CONCURRENT PROGRAMMING LANGUAGE SUPPORT FOR INVOCATION HANDLING: AN EXTENSION TO SR

By

Mandy Lai Hung Chung

B.S. (University of Hong Kong) 1990

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Ronald A. Olsson, Chair

Carole M. McNamee

Biswanath Mukherjee

Committee in Charge

1996

Acknowledgements

I am deeply indebted to my advisor, Ronald Olsson. Without his help and guidance, I was not able to finish this work. I thank him for all the time and energy that he has spent working with me. I must also thank him for spending his holidays reading drafts of this thesis even when he was sick. He is an outstanding advisor and mentor. I feel very privileged to have worked with him.

I would also like to thank the other members of my committee, Carole M. McNamee and Biswanath Mukherjee, for their valuable comments on this thesis.

Finally, I would like to thank my family and friends for their support and constant encouragement, especially Calvin, for his patience and understanding and also for his great assistance in the performance measurement for this work.

Contents

Chapter 1

Introduction

Many concurrent programming languages provide mechanisms that allow processes to communicate via some form of message passing. For example, rendezvous is provided in a number of languages, such as Ada [18], Concurrent C [19], CSP [22], Orca [6], and SR [3]; Concurrent C and SR provide both asynchronous and synchronous message passing facilities. Concurrent programming languages provide mechanisms that represent a communication channel (or *operation* [2]) and for generating (or invoking) and servicing invocations of operations.

Invoking and servicing operations are key mechanisms in concurrent programming languages. Many such languages provide mechanisms that can service invocations from one of several operations. These mechanisms may also allow explicit control in selecting which operation to service or in selecting which invocation for a particular operation to service. For example, Ada provides `select/accept` and SR provides input statements. Some languages, such as Concurrent C and SR, allow the selection of which invocation to service to be based on the values of an invocation's parameters. Some also provide a mechanism that gives the number of pending invocations for an operation. For example, SR has a '?' operator whereas Ada has a `COUNT` attribute associated with each entry, Ada's equivalent to an operation.

No existing mechanisms, however, provide a simple and efficient way to examine pending invocations of an operation or allow the selection decision to be based on more than just a single invocation's parameters. As will be discussed later, examining invocations and "cross-invocation" or "cross-operation" selection are important in a number of real applications, including debugging, visualization, and scheduling algorithms. Solutions to such problems using current mechanisms often result in cumbersome and inefficient code.

This research studies how to provide better support for handling invocations in concurrent programming languages. To make our work concrete, we use SR as the base language for our research. Although the variety of synchronization mechanisms in SR provides great flexibility [7, 8], current mechanisms for selection control are not expressive enough. Even those mechanisms that allow invocation parameters to be examined or that can query the number of pending invocations are limited. We designed several new mechanisms that supplement the current SR mechanisms; the key new mechanism is called **rd** (pronounced as "R–D"). To facilitate the evaluation of our work, we also implemented an experimental version of SR that includes our new mechanisms as extensions to SR. In this thesis, we refer to our experimental version of SR as SRR (SR with **rd**), and to the standard SR simply as SR. We evaluated our work both qualitatively and quantitatively. For the former, we explored the programming style that results from using each of the new mechanisms. For the latter, we measured the individual performance of the new mechanisms as well as the performance of more realistic programs that consist of many language elements.

This thesis describes how our new mechanisms make invocation handling more flexible and demonstrates that by using a number of representative examples. The new mechanisms have simple but expressive semantics. The overall performance of our new mechanisms is reasonable. Our new mechanisms are very efficient in simple cases. To examine an invocation, the new mechanism takes only 39% of the execution time for SR's mechanisms. However, the most general cases are necessarily expensive to implement. Several optimizations are applied to the new mechanisms to increase their efficiency. Two of SR's existing mechanisms — input statements and the '?' operator — can be simulated using the new mechanisms. In general, the simulation using the new mechanisms of an input statement is slower than the real input statement by 46%. On the other hand, simulations using our new mechanisms of an input statement with a synchronization or scheduling expression are in some cases more efficient than the original input statement. In addition, the performance of some common applications can be improved using the new mechanisms. Some optimizations similar to those of our new mechanisms can be applied to improve the current implementation of SR.

Our results illustrate the tradeoff between the expressiveness of a language and its implementation cost. Current mechanisms for invocation handling are not expressive enough. However, some of this expressive power is already present in the SR implementation. Our new mechanisms provide the programmer with ways to access such power and make the language more expressive. However, as indicated earlier, the most general case of the new mechanisms is necessarily costly to implement. Furthermore, the implementation of the new mechanisms does affect the implementation of the existing mechanism for invocation servicing (i.e., the input statement), making it less efficient in SRR than its original implementation.

Very little work in the literature has addressed *examining* invocations. In this thesis, we study this problem in depth. We have resolved several semantic problems in defining our new mechanisms that allow examination of invocations. Our results show that the semantics and the implementation of the new mechanisms are reasonable. Although we designed the new mechanisms based on SR and implemented them as an extension to SR, the underlying concept is language-independent. Our work is applicable to other concurrent languages, in both the design and implementation.

The rest of this thesis is organized as follows. Chapter 2 gives a brief overview of relevant SR language background. Chapter 3 illustrates the shortcomings of invocation handling mechanisms in SR and Chapter 4 describes similar shortcomings in other languages and systems. Chapter 5 describes the new mechanisms in SRR that provide additional support for invocation handling. It defines the syntax and semantics of the new mechanisms and provides several small examples to illustrate their use. Chapter 6 presents several larger and more realistic examples that use the new mechanisms to demonstrate how the problems raised in Chapter 3 are resolved in SRR. Chapter 7 describes our implementation of the new mechanisms and discusses some of the implementation issues. Chapter 8 provides some measurements of the cost of individual mechanisms as well as the performance of more realistic programs. Chapter 9 discusses some of the design and implementation decisions that influenced the development of the new mechanisms. It also identifies some optimization possibilities in the current implementation of SRR as well as SR and discusses the applicability of our work to other languages. Finally, Chapter 10 summarizes our work and discusses possible future research.

Chapter 2

Background: The SR Language

The SR concurrent programming language [3] provides a variety of mechanisms for writing parallel and distributed programs. Operations and the different ways to invoke and service them form the bases for SR's concurrent programming mechanisms. We focus here on SR's primitives relevant to operations. Refer to [3, 4] for a complete description of the language, for examples, and for further implementation details. The SR implementation runs on most UNIX systems and is in the public domain. It has found use in many research and teaching projects; e.g., see [3, 5, 21, 30, 31].

2.1 Operations

An operation can be considered a generalization of a procedure: it has a name, and can take parameters and return a result. An operation is serviced in one of two different ways — by a **proc** or by **in** statements — and can also be invoked in two ways — by **call** and **send**. A **proc** resembles a conventional procedure in having parameters and a body, but invoking a **proc** causes a new process to be created to service that invocation. The effects of executing the various combinations of **call**/**send** and **proc**/**in** are as follows:

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

2.2 Operation Invoking

An operation is invoked explicitly using a **call** or **send** statement. The general form is

> **call** *operation* (*formal_ids*)

or

> **send** *operation* (*formal_ids*)

Execution of a **call** terminates once the code associated with the operation has been executed and a result, if any, returned. Its execution is thus synchronous. Execution of a **send**, on the other hand, is asynchronous. It terminates when the target process has been created (if the operation is serviced by a **proc**), or when the invocation has been queued for the process servicing the operation (if the operation is serviced by an **in** statement).

2.3 Invocation Servicing

In this thesis, we focus on operations serviced by **in** statements. The general form of an **in** statement is

> **in** *op_command* [] *op_command* [] ... **ni**

Each operation command specifies an operation to service, an optional synchronization clause, an optional scheduling clause, and a block of code. An operation command has the general form

> *operation* (*formal_ids*) [**returns** *result_id*] [**st** *synch_expr*] [**by** *sched_expr*]
> −> *block*

A process executing an **in** statement is delayed until some invocation is selectable. An invocation is *selectable* if the boolean-valued synchronization expression for the corresponding operation is true (or omitted). In general, the oldest selectable invocation is serviced. However, if the corresponding operation contains a scheduling expression, the invocation that is serviced is the oldest one that is selectable and that also minimizes the scheduling expression. Both the synchronization and scheduling expressions can reference the invocation parameters so that selection can be based on their values. To illustrate, consider the code in Figure 2.1 for a server process that implements a shortest-job-next request allocator. The first arm of the **in** statement uses a synchronization expression

```
# shortest-job-next request allocator
process server
   var free := true
   do true ->
      in request(size) st free by size -> free := false
      [] release() -> free := true
      ni
   od
end
```

Figure 2.1: *Shortest-job-next request allocator.*

(**st** clause) to accept an invocation of `request` only when the server is free, and uses a scheduling expression (**by** clause) to select the shortest job among those acceptable invocations. The scheduling expression associated with `request` uses the invocation parameter `size` to determine which job's size is the smallest. Hence, on each iteration of the loop, the server process will service the invocation of `request` with the smallest `size` — but only if `free` is true — or it will service an invocation of `release`.

