop is transformed into *n* procedures, each requesting the semaphore *mutex* and each checking the conditions associated with their operation in the original server. The delaying mechanism when the operation's condition cannot be met is straightforward as is the signalling mechanism present at the end of each procedure. Whether the transformations are applicable and how they are applied can be determined solely by syntactic analysis of the program being optimized.

Unfortunately, the optimizing transformations valid in both distributed and shared memory environments (i.e., similar to the one shown in Fig. 10) are not easily automated as they require problem-specific semantic information.

Given that the transformations can be generalized, it becomes the task of the compiler to determine when the transformation can be applied. When the clients and servers are not declared in the same compilation unit (i.e., a *resource* in SR), some form of "inter-resource" analysis is necessary to determine whether or not the clients and servers reside in the same shared memory environment. The transformations may also be useful in situations where some of the clients share memory with the server and some do not. The inter-resource analysis is complicated by the fact that SR allows pointers to operations, making it difficult to identify the locations of all potential invocations of an operation. This problem is shared by Concurrent C, which also allows pointers to transactions.

Inter-resource analysis is also necessary to extend SR's existing RPC and semaphore optimizations beyond *resource* boundaries. Current versions of SR perform a minimal amount of inter-procedural analysis to identify semaphore optimizations within a resource. For the compiler to identify all of the semaphore optimizations in an entire program, some form of inter-resource analysis is required. The optimization of SR's RPC implemented by a proc operation is currently shared by the compiler and the run-time support system. The use of inter-resource analysis would enable the compiler to perform more such optimizations.

One drawback of employing source-to-source transformations, as advocated in this paper, is that they can make debugging more difficult. In particular, the program as written by the programmer is not the same as the one being executed and debugged. The problem of debugging optimized

programs has been investigated for sequential programs.[26] Debugging concurrent programs, however, is more difficult,[27] and the transformations described in this paper can exacerbate the difficulties. Specifically, the transformations might have: added or eliminated processes; added, eliminated, or modified operations and their invocations; or altered the order in which processes execute. One solution is to debug the original untransformed program. However, some concurrent programming errors arise only when processes execute in a particular order. If the transformations affect that order, then debugging the transformed program is not the same as debugging the original program. This problem is worthy of further investigation.

6. CONCLUSIONS

This paper has presented transformations that can be applied to parallel programs to improve the performance of their interprocess communication, often dramatically.

- Our transformations significantly extend previous transformations as they apply to both guarded synchronous and asynchronous message passing including rendezvous.
- The new transformations reduce the need for context switching, simplify the specific form of interprocess communication, and/or reduce the complexity of the given form of communication.
- An additional transformation increases the number of processes as well as the number of context switches to improve program performance.
- A new application of the passing-the-baton derivation technique, to program transformation and optimization, is presented.
- The performance of some transformations applied to programs using synchronization only is better than the performance of these transformations applied to programs whose synchronization includes the passing of data.

The transformations were demonstrated on the readers/writers problem coded in SR. Performance of the transformations was measured under varying conditions including the number of processes and the number of requests for service per process. The transformations were shown to reduce the cost of interprocess communication and synchronization by more than 90% in many cases; in some cases, however, a performance loss is observed. The performance of these transformations giving mixed results,

particularly those using additional active processes, may improve significantly when run in a true multiprocessor environment.

The transformations were also shown to generalize to other languages, such as Ada and Concurrent C, and to other synchronization problems. A general transformation technique was also demonstrated. Significantly, that technique requires no problem-specific semantic information. Thus, it should be realizable as an optimization step, performed automatically by a compiler.

Future work will identify further transformations and incorporate them into a compilation system.

ACKNOWLEDGMENTS

The authors thank Gene Fisher, Mudita Jain, and the anonymous referees for helpful comments on earlier drafts of this paper and for insightful comments on several of the transformations.