The final arm of an **in** statement can optionally be an **else** arm. The statements associated with this arm will be executed if no invocation is selectable. For instance, consider the following code

```
# service all invocations of f for which x = 3
do true ->
   in f(x) st x = 3 ->
        ...
   [] else ->
      exit
   ni
od
```

This loop services all pending invocations of `f` whose parameter `x` is equal to 3. Each iteration of the loop services one such invocation, if there is one; otherwise, the else arm is executed and exits the loop.

2.4 Shared Operations

Operations can be shared (invoked or serviced) by more than one process. As an example of a shared bag of tasks using a shared operation, consider the matrix multiplication problem in which the product of two `NxN` integer matrices `a` and `b` is computed and stored in the result matrix `c` by `W` `worker` processes. The following code implements the `worker` processes that share a bag of tasks.

```
process worker(id := 1 to W)
   var i: int              # index of row of c to compute
   do true ->
      receive bag(i)
      fa j := 1 to N ->
         var inner_prod := 0
         fa k := 1 to N ->
            inner_prod +:= a[i,k]*b[k,j]
         af
         c[i,j] := inner_prod
      af
   od
end
```

The solution to the matrix multiplication problem employs an operation, `bag`, to represent the shared bag of tasks. Each task in `bag` represents a row index of the result matrix `c`. Each `worker` process repeatedly receives a row index `i` from `bag` and computes `N` inner products, one for each element of row `i` of result matrix `c`. The **receive** statement is an abbreviation for a simple form of an **in** statement. For example, the above **receive** statement is equivalent to

```
in bag(x) -> i := x ni
```

Note that when a worker attempts to receive from an empty `bag`, it blocks waiting for new invocations to arrive. Thus, the computation terminates (in deadlock) when all `worker` processes are blocked; i.e., all tasks have been processed by workers.

2.5 Arrays of Operations

SR also provides arrays of operations, which can be serviced by **in** or **receive** statements. A particular element of an array of operations is serviced by specifying its index. For example, suppose there are `N` different kinds of requests and each kind of request is serviced by a different scheduler. Consider the following program fragment in which each scheduler process services one kind of job, i.e., one element of the array `request`.

```
op request[N] (x: int)
process scheduler(kind := 1 to N)
   ...
   in request[kind](x) -> ... ni
   ...
end
```

An input statement can service any one of a group of elements of an operation array by specifying a quantifier. For example, a job scheduler can service any kind of request as follows:

```
in (kind := 1 to N) request[kind](x) -> ... ni
```

The above **in** services an invocation from any one element of `request[1:N]`. The value of the quantifier variable `kind` is accessible within the operation arm of `request`; for example, `kind` can be used within the command block. Its value indicates which element of the operation array is being serviced.

2.6 Other Constructs

Several other SR constructs dealing with synchronization are relevant to our work. SR's '?' operator returns the number of invocations currently pending for an operation. For example, it can be used to control a loop that services all pending invocations of operation `f`.

```
do ?f > 0 ->
   in f(x) ->
      ...
   ni
od
```

As another example, the '?' operator can be used to give preference in servicing one operation over another. For example, Figure 2.2 shows a job scheduler that gives preference to interactive requests over batch requests.

```
in interactive(...) -> ...
[] batch(...) st ?interactive = 0 -> ...
ni
```

Figure 2.2: *Job Scheduler.*

Three other statements affect invocation servicing within an **in** statement. A **return** statement transmits results to the invoker, if the operation was called, and terminates execution of the **in** statement. SR's **reply** statement allows a servicing process to send an "early reply" to its invoker transmitting results to the invoker. After execution of a **reply** statement, the invoking and servicing processes can both continue their executions. A **forward** statement defers replying to a called invocation and instead passes on this responsibility to another operation. It does so by generating a new invocation. The forwarding process continues execution following **forward**.

Chapter 3

Shortcomings of SR's Invocation Handling

Although invocation handling in SR is quite flexible, it is limited in two key ways:

- No *peeking*: Invocations can be examined only by actually servicing the invocation (or by employing very obscure and inefficient synchronization and scheduling expressions).

- Single-invocation based decisions: When choosing which invocation to service, only one invocation at a time can be considered within a synchronization or scheduling expression.

These limitations make it difficult to solve common programming situations encountered in program visualization and debugging as well as a number of more sophisticated scheduling scenarios.

Historically, the synchronization and scheduling expressions (**st** and **by** clauses) as well as the '?' operator were added to SR to provide linguistic mechanisms to examine invocation queues and to control selection of invocations to service. Although they are sufficient to solve many problems, they are limited in the ways noted above. The only way to examine an invocation in SR is to actually service it, i.e., remove it from the queue of invocations and, if it is not desirable to service it, return it to the invocation queue by re-invoking the operation. As a result, code that needs to examine invocations is cumbersome and inefficient. Cross-invocation or cross-operation selection also requires examination of invocations. To implement this kind of scheduling problems using current SR's mechanisms often requires changes in the interface. That is, besides modifying the servicing code for the server, the code for the clients often needs to be changed as well to adapt the modified interface, for example, to invoke a new operation or to invoke the same operation name but with different parameterization. The following examples illustrate the kinds of problems where examining invocations is important.

3.1 Peeking at Pending Invocations

3.1.1 First-in, First-out Order Peeking

During debugging, programmers often want to display pending invocations and their parameter values for a given operation. Similarly, visualizing or animating the execution of a concurrent program often requires displaying pending invocations. Such *peeking* should not disturb the invocations or their order on the operation's queue.

Consider, for example, an elevator controller simulation in which each person is represented as a process and the elevator controller is represented as a separate process. When a person process wants to take the elevator to another floor, it invokes operation `f` in the elevator controller; the controller services the invocation when it schedules the elevator to pick up this passenger. In a visualization of an execution of the simulation, the picture should show which people are waiting on which floors, i.e., the currently pending invocations for `f`.

One way to display all invocations of `f` is shown in Figure 3.1. Note how it services each invocation of `f` and then forwards that invocation back to `f`. This code uses **forward** (rather than **send**) to delay replying to the invoker until the invocation is really serviced rather than just examined. Its cost is high even though the invocation queue, in effect, does not change over this program fragment. Each invocation of `f` being examined is

```
op f(x: int)
var count := ?f
fa i:= 1 to count ->
   in f(x) ->
      write(x) # print out parameter
      forward f(x)
   ni
af
```

Figure 3.1: *Examining an operation while preserving arrival order of invocations.*

serviced once and re-invoked once.

A major shortcoming of the code in Figure 3.1 is that it does not necessarily preserve the arrival order of invocations of `f`. In particular, new invocations of `f` that arrive during execution of that code can end up at the front of the queue rather than at the back, or interspersed with old invocations. Thus, the seemingly simple act of examining invocations can have a nasty side effect that can result in confusion in debugging and can introduce unfairness (and possibly starvation) into how invocations are serviced. Note that we are concerned with the order in which invocations arrive rather than the order in which they are invoked. Invocations are serviced in in FCFS order based on their arrival order if there are more than one selectable invocation. For example, suppose process A invokes an operation `f`, say `f(A)`, before process B invokes `f`, `f(B)`. The invocation `f(B)` could arrive earlier in the invocation queue than the invocation `f(A)` (possibly due to network delay). In that case, if these two invocations both are selectable to service, `f(B)` will be serviced first instead of `f(A)` although `f(A)` is sent first.

To illustrate, consider the pending invocations of `f` at three different execution states shown in Figure 3.2. Each square represents one pending invocation of `f` containing the

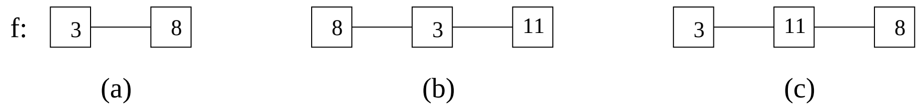

Figure 3.2: *Pending invocations of* `f` — *(a) before entering the loop (b) after the first iteration (c) after the second iteration.*

indicated value of the invocation parameter `x`. Initially, before examining `f` and executing the **fa** loop in Figure 3.1, invocations `f(3)`[1] and `f(8)` are pending as shown in Figure 3.2(a) and `count` is hence initialized to 2. During the first iteration of the **fa**, invocation `f(3)` is examined (via **in**) and then forwarded back to `f`. At the end of this iteration, a new invocation `f(11)` arrives and is appended onto the invocations queue, resulting in the invocation queue shown in Figure 3.2(b). The second iteration, which is the last iteration of the **fa**, examines invocation `f(8)` and then forwards it back to `f`. As a result, after the **fa** terminates, the new invocation `f(11)` ends up at the middle of the invocation queue (rather than at the back), which is shown in Figure 3.2(c). In this example, the first execution of the code in Figure 3.1 displays the correct output as described above (output of 3 and 8); however, printing the invocations again will show a different initial subsequence (output of 3, 11, and 8 as seen in Figure 3.2(c)).

In addition, this code has one potential programming pitfall. As shown in Figure 3.1, a separate variable `count` is needed to save the number of pending invocations to be examined before executing the **fa**. This is necessary because the final expression in **fa** is evaluated once to determine if the loop should be executed initially and again after each iteration of the loop. The variable `count` is used to avoid the value of the final expression in the **fa** being changed during the execution of **fa**. A potential problem will arise if the final expression in the **fa** is replaced with `?f`. In particular, when new invocations of `f` arrive during execution of that code, new invocations can be interspersed with the examined invocations. That code will most likely not output the correct result but reprint some previously examined invocation. As seen in Figure 3.2, a new invocation `f(11)` arrives at the end of the first iteration. Suppose the final expression in the **fa** is replaced with `?f`. During the second iteration, `?f` is reevaluated and returns 3. Therefore, unlike what we described above, the **fa** does not terminate after printing invocation `f(8)` in the second iteration. The third iteration will examine `f` in the execution state that is shown

[1]The notation `f(3)` might introduce potential confusion because pending invocations of `f` do not necessarily have unique parameter values (e.g., `f` can be invoked twice with parameter 3). In our examples, however, each operation will be invoked only with unique parameter values to avoid this potential confusion.

in Figure 3.2(c) and thus print invocation `f(3)` (giving output of 3, 8, and 3), which has already been examined during the first iteration.

The arrival order of invocations of `f` can be preserved with additional programming effort, as shown in Figure 3.3. That code maintains invocations in their arrival order

```
op f(x: int)
op aux_f(x: int)
# examined invocations of f are kept in aux_f
var count := ?aux_f
fa i := 1 to count ->
   in aux_f(x) ->
      write(x) # print out parameter
      forward aux_f(x)
   ni
af
# new invocations of f are forwarded to aux_f after printing out parameter
do ?f > 0 ->
   in f(x) ->
      write(x) # print out parameter
      forward aux_f(x)
   ni
od
```

Figure 3.3: *Examining an operation while preserving the arrival order of invocations.*

by using an auxiliary operation `aux_f`. Invocations of `f` are divided into two groups: invocations that have been examined and those that have not. All previously examined invocations are kept in `aux_f` while the rest remain in `f`. The solution consists of two loops, a **fa** and a **do**. To examine all invocations of `f`, the code first prints all examined invocations (by the **fa**) and then prints all newly arrived invocations (by the **do**). The examined invocations are then forwarded to `aux_f` in their arrival order. The solution employs the code in Figure 3.1 to display all examined invocations in `aux_f`. It does, in this case, preserve the arrival order of invocations because no invocations would arrive at `aux_f` while being examined.

The code in Figure 3.3 has two drawbacks. First, it is very costly, like the code in Figure 3.1, because it services each invocation and then forwards it. Second, the original servicing code for `f` needs to be modified (which is generally, but not always, possible) so that it services invocations from `aux_f` first and then invocations from `f`.

3.1.2 Multiple Servicing Processes

In Figures 3.1 and 3.3, we assumed that the process examining the operation `f` is the only process servicing invocations of `f`. If other processes are servicing invocations of `f` at about the same time as the examining process, some invocation might be removed while the examining process assumes that invocation still exists. For example, in Figure 3.3, whenever examining `aux_f`, the process first initializes the variable `count` to the number of pending invocations of `aux_f`. If another process services one pending invocations from `aux_f` while `aux_f` is being examined, the **fa** will then not terminate. This problem can be solved, but at greater complexity and expense.

One intuitive and straightforward way to handle the multiple servicing processes problem is to disallow the concurrent execution of the servicing code for `f` and the examining code for `f` by using "locking". This can be accomplished by enclosing such code in critical sections, using a semaphore operation to implement mutual exclusion.

```
P(mutex)                                    P(mutex)
# servicing code                            # examining code
  ...                                         ...
V(mutex)                                    V(mutex)
```

Processes have to obtain permission to service or examine an operation. A semaphore `mutex` is initialized to 1 so that either the servicing code or the examining code for `f` can be executed at a time. Processes wait on the semaphore in first-come, first-served order based on the order in which they execute `P` operations. Therefore, for invocation servicing, fairness is guaranteed as long as the servicing code is fair. However, a major shortcoming of this approach is that it requires changes in the interface, which is not always possible. Another drawback is that it is easier for deadlock to occur. In particular, if the servicing code contains an **in** statement without an else command, the servicing code might block on the **in** after obtaining the `mutex` when no invocations are selectable for service. Finally, as the servicing processes and the examining process are forced to execute sequentially, the level of concurrency is reduced and hence the execution may be slower.

Another way to solve this problem is shown in Figure 3.4. That code is similar to the code in Figure 3.3 except that a temporary operation `temp` is used for examining invocations from `aux_f`. This operation `temp` is local to the examining process only. One major shortcoming of that code is that the order of invocations servicing is nondeterministic. The servicing processes might not service the invocations that have been forwarded to `temp` but are not yet moved back to `aux_f`. Instead, they service only invocations that currently stay in `aux_f`. This nasty side effect can introduce unfairness and possibly starvation into how invocations are serviced.

```
op f(x: int)
op aux_f(x: int)
op temp(x: int)
# examined invocations of f are kept in aux_f
do ?aux_f > 0 ->
   in aux_f(x) ->
      write(x) # print out parameter
      forward temp(x)
   ni
af
do ?temp > 0 ->
   in temp(x) ->
      forward aux_f(x)
   ni
od
# new invocations of f are forwarded to aux_f after printing out parameter
do ?f > 0 ->
   in f(x) ->
      write(x) # print out parameter
      forward aux_f(x)
   ni
od
```

Figure 3.4: *Multiple servicing processes.*

To illustrate this side effect clearly, consider how a process services an invocation of `aux_f` while another process is executing the code in Figure 3.4. Figure 3.5 shows invocations of operations `aux_f` and `temp` during execution of that code. No invocations of `f` arrive during the execution. Suppose four invocations are pending for `f` and they have

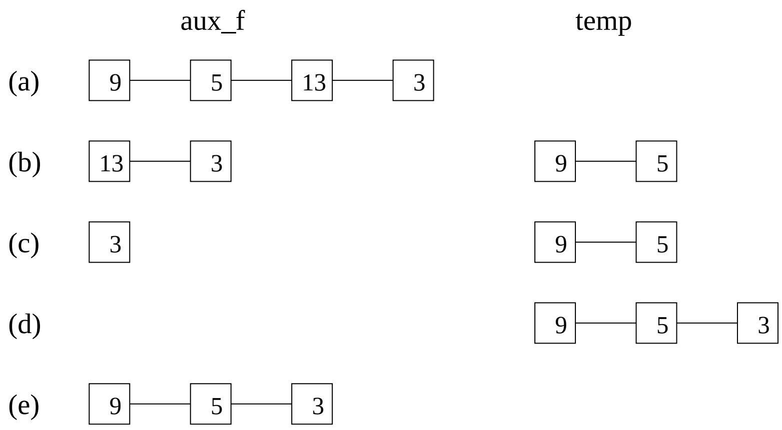

Figure 3.5: *Execution of examining invocations of* `f` *and* `aux_f`

already been examined and forwarded to `aux_f` earlier as shown in Figure 3.5(a). They are now being examined again. Figure 3.5(b) shows the operations `aux_f` and `temp` after two invocations `f(9)` and `f(5)` have been examined. These two invocations have been moved from `aux_f` to `temp`. At this time, another process wants to service the first pending invocation of `aux_f`. This servicing process thus services invocation `f(13)` (as shown in Figure 3.5(c)) rather than `f(9)` which is the oldest pending invocation of `f`. When the examining process continues its execution, invocation `f(3)` is then examined and forwarded to `temp`. Figure 3.5(d) shows that the arrival order of invocations is preserved after they have been examined. The invocations are then forwarded back to `aux_f` in the same order as shown in Figure 3.5(e). This example illustrates that the servicing process is no longer guaranteed to service invocations in FCFS order when another process is executing the code in Figure 3.4; i.e., it services `f(13)` instead of `f(9)` in the above example. Furthermore, an invoking process might be starved. For example, if invocations of `aux_f` are always serviced after `f(9)` have been forwarded to `temp`, the invocation `f(9)` would then never get serviced.

3.1.3 Threshold Order Peeking

In Section 3.1.1, we discussed the problem of peeking at pending invocations in their arrival order. All pending invocations are examined as one whole group. However, some other applications may require examining invocations in several different groups, each of which for example, contains pending invocations whose parameter values are above a certain threshold, displaying one group of invocations at a time. Once all invocations of a group have been examined or no invocations of that group are pending, the threshold is then lowered to examine the next group of invocations.

Consider, for example, an airline boarding simulation in which each passenger is represented as a process and the plane controller is also represented as a separate process. When a passenger process checks in for a flight, it invokes operation `f` in the plane controller; when the plane lands at the destination, the controller schedules the passenger to disembark and services the invocation. When the plane is ready to take off, passengers are called for boarding according to their seat numbers. Passengers who sit with seats at the rear of the plane are boarded first, one row at a time. The controller examines `f` and starts boarding until all passengers go into the plane. Some passengers may arrive late after their seat numbers have been called. Such passengers are allowed to board after they arrive. The code in Figure 3.6 implements the airplane boarding procedure for a total of `N` passengers with initial threshold of `T`, where `ROW` is the number of seats in a row.

```
threshold := T  # initial threshold
count := 0
do count < N ->
   in f(seat) st seat >= threshold ->
      count++
      forward boarding(seat)
   [] else ->
      if threshold > 0 ->
         threshold -:= ROW
      fi
   ni
od
```

Figure 3.6: *Airplane boarding.*

Note that a synchronization expression is used in the **in** to select those passengers with seat number above a certain threshold for boarding at a time. The threshold is lowered when no such passengers are waiting.