REFERENCES

1. G. R. Andrews and F. B. Schneider, Concepts and Notation for Concurrent Programming, *ACM Computing Surveys*, **15**(1):3–43 (March 1983).
2. D. Padua, D. Kuck, and D. Lawrie, High-speed Multiprocessors and Compilation Techniques, *IEEE Transactions on Computers*, **C-29**(9):763–776 (September 1980).
3. M. J. Wolfe, Advanced Loop Interchange, *Proc. of Int'l. Conf. on Parallel Processing*, pp. 536–543 (1986).
4. M. J. Wolfe, Optimizing Supercompilers for Supercomputers, Technical Report DCS Report No. UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign, Ph.D. Dissertation (October 1982).
5. A. Burns, A. M. Lister, and A. J. Wellings, *A Review of Ada Tasking*, *Lecture Notes in Computer Science*, Volume 262, Springer-Verlag (1987).
6. A. N. Habermann and I. R. Nassi, Efficient Implementation of Ada Tasks, Technical Report CMU-CS-80-103, Carnegie-Mellon University, Department of Computer Science (January 1980).
7. E. S. Roberts, A. Evans, C. R. Morgan, and E. M. Clarke, Task Management in Ada—A Critical Evaluation for Real-time Multiprocessors, *SOFTWARE—Practice and Experience*, **11**:1019–1051 (1981).
8. J. Schauer, Vereinfachung von Prozess-systemen Durch Sequentialisierung, Technical Report Bericht 30/82, Universität Karlsruhe, Institut für Informatik II, Dissertation (1982).
9. G. R. Andrews, A Method for Solving Synchronization Problems, *Science of Computer Programming*, **13**(1):1–21 (December 1989).
10. G. R. Andrews and R. A. Olsson, Report on the SR Programming Language, Version 1.1. Technical Report CSE-89-11, University of California, Davis, Division of Computer Science (May 1989).
11. G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend, An Overview of the SR Language and Implementation, *ACM Transactions on Programming Languages and Systems*, **10**(1):51–86 (January 1988).

12. American National Standards Institute, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A (1983).
13. N. Gehani and W. D. Roome, *The Concurrent C Programming Language*, Silicon Press, New Jersey (1989).
14. C. M. McNamee, Toward the Optimization of Interprocess Communication Mechanisms, Technical Report CSE-89-24, University of California, Davis, Division of Computer Science (September 1989).
15. T. Hikita and K. Ishihata, A Method of Program Transformation Between Variable Sharing and Message Passing, *SOFTWARE—Practice and Experience*, **15**(7):677–692 (July 1985).
16. P. N. Hilfinger, Implementation Strategies for Ada Tasking Idioms, *Proc. of the AdaTEC Conference on Ada*, pp. 26–30 (October 1982).
17. A. Jones and A. Ardo, Comparative Efficiency of Different Implementations of the Ada Rendezvous, *Proc. of the AdaTEC Conference on Ada*, pp. 212–223 (October 1982).
18. S. Vestal, Mixing Coroutines and Processes in an Ada Tasking Implementation, *Ada Letters*, **9**(2):90–101 (March/April 1989).
19. D. R. Stevenson, Algorithms for Translating Ada Multitasking, *SIGPLAN Notices*, **15**(11):166–175 (November 1980).
20. P. J. Courtois, F. Heymans, and D. L. Parnas, Concurrent Control with Readers and Writers, *Communications of the ACM*, **14**(10):667–668 (October 1971).
21. R. A. Olsson, Issues in Distributed Programming Languages: The Evolution of SR, Technical Report TR 86-21, University of Arizona, Department of Computer Science, Ph.D. Dissertation (August 1986).
22. T. D. Newton, An Implementation of Ada Tasking, Technical Report CMU-87-169, Carnegie-Mellon University, Department of Computer Science (October 1987).
23. R. A. Olsson, Using SR for Discrete Event Simulation: A Study in Concurrent Programming, *SOFTWARE—Practice and Experience* **20**(12):1187–1208 (December 1990).
24. M. S. Atkins and R. A. Olsson, Performance of Multitasking and Synchronization Mechanisms in the Programming Language SR, *SOFTWARE—Practice and Experience*, **18**(9):879–895 (September 1988).
25. R. M. Clapp, L. Duchesneau, R. A. Volz, T. N. Mudge, and T. Schultze, Toward Real-time Performance Benchmarks for Ada, *Communications of the ACM*, **29**(8):760–781 (1986).
26. J. Hennessy, Symbolic Debugging of Optimized Code, *ACM Transactions on Programming Languages and Systems*, **4**(3):323–344 (July 1982).
27. H. Garcia-Molina, F. Germano, and W. H. Kohler, Debugging a Distributed Computing System, *IEEE Transactions on Software Engineering*, **SE-10**(2):210–219 (March 1984).