The code in Figure 3.6 has three drawbacks. First, an additional operation `boarding` is employed; similar to the code in Figure 3.3, the original servicing code for `f` needs to be modified so that it services invocations from `boarding` instead of `f`. If the original servicing code for `f` is to be retained, all invocations need forwarding back to `f` after boarding is done. Second, the cost is high because it services each invocation and then forwards it. Third, the **do** loop is busy waiting until `N` passengers are boarded.

3.2 Cross-invocation and Cross-operation Scheduling

As noted above, SR's input statement does not directly allow the decision as to which invocation is to be serviced to be based on multiple invocations or operations, a more general form of peeking. A number of fairly common and important problems, however, need exactly that ability to be implemented easily and efficiently. Below we present a few representative examples.

3.2.1 Lottery Scheduling

Consider the problem of implementing a lottery scheduling algorithm for resource management [33] in a system of user processes, which make requests of a manager process. Each request includes the user's identifier, *uid*. The manager process picks a uid randomly from the uids of all requests that are pending. In SR pseudocode, one possible structure for the manager process code is

```
do true ->
   in request(uid,...)
      st uid = "the uid chosen randomly from those in all pending requests" ->
      # service this request for some quantum
        ...
      if "more work left before this request can finish" ->
         # needs further service, so put it back on queue
         forward request(uid,...)
      fi
   ni
od
```

The above synchronization expression cannot be expressed directly in SR, so indirect means must be used. Figure 3.7 shows the code that implements the lottery scheduler.

```
var count := ?request
fa i := 1 to count ->
   in request(uid,...) ->
      # save the uid for later selection
        ...
      forward request(uid,...)
   ni
af
# set theuid to the randomly chosen one
theuid := random_uid()
in request(uid,...) st uid = theuid ->
   /* body as before */
ni
```

Figure 3.7: *Lottery scheduler.*

That code, which is very inefficient, uses a loop that examines all invocations before the **in** statement, much like the code in Figures 3.1 and 3.3, and sets `theuid` to the randomly chosen one; the procedure `random_uid` chooses and returns one uid randomly from those uids saved in the **fa** loop. The **in** statement then uses that variable in the synchronization expression to service that invocation.

3.2.2 Extended Job Scheduling

As another example, consider extending the job scheduler in Figure 2.2 so that it gives preference to interactive requests unless there is a batch job submitted by the superuser (whose `uid` is 0). As with the lottery scheduler, there is no direct way in SR to solve this, but there are four noteworthy indirect solutions.

- **Pre-in Loop**

  First, as shown in Figure 3.8, a "pre-**in** loop", in the style suggested for the lottery scheduler, can be used to determine whether any superuser batch invocations are pending. Note that an auxiliary operation `aux_batch` is used in that code. Before servicing an invocation, the "pre-**in** loop" first counts the number of superuser batch invocations and then forwards all `batch` invocations to `aux_batch`. It services the superuser batch invocations from `aux_batch`, if any. If there are no such invocations pending (i.e., `superbatch = 0`), it then services an invocation, as it did originally.

```
superbatch := 0
do true ->
   # check whether there are any superuser batch jobs
   do true ->
      in batch(uid,...) ->
         if uid = 0 -> superbatch++ fi
         forward aux_batch(uid,...)
      [] else ->
         exit
      ni
   od
   in aux_batch(uid,...) st uid = 0 ->  # superuser batch jobs
      superbatch--
      ...
   [] interactive(uid,...) st superbatch = 0 -> ...
   [] aux_batch(uid,...) st superbatch = 0 and ?interactive = 0 -> ...
   ni
od
```

Figure 3.8: *Extended job scheduler using a pre-in loop to examine superuser batch invocations.*

The above "pre-**in** loop" only examines invocations of `batch` without actually servicing them. Another kind of "pre-**in** loop" is also possible for the extended job scheduler, as shown in Figure 3.9. In contrast to the code in Figure 3.8,

```
superbatch := 0
do true ->
   # service all superuser batch jobs first
   do true ->
      in batch(uid,...) st uid = 0 ->
         ...
      [] else ->
         exit
      ni
   od
   in interactive(uid,...) -> ...
   [] batch(uid,...) st ?interactive = 0 -> ...
   ni
od
```

Figure 3.9: *Extended job scheduler using a pre-in loop to service superuser batch invocations.*

this "pre-**in** loop" services all superuser batch invocations instead of just examining them. No additional operation is needed in this code. This code is more efficient because invocations are not forwarded to another operation. However, this kind of "pre-**in** loop" can only be used when a synchronization expression in **in** can be used to identify which invocations to service. It could not be used, for example, for the lottery scheduler in Section 3.2.1 because all requests must be examined before selecting a request to service and a synchronization expression could not specify such selection.

- **Splitting**

  Second, the batch operation can be *split* [10], e.g.,

```
in interactive(uid,...) st ?superbatch = 0 -> ...
[] batch(uid,...) st ?superbatch = 0 and ?interactive = 0 -> ...
[] superbatch(uid,...) -> ...
ni
```

In this case, the interface has to be changed and the superuser process should invoke the operation `superbatch` instead of `batch` to submit a batch job.

- **Combining**

  Third, the batch and interactive operations can be *combined* into an array of operations [29], say `request`, with indices ranging over `SUPERBATCH`, `INTERACTIVE`, and `BATCH`, e.g.,

```
in (kind := SUPERBATCH to BATCH) request[kind](uid,...)
    st myturn(kind) ->
   ...
ni
```

This **in** will service a single invocation of `request`. Because the order in which elements of `request` are checked is nondeterministic, the procedure `myturn` is used to enforce the desired ordering by checking whether any invocations with higher priority are pending, e.g.,

```
proc myturn(kind) returns ans
   ans := true
   # are there any pending higher priority invocations?
   # check up through kind's predecessor
   fa k := SUPERBATCH to pred(kind) ->
      if ?request[k] > 0 ->
         ans := false
         return
      fi
   af
end
```

The quantifier variable `kind` in the **in** statement is passed to `myturn` as a parameter. This procedure returns true if no invocations with priority higher than `kind` are pending; otherwise, it returns false. The **in** statement invokes `myturn` for each pending invocation until a selectable invocation with true return value from `myturn` is found, i.e., the highest priority invocation that is pending.

Similarly, the batch and interactive operations can be combined into a single operation with an additional parameter, indicating the priority of each request e.g.,

```
in p_request(uid,...,priority) by -priority -> # prioritized requests
   ...
ni
```

This **in** will service the invocation of `p_request` whose priority is the highest.

- **Nesting**

  Fourth, the batch and interactive operations can be serviced by nested **in** statements, e.g.,

  ```
  in batch(uid,...) st uid = 0 -> ... # any superuser requests?
  [] else ->
     in interactive(uid,...) -> ...
     [] batch(uid,...) st ?interactive = 0 -> ...
     ni
  ni
  ```

  This solution first checks whether any invocations of `batch` are from the superuser. If not, it then services an invocation of `interactive` or an invocation of `batch` if no interactive invocations are pending.

- **Preferences**

  Another approach is to use preferences, as proposed in [16, 17, 29] and implemented in occam [9] and in an experimental version of SR [15]. For example, the extended job scheduler can be written

  ```
  in [0] batch(uid,...) st uid = 0 -> ...
  [] [1] interactive(uid,...) -> ...
  [] [2] batch(uid,...) -> ...
  ni
  ```

  Here, we denote the preference assigned to each arm by the integer expression, lower-valued for higher-priority, at the start of each arm.

These approaches — "pre-**in** loop", splitting, combining, nesting, and preferences — have disadvantages. The "pre-**in** loop" is costly since it involves generating new invocations (see Section 3.1.1). The other approaches work well only when the number of choices is not too large, because each choice translates into a distinct arm of the input statement or a distinct operation index. Splitting or combining requires changes in the interface. Combining will not work easily if the original operations have different parameterizations. Nesting also incurs extra overhead due to its executions of separate **in** statements. Preferences require additional implementation effort and costs [15, 29]. "Pre-**in** loop", splitting, nesting, or preferences can require code to be repeated.

3.2.3 Other Scheduling Scenarios

Below are some other fairly common and important problems that require cross-invocation scheduling.

Two-dimensional scheduling: picks the least invocation based on invocation parameters x and y according to the usual ordering among pairs, i.e., $(x_1, y_1) < (x_2, y_2)$ if and only if $x_1 < x_2$ or ($x_1 = x_2$ and $y_1 < y_2$). This type of ordering occurs in priority scheduling, e.g., x represents class of user and y represents size of memory request; it also occurs in updating images, e.g., x and y represent coordinates on a screen.

The two-dimensional scheduler can be written

```
op f(x,y: int)
var z[1:MAX, 1:MAX]: int
# initialize the matrix z to represent the scheduling order
# this initialization needs to be done just once
fa i := 1 to MAX, j := 1 to MAX ->
   z[i,j] := (i-1) * MAX + j
af
...
# service all pending invocations of f
do true ->
   in f(x,y) by z[x,y] ->
      write(x,y) # print out parameters
   ni
od
```

This code assumes that the invocation parameters are of integer type and that their values are bounded (between 1 to `MAX`). A two-dimensional array `z` is predefined to represent the ordering of all possible parameter values so that it is used in the scheduling expression of **in** to select the least invocation according to the predefined order.

That assumption, however, is not always valid in practice; for example, when an invocation parameter is a real number or when the parameter value is unbounded. The memory allocated for the matrix may also be too large to use in practice, even if the above bounded scheduling approach can be employed. Figure 3.10 gives a more general solution to the two-dimensional scheduling. It first finds the smallest `x` parameter value and then

```
do true ->
   in f(x,y) by x ->
      forward f(x,y)
      in f(x2,y2) st x = x2 by y2 ->
         write(x2,y2) # print out parameters
      ni
   ni
od
```

Figure 3.10: *Two-dimensional scheduling without preserving arrival order of invocations.*

selects the invocation whose `y` parameter value is the least among all pending invocations that have that `x` parameter value. However, that solution suffers the same problem as the code in Figure 3.1; it does not necessarily preserve the arrival order of invocations of `f`. As described in Section 3.1.1, it can result in an unfair scheduling.

As shown in Figure 3.11, the arrival order of invocations of `f` can be preserved by using a "pre-**in**" loop, like the code in Figure 3.3, that examines all invocations to find the least invocation first before servicing. Nonetheless, all three of these solutions have their shortcomings as discussed in Section 3.1.

Median scheduling: selects the *median* invocation based on arrival order or on parameter value.

Figure 3.12 shows one way to select the median invocation based on the arrival order. That code uses the procedure `seqno()` to assign a sequence number to each pending invocation of `f` according to its arrival order until an invocation whose sequence number is the median is found and serviced. We define the median sequence number as follows: If the number of pending invocations, say n, is odd, then the median is the one with

```
do true ->
   # examine all invocations to find the least invocation
   count := ?aux_f
   fa i := 1 to count ->
      in aux_f(x,y) ->
         # minpt is a record structure point(x,y)
         if x < minpt.x or (x = minpt.x and y < minpt.y) ->
            minpt := point(x,y)
         fi
         forward aux_f(x,y)
      ni
   af
   do ?f > 0 ->
      in f(x,y) ->
         if x < minpt.x or (x = minpt.x and y < minpt.y) ->
            minpt := point(x,y)
         fi
         forward aux_f(x,y)
      ni
   od
   # service the selected least invocation
   in aux_f(x,y) st x = minpt.x and y = minpt.y ->
      write(x,y) # print out parameters
   ni
od
```

Figure 3.11: *Two-dimensional scheduling preserving arrival order of invocations.*

```
op f(x: int)
var count: int

# assign a sequence number to an invocation
procedure seqno() returns index: int
   index := ++count
end

# service all pending invocations of f
do ?f > 0 ->
   # service median invocation
   median := ?f/2 + 1
   count := 0
   in f(x) st seqno() = median ->
      write(x) # print out parameter
   ni
od
```

Figure 3.12: *Median-arrival scheduling.*

sequence number $\lfloor n/2 \rfloor + 1$. If n is even, there are two medians and we select the one with the higher sequence number. On each iteration to service an invocation, `?f` is called and its return value is used to calculate the median sequence number. Variable `count` is reset to zero before the **in** executes. This is necessary because the procedure `seqno` increments `count` and returns its value as the current sequence number for an invocation. Resetting `count` to zero guarantees that `seqno` assigns sequence numbers starting from 1.

A variation of median-arrival scheduling is to select the median invocation based on parameter value. Figure 3.13 shows a solution to the median-value scheduling. That code employs an additional operation `ordered_f` to keep all pending invocations in non-decreasing order according to their parameter values, and then services the median invocation from `ordered_f` using the median-arrival scheduling approach. The process sorts invocations of `f` using the scheduling expression of the **in**. The code is quite complicated; it is also inefficient because all pending invocations are sorted before an invocation is selected and serviced. This sorting requires invocation servicing and re-invoking to move invocations between `f` and `ordered_f` and also requires multiple accesses to invocations to find the smallest invocation (via the **by** clause of the **in**).

```
op f(x:int)
op ordered_f(x:int)

# assign a sequence number to an invocation
procedure seqno() returns index: int
    index := ++count
end

# service all pending invocations of f
do ?f > 0 ->
   # forward all examined invocations back to f for sorting
   do ?ordered_f > 0 ->
      in ordered_f(x) ->
         forward f(x)
      ni
   od
   # sort all invocations of f using scheduling expression
   # and keep in ordered_f
   do ?f > 0 ->
      in f(x) by x ->
         forward ordered_f(x)
      ni
   od

   # service median invocation in their sorted order
   median := ?ordered_f/2 + 1
   count := 0
   in ordered_f(x) st seqno() = median ->
      write(x) # print out parameter
   ni
od
```

Figure 3.13: *Median-value scheduling.*

The major drawback of the code in Figure 3.13 is that the arrival order of invocations is not preserved as in Figure 3.1. Execution of the code results in the sorted invocation queue. The problem can be solved while preserving the arrival order of invocations using the "pre-**in** loop" as shown in Figure 3.14. This solution uses the code in Figure 3.3 as the "pre-**in** loop", which examines all pending invocations and stores their parameter values. The procedure `median_find` determines and returns the median value among those collected parameter values. It then services the invocation with that median value.

```
count := ?aux_f
fa i := 1 to count ->
   in aux_f(x) ->
      # store x for later median finding
        ...
      forward aux_f(x)
   ni
af
do ?f > 0 ->
   in f(x) ->
      # store x for later median finding
        ...
      forward aux_f(x)
   ni
od
# determine the median value among those parameter x collected
median := median_find()
in aux_f(x) st x = median ->
   ...
ni
```

Figure 3.14: *Median-value scheduling preserving the arrival order of invocations.*

Last-in, first-out scheduling: selects the most recently arrived invocation. Last-in, first-out (LIFO) scheduling is very similar to the median-arrival scheduling in the sense that both are based on the arrival order of invocations. Figure 3.15 shows the code for implementing the LIFO scheduler in which the procedure `seqno()` in Figure 3.12 is used. That code is nearly the same as the code in Figure 3.12 except that the sequence number of the selected invocation is `last` rather than `median`.

```
do true ->
   # service the last pending invocation
   last := ?f
   count := 0
   in f(x) st seqno() = last ->
      write(x) # write out parameter
   ni
od
```

Figure 3.15: *Last-in, first-out scheduling.*

Another way to service the last pending invocation is:

```
# service the invocation with highest sequence number
count := 0
in f(x) by -seqno() ->
   write(x) # write out parameter
ni
```

This code uses a scheduling expression in the **in** to service the invocation with the highest sequence number (rather than a synchronization expression as in Figure 3.15). Note that the variable `count` is also reset to zero in this case. However, unlike the solution in Figure 3.15, `count` is reset to avoid value overflow. In practice, unless the integer size is very small (i.e., the number of bits is small), overflow of the number of invocations should not be a concern.

Chapter 4

Shortcomings of Invocation Handling in Other Languages and Systems

As seen in Chapter 2, SR provides a variety of concurrent programming mechanisms through various combinations of its invoking and servicing mechanisms. In particular, it provides remote procedure call, rendezvous, dynamic process creation, and message passing. Chapter 3 discusses the limitations of invocation handling in SR. This chapter discusses the shortcomings of invocation handling in several other representative concurrent programming languages and systems: Ada, Concurrent C, Linda, and PVM. We focus only on the synchronization mechanisms relevant to our work.

4.1 Ada

4.1.1 Language Background

The Ada programming language provides only one interprocess[1] synchronization mechanism: rendezvous [18]. Entries, similar to SR's operations, are the primary means of communication between processes. A rendezvous occurs when a process issues a (block-

[1] In Ada, parallel processes are called *tasks*.

ing) call to an entry. Another process can service the invocation by executing an `accept` statement. Ada provides a `select` statement that allows the servicing of invocations from one of several entries. The selection among the entries is nondeterministic. The `select` statement can include `when` clauses to explicitly control which entry is chosen to service. However, the `when` clauses cannot access invocation parameters. As in SR, Ada also provides a `COUNT` attribute associated with each entry, which returns the number of invocations waiting at the entry (equivalent to the SR's '?' operator).

To illustrate how these mechanisms are used, consider the following Ada program implementing the job scheduler described in Section 2.6. This code is equivalent to the SR input statement of Figure 2.2.

```
select
   accept interactive(...) do
      # service interactive job
        ...
   end interactive;
or
   when interactive'COUNT = 0 =>
      accept batch(...) do
         # service batch job
           ...
      end batch;
end select;
```

The above `select` statement accepts `batch` requests only if no `interactive` requests are pending; i.e., it gives preference to `interactive` requests over `batch` requests.

4.1.2 Shortcomings

Invocation (entry call) handling in Ada is very limited. Unlike SR, Ada does not permit parameters in an invocation to be used in deciding which invocation to service. This limitation makes it difficult to solve common scheduling problems, even a simple one such as the shortest-job-next request allocator in Figure 2.1. Invocations in an entry must be examined explicitly before selecting an invocation to service. Ada does not provide any mechanism that examines invocations (or entry calls) without servicing them.

A new `requeue` statement is provided in Ada 9X, which has a meaning similar to the SR's **forward** statement. The `requeue` statement defers replying to the caller after examining an invocation and requeues the invocation to another entry. An invocation in an entry can thus be examined using a similar approach used for SR as described in Section 3.1: first remove the invocation from the entry by `accept` and then send it back to the entry by `requeue`. For the same reasons as described for SR's **forward** statement, the expected cost of examining an invocation in an entry is expensive.

4.2 Concurrent C

4.2.1 Language Background

Concurrent C provides both a rendezvous mechanism and an asynchronous message passing mechanism [19]. Transactions are equivalent to SR's operations and Ada's entries. Similar to Ada, transactions are serviced by `accept` statements. A `select` statement is also provided; a `select` statement can service invocations from one of several transactions nondeterministically. Similar to SR, selection of an invocation can be based on values of invocation parameters. By using a `suchthat` clause, to determine whether an invocation is selectable, and a `by` clause, to pick the invocation with the minimum `by` clause's value from among the selectable invocations. The Concurrent C equivalent of the SR's input statement of Figure 2.1 implementing the shortest-job-next allocator is shown below.

```
select {
   accept request(size) suchthat(free) by(size)
      free = false;
or
   accept release()
      free = true;
}
```

Transactions can be declared to be `async` for asynchronous message passing. Invocation of `async` transactions are non-blocking and are serviced in the same way that synchronous transactions are. Concurrent C also provides a function `c_transcount`, similar to Ada's

`COUNT` attribute and SR's '?' operator, that returns the number of pending invocations for a given transaction.

4.2.2 Shortcomings

Concurrent C, being similar to SR, has the same limitations in handling invocations as described in Chapter 3. First, invocations can be examined only by actually servicing the invocation. Second, only one invocation at a time can be considered within a `suchthat` or `by` clause in choosing which invocation to service. The same difficulties arise in solving the problems described in Chapter 3.

4.3 Linda

4.3.1 Language Background

Linda [12, 20] consists of a few primitives to access the communication mechanism called "tuple space". Tuple space (TS) consists of a collection of data records (called tuples) and is logically shared by processes. There are six primitives to access TS: `out`, `in`, `inp`, `read`, `readp`, and `eval`. A tuple is added to the TS by `out`, similar to SR's **send** statement. A process can withdraw a tuple from TS by executing `in`. `in` removes only a tuple that matches its arguments (template) from TS. If no tuple matches the template, the executing process blocks until TS contains one matched tuple. If more than one tuple matches the template, one is selected nondeterministically. `read` is similar to `in` except that it does not remove the matched tuple from TS. `inp` and `readp` are non-blocking variants of `in` and `read` respectively. They return true if there is a matching tuple in TS and false otherwise. `eval` is similar to `out`, but it creates a process to evaluate the expressions specified in the tuple. The tuple is added to TS when the evaluation is complete and the process terminates.

To illustrate how these primitives are used, consider the implementation of barrier synchronization described below. Suppose there are `n` worker processes to complete a task. After some computation, each process must wait until all processes have reached the barrier before continuing execution. Initially, a barrier is set up before the worker processes begin by executing

```
out("barrier",n)
```

When reaching the barrier point, each process executes the following:

```
in("barrier",?val)   # assign tuple value to val
out("barrier",val-1)
read("barrier",0)
```

First, each process decrements the value of `barrier` tuple by using `in` and `out`. The `in` removes the `barrier` tuple and assigns its value to the variable `val` via `?` operator; the following `out` adds the `barrier` tuple back with value `val-1`. The process then blocks until the value of `barrier` tuple becomes zero; i.e., all `n` processes have reached the barrier point.

4.3.2 Shortcomings

Linda provides a `read` primitive to examine a matched tuple from TS, unlike other languages. However, it is nondeterministic, which means that it cannot be used to read through all pending tuples. A new `collect` primitive [11] proposed for Linda allows groups of tuples to be moved between tuple spaces. When several processes execute `collect` and compete for the same tuples, the common tuples are then partitioned between these processes nondeterministically. This primitive, however, does not make the invocation handling more flexible. Tuples can be examined only by actually removing the tuples and then inserting them back to the TS, similar to the approach used in Section 3.1. The `collect` primitive can only help move a group of tuples to another tuple space without using a loop. An alternative primitive, `forall`, is also proposed but rejected in [11]. This primitive allows iteration through elements of a tuple space, which will be further discussed in Section 9.6.

4.4 PVM

4.4.1 Language Background

PVM (Parallel Virtual Machine System) is a software system that allows users to use a collection of heterogeneous computers as a coherent concurrent computational resource [32]. PVM provides a set of library routines for user applications to communicate with other entities in the PVM system.

Three primary library routines provided for message passing are `pvm_send`, `pvm_recv`, and `pvm_probe`. The `pvm_send` routine is asynchronous. It sends a labeled message to another PVM process. The execution of the sending process continues immediately without blocking. This message can then be received by `pvm_recv`. The `pvm_recv` routine allows specifying the sender and the label of a message to be received. When a matching message is not found, the receiving process blocks until such a message arrives. PVM also provides other routines to receive messages without blocking or with timeout. Like Ada, the content of messages cannot be used in deciding which message to receive. The `pvm_probe` routine verifies the arrival of a particular message without receiving it. This routine can check for a message with a specific label sent from a specific process.

4.4.2 Shortcomings

PVM has limitations similar to those discussed above. Message selection cannot be based on the message content. Moreover, PVM provides no straightforward way to examine messages without servicing them.

Chapter 5

New Mechanisms in SRR

As discussed in Chapters 3 and 4, existing mechanisms in SR and other languages do not facilitate straightforward solutions to a number of problems. Although solutions can be programmed using existing mechanisms, these solutions are cumbersome and inefficient. Three other shortcomings are also associated with the current mechanisms. First, it is difficult to examine invocations while preserving the arrival order of invocations in the invocation queue. To preserve the ordering often requires using additional operations and thus requires modification of the original servicing code. Second, to examine an operation that might be serviced by more than one process, the programmer has to control mutually exclusive access to the operation explicitly, for example, via semaphores. Last, cross-invocation or cross-operation selection also requires examination of invocations. Solving these kinds of problems using existing mechanisms often requires changes in the interface. For example, operations might need to be modified to use different parameterization; moreover, operations might be split into several different operations or combined to form one operation.

SRR provides four new statements — **rd**, **mark**, **take**, and **fetch** — that deal with invocations and that also overcome these difficulties. The **rd** (pronounced as "R–D") statement reads invocations in an invocation queue without disturbing their order or contents. The **mark** statement marks the invocation currently being examined via **rd**. A marked invocation can be serviced later by a **take** statement. The **fetch** statement is an abbreviation of the **mark** and **take** statements to service an invocation currently being examined via **rd**. These new statements have simple meanings; they are also expressive. Exclusive access to operations is handled implicitly by the mechanisms.

In this chapter, we first discuss the general form and semantics of the **rd** statement and then describe additional aspects of the **rd**. We then describe the other three statements — **mark**, **take**, and **fetch** — and their uses with other statements such as **return** and **forward**.

5.1 Rd Statement

5.1.1 General Form and Semantics

Syntactically, the **rd** statement is similar to the **in** statement:

rd *operation_command* [] *operation_command* [] ... **dr**

An operation command (see Chapter 2) specifies an operation to read, an optional synchronization clause, an optional scheduling clause, and a block of code. The last arm of an **rd** statement can be an else command, which has the form

[] **else** −> *block*

The **rd** statement is an iterator that reads all pending invocations, one per iteration, for operations appearing in the same **rd** in their arrival order. In general, the first iteration of **rd** reads the oldest invocation and each subsequent iteration reads the next invocation until all pending invocations have been accessed. Each iteration neither removes an invocation nor modifies its contents. When no invocations are pending or all pending invocations have been read by the **rd**, the process is delayed until one of the operations appearing in the **rd** is invoked. If the **rd** statement contains an else command, then the else command's block of code is executed before the process is delayed. The else command is typically used to terminate, via an **exit** statement, the **rd** statement.

As a simple example, the following **rd** statement reads an operation `f` whenever it is invoked.

```
rd f(x) ->
   write(x) # print out parameter
dr
```

Execution of this code does not terminate; it reads all pending invocations and then blocks until new invocations of `f` arrive. The body of the **rd** is executed only once for each invocation of `f`.

As another example, the **rd** statement in Figure 5.1 examines all invocations *currently* pending for `f` without blocking the process; this code is similar in effect to that in Figure 3.3. The iterative nature of **rd** allows the **rd** statement to examine, one at a time, each invocation of the operations that appear in the **rd** statement; thus, **rd** treats the collection of invocations as a group rather than individually, as **in** does. This behavior eliminates the potential of rearranging the order of invocations on the invocation queue that was present in Figure 3.1, which employed **in**. As discussed in Section 3.1.1, debugging and visualizing the execution of a concurrent program often require displaying pending invocations. The code in Figure 5.1 does exactly what these applications need. That solution is simple and elegant, unlike the code in Figures 3.1 and 3.3; it does not need cumbersome code to service and then re-invoke invocations. Each invocation in the invocation queue is accessed only *once* throughout the execution of the **rd**.

```
rd f(x) ->
   write(x) # print out parameter
[] else ->
   exit
dr
```

Figure 5.1: *Examining an operation.*

The semantics of an else command of an **rd** is different from that of an **in** statement. An **in** statement will terminate after the execution of its associated else command. On the contrary, an **rd** statement will not terminate after the execution of the else command unless it executes a statement, such as **exit**, that explicitly terminates the **rd**. After the execution of the else command of an **rd**, the process delays waiting for new invocations to arrive. For example,

```
rd f(x) ->
   write(x) # print out parameter
[] else ->
   write("wait for new invocations")
dr
```

This **rd** statement examines all pending invocations of `f`. When all pending invocations are printed, it executes the `write` statement within the else arm *once* and then blocks until new invocations of `f` arrive. The executing process does not busy wait. In particular, when a new invocation of `f` arrives, the execution of **rd** is continued and thus prints the parameter value of the new invocation. After that, the process executes the else command and blocks again. The else command's block of code is executed only when no invocations are pending for `f` or when some invocations have been examined and no more pending invocations are selectable for reading. The number of times that the else arm executes is nondeterministic for any execution of the **rd**.

To illustrate this nondeterministic behavior, consider that two invocations are pending for `f` initially, say `f(1)` and `f(2)`. When a process executes the **rd** statement, it outputs 1 and 2 before executing the else arm. It then executes the `write` statement in the else arm to display the wait message and blocks until new invocations arrive. Now, suppose another process invokes `f` twice, `f(3)` and `f(4)`, and no processes are servicing `f`. If these two invocations arrive in the queue before the examining process is scheduled to execute (although the process may wake up after one invocation arrives), the process will output 3 and 4 and then display the wait message. If the examining process resumes execution immediately after the invocation `f(3)` arrives, then it will examine invocation `f(3)` only. It will block after printing 3 and the wait message. When the invocation `f(4)` arrives later, it will wake up again to print 4 and will then execute the else arm. Tables 5.1 and 5.2 summarize the execution of these two cases after examining four invocations of `f`. The outputs of these two cases are different; the former iterates 6 times and displays the wait message twice whereas the latter iterates 7 times and displays the wait message three times.

Iteration	Output
1	`1`
2	`2`
3	`wait for new invocations`
4	`3`
5	`4`
6	`wait for new invocations`

Table 5.1: f(3) *and* f(4) *arrive before the process continues executing iteration 4 of the* **rd**.

Iteration	Output
1	`1`
2	`2`
3	`wait for new invocations`
4	`3`
5	`wait for new invocations`
6	`4`
7	`wait for new invocations`

Table 5.2: f(3) *arrives before the process continues executing iteration 4 of the* **rd** *and* f(4) *arrives later after iteration 5.*

Similar to the **in** statement, **rd** can also examine an array of operations, e.g.,

```
rd (i := 1 to N) f[i](x) ->
   ...
dr
```

The above **rd** reads an invocation from any one element of `f`. The scope of the quantifier variable `i` extends to the end of the operation arm of `f`. Therefore `i` can be used in the command block.

5.1.2 Synchronization Expressions

A boolean-valued synchronization expression can be used to control which invocation an **rd** reads next. For example, consider the following **rd** statements:

```
rd f(x) st x = 3 -> ... dr

rd f(x) st x > t -> ... dr

rd f(x) st x > 0 -> ...
[] g(y,z) st y = h(z) -> ...
dr
```

The first **rd** statement examines all invocations of `f` whose parameter `x` is equal to 3. The second **rd** statement examines all invocations of `f` whose parameter `x` is greater than `t`. The third **rd** statement reads all invocations of `f` whose parameter value is positive, as well as invocations of `g` whose parameters satisfy the condition `y = h(z)`, where `h` represents a user-defined or pre-defined operation.

As in an **in** statement, the synchronization expressions in an **rd** can reference invocation parameters. That ability makes it easy to solve many problems, as we shall see later in Chapter 6. However, it is more expensive to implement synchronization expressions in some cases than to implement a simple **rd** without synchronization expressions (see Chapter 7).

5.1.3 Scheduling Expressions

In general, invocations for operations appearing in the same **rd** statement are read in their arrival order. However, synchronization and scheduling expressions can alter the order in which invocations are read, just as they can within an **in** statement (see Chapter 2). A scheduling expression can be used to control which invocation — of those whose synchronization expression is true — an **rd** statement reads next.

For example, consider the problem of lottery scheduling presented in Section 3.2.1. The manager process picks a uid randomly from the uids of all requests that are pending. The manager process can examine all pending requests to obtain their uids as below

```
rd request(uid,...) by uid ->
   # save unique uid for later selection
     ...
[] else ->
   exit
dr
```

The scheduling expression forces this **rd** to read pending requests in non-decreasing order of the value of their invocation parameter `uid`. This sorted order helps the process to collect all distinct uids of requests. Scheduling expressions, like synchronization expressions, make it easy to solve many problems, but they do incur an implementation cost (see Chapter 7).

5.1.4 Exit and Next Statements

In other SR mechanisms, the **exit** statement is used to force early termination of the smallest enclosing iterative statement. The **next** statement is used to force return to the loop control of the smallest enclosing iterative statement. Since **rd** is also an iterative statement, **exit** and **next** statements may also appear within it. Execution of an **exit** statement terminates the enclosing **rd** statement, as seen in Figure 5.1. Within an operation arm of an **rd**, execution of a **next** statement forces the **rd** to read the next pending invocation. Any statements that follow the **next** within the same block of code are not executed. For example,

```
rd f(x) ->
   ...
   next
   write("statement not reached")
dr
```

The `write` statement that follows **next** is not reached on any execution of the **rd**. Because the **next** statement transfers control, it typically appears as the last statement within an **if** statement (as it is used in the next example).

The **next** statement can also be used within an else arm of an **rd**. Consider, for example, the problem of threshold order peeking presented in Section 3.1.3. Invocations of `f` are examined in several rounds; invocations whose parameter `x` is above a threshold are examined in each round. The threshold is lowered when all such invocations have been examined. One way using **rd** to display invocations of `f` in this way is shown below.

```
rd f(x) st x >= threshold ->
   write(x)                # print out parameter
[] else ->
   if threshold > 0 ->
      threshold -:= 10     # lower the threshold
      next
   fi
dr
```

Execution of a **next** statement within an else arm of an **rd** forces the **rd** to continue execution immediately. Note that the above **rd** statement examines all pending invocations whose parameter `x` is greater than or equal to `threshold`. In fact, all invocations pending for `f` are inspected by the **rd** but only those invocations whose synchronization expression of `f` is true are selected (see Chapter 7 for details). When all selected invocations are read and printed, the threshold is lowered (if its value is greater than zero) and the next iteration of the **rd** is then forced to continue. Execution of this iteration will select, for examination, invocations from among those previously *inspected* but not yet *read* invocations. To illustrate, consider examining the pending invocations of `f` shown in Figure 5.2. Suppose that before executing the above **rd** statement, the initial `threshold` is 30. Also

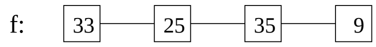

Figure 5.2: *Pending invocations of f.*

suppose that no invocations arrive during the execution of this **rd**. The first set of invocations printed out are invocations `f(33)` and `f(35)`, whose parameters are greater than 30. The else command's block of code is then executed to lower the `threshold` to 20. The execution of the **next** statement forces the **rd** to continue its next iteration rather than blocking the process. The next iteration examines the remaining two *unread* invocations and selects to print out invocation `f(25)`. After that, the else arm is executed again and the `threshold` is further lowered to 10. The **rd** continues its execution and determines that invocation `f(9)` is not above the threshold 10. Since no invocations are examined after executing the **next**, the **rd** blocks waiting for new invocations of `f` to arrive as if no **next** statement in the else arm had been executed. After the else command's block of code is executed once, it is executed the next time only when some invocations have been read and no more pending invocations are selectable to be examined. Section 9.1.3 discusses why we chose this semantics. Note in the above example that it may not be desirable to change the threshold value continuously without servicing an invocation. In addition, it is not desirable for the executing process to busy wait for invocations to arrive. This semantics was highly motivated by these requirements.

5.1.5 Return, Reply, and Forward Statements

The three statements, **return**, **reply**, and **forward**, have no special meaning within an **rd** statement. However, an **rd** statement can be nested within a **proc** or an **in** statement; in that case, these three statements have the effects described in Section 2.6 on the enclosing

proc or **in**. For example,

```
in f(x) ->
   # note that invocation being serviced by in is no longer
   # on the invocation queue, hence not available to rd
   rd f(y) st y = x ->  # any invocations with parameter y = x
      reply; exit
   [] else ->
   dr
   # further processing
     ...
ni
```

This **in** services an invocation of `f` and sends an early reply to the invoker if there is more than one pending invocation whose parameter value is the same as that of the servicing invocation. Similarly, **return** and **forward** can be used in this way.

5.2 Mark and Take Statements

The **mark** statement *marks* for possible later servicing an invocation currently being examined by an **rd** statement. Its form is simply

mark

The **take** statement services a previously marked invocation. Its general form is

take *op* (*formal_id_list*) [**returns** *result_id*] −> *block* **ekat**

The **take** statement will service the most recently marked invocation, by a given process, of the specified operation. The selected invocation is serviced by executing the associated block of code.

To illustrate the **mark** and **take** statements, the code below services invocations in LIFO order (see Section 3.2.3):

```
# service the most recently arrived invocation of f
rd f(x) ->
   mark
[] else ->
   exit
dr
take f(x) ->
   write(x) # service invocation
ekat
```

The **rd** statement examines and marks all invocations of `f`, one at an iteration. As a process can only mark at most one invocation of each operation, each subsequent execution of the **mark** statement replaces the previously marked invocation of `f` with the currently examined one for a given process. In other words, when the **rd** terminates, only the last examined invocation is marked and then taken by a given process. Note that **mark** must be used within an operation arm of an **rd** statement, but **take** can appear outside an **rd**, as it does above.

A **take** statement can also service a marked invocation of any one of a group of elements of an operation array by specifying a quantifier. For example, consider the following **take** statement:

```
# service a marked invocation of any one of request[1:4]
take (i := 1 to 4) request[i](x) ->
   write(i,x)
ekat
```

It services a marked invocation of any one element of `request`. This array of operations, `request`, can be considered as four individual operations, and at most one invocation of each of the operation elements can be marked per process. Thus, for example, suppose one invocation is pending for each element of the array operation `request`; there are a total of four pending invocations. The following **rd** statement

```
rd (i := 1 to 4) request[i](x) ->
   mark
[] else ->
   exit
dr
```

examines all invocations of `request` and marks at most one invocation for each element of `request` for a given process. In this case, all four invocations are marked. The above **take** statement selects the least recently marked invocation (among these four marked invocations) to service. Therefore, executing that **take** statement four times will service all these four marked invocations.

The **take** statement can have an optional else arm, which is typically used to output error messages when the **take** statement fails to service an invocation. For example, suppose two processes are each executing the following **rd** statement at about the same time and mark the same invocation.

```
rd f(x) ->
   mark; exit
dr
```

If both processes attempt to take the marked invocation by executing

```
take f(x) ->
   ...
[] else ->
   write("take failed: marked invocation has been serviced")
ekat
```

then the second **take** statement to execute will fail. The else arm's block of code will then be executed. If the failed **take** statement does not contain an else arm, it terminates immediately and the executing process continues after the **ekat**. Unlike the **in** statement, the executing process does not block when the **take** statement does not service an invocation.

5.3 Fetch Statement

The **fetch** statement is an abbreviation for use within an operation arm of an **rd** for marking and taking from the arm's operation. Its general form is

fetch *op* (*formal_id_list*) [**returns** *result_id*] −> *block* **hctef**

The **fetch** statement services the invocation of the specified operation currently being examined by an **rd**. For example,

```
rd f(x) ->
   fetch f(x) ->
      write(x) # print out parameter
   hctef
dr
```

is equivalent to the following **rd** statement:

```
rd f(x) ->
   mark
   take f(x) ->
      write(x) # print out parameter
   ekat
dr
```

The **fetch** statement can also have an optional else arm; it has the same meaning as an else within a **take** statement. Thus, if the **fetch** statement does not succeed in servicing the currently examined invocation, the else arm's block of code is executed before the **fetch** statement terminates.

5.4 Return, Reply, and Forward Statements

Three other statements — **return**, **reply**, and **forward** — that affect invocation servicing within an **in** statement (see Chapter 2) can also be used, with the same meanings, within the **take** and **fetch** statements. For example,

```
take f(x) ->
   forward g(x)
ekat
```

This **take** will service a marked invocation of `f` and then forward it to `g` without sending a reply to the invoker.

Chapter 6

Examples

As seen in Chapter 3, current mechanisms in SR for examining invocations result in cumbersome and inefficient code. The four new statements provided by SRR — **rd**, **mark**, **take**, and **fetch** — are designed to supplement the currently existing mechanisms in SR. These new mechanisms provide additional support for handling invocations. Chapter 5 presented some small examples of using these new mechanisms. This chapter presents several large and realistic examples using these new mechanisms. It also discusses how the input statement and the '?' operator can be simulated using these mechanisms. This chapter also briefly discusses the performance of the new mechanisms. Chapter 8 will present a complete discussion about the performance issue.

6.1 Peeking at Pending Invocations

As seen in Figures 3.1 and 3.3, the code for examining an operation is fairly costly; invocations are serviced and then re-invoked. Below is one solution using **rd** for peeking at pending invocations of an operation `f`.

```
# examining an operation while preserving arrival order of invocations
rd f(x) ->
   write(x) # print out parameter
[] else ->
   exit
dr
```

Compared to the solutions using **in** statements in Figures 3.1 and 3.3, this code is simpler and, most importantly, is much more efficient. Invocations of `f` no longer need to be removed from and appended onto the invocation queue to examine them. The **rd** statement reads invocations from an invocation queue without disturbing their order or contents. This code can also handle the situation when multiple processes (servicing and examining processes) are accessing the invocation queue at the same time (see Section 3.1.2). For example, if other processes are servicing invocations of `f` at about the same time as the examining process, some invocation might be removed. The removal of an invocation during the execution the **rd** will not affect its correctness because the **rd** does not require to use any variable to store the number of pending invocations at a particular state; instead, the **rd** implicitly detects when all pending invocations have been examined. Unlike the code in Figure 3.4, no auxiliary or temporary operations are needed.

6.2 Cross-invocation and Cross-operation Scheduling

Section 3.2 presented several common scheduling problems and their solutions. SR's current single-invocation based selection mechanisms require most of those solutions to use a "pre-**in** loop" to select an invocation to service first and then save its parameter values for later servicing (usually by the following **in** statement). As discussed in Section 6.1, using **rd** to examine pending invocations preserves the arrival order of invocations and also handles the multiple servicing processes problem. Most importantly, the code is simple and efficient (see Chapter 7). Below we present SRR solutions to the scheduling problems described earlier in Chapter 3.

6.2.1 Lottery Scheduling

Figure 6.1 shows an SRR solution to the lottery scheduling problem from Section 3.2.1. The solution firsts examines all invocations of `requests` and saves all distinct uids. The manager process then picks a uid randomly from those saved uids and assigns it to `theuid`. The **in** statement then uses `theuid` to service a request.

```
rd request(uid,...) by uid ->
   # store unique uid for later selection
     ...
[] else -> exit
dr
# set theuid to the randomly chosen uid
theuid := random_uid()
in request(uid,...) st uid = theuid ->
   # service this request for some quantum
     ...
   if "more work left before this request can finish" ->
      # needs further service, so put it back on queue
        forward request(uid,...)
   fi
ni
```

Figure 6.1: *SRR lottery scheduler using* **rd**.

6.2.2 Extended Job Scheduling

Figure 6.2 shows an SRR solution to the extended job scheduling problem from Section 3.2. The **rd** statement reads invocations of `batch` and `interactive`. Specifically, it examines all pending invocations of `batch` until there are no more or it finds one from the superuser; it reads only the first invocation of `interactive` provided it has not already seen an invocation of `batch` from the superuser. The **rd** terminates via the **exit** statement in its else arm. The **exit** is executed if an invocation was marked; otherwise, the process blocks waiting for an invocation of `batch` or `interactive` to arrive before it executes an arm of the **rd**. This code avoids the problems of the solutions presented in Section 3.2 to this same example. Specifically, this code would not modify the invocation queue when selecting an invocation to service via **rd**. Second, it does not require changes in the interface; processes service invocations of `batch` and `interactive` as they originally did. Last, it does not require code to be repeated for servicing superuser batch jobs and normal batch jobs. This solution is, however, lower-level in the sense that it explicitly marks and takes invocations and uses boolean flags to control doing so.

```
var gotsuper := false, gotbatch := false, gotinteractive := false
# find the invocation to service
rd batch(uid,...) st not gotsuper ->
   if uid = 0 # any superuser requests?
      mark batch
      gotsuper := true
   [] uid != 0 & not gotbatch -> # mark first one
      mark batch
      gotbatch := true
   fi
[] interactive(uid,...) st not gotinteractive & not gotsuper ->
   mark interactive
   gotinteractive := true
[] else ->
   if gotsuper | gotbatch | gotinteractive -> exit fi
dr
# now service
if gotsuper | (gotbatch & not gotinteractive) ->
   take batch(uid,...) -> ... etak
[] else -> # i.e., gotinteractive
   take interactive(uid,...) -> ... etak
fi
```

Figure 6.2: *SRR extended job scheduler using* **rd**.

Other, higher-level solutions are also possible. For example, an **rd** could be used, in the style of a "pre-**in** loop", to determine whether any superuser invocations of `batch` are present. If so, the invocation could be serviced using **fetch**; if not, the original **in** statement could be used. This approach can be programmed as

```
rd batch(uid,...) st uid = 0 ->
   # service superuser batch invocation
   fetch batch(uid,...) ->
      ...
   hctef
[] else ->
   exit
dr
in interactive(uid,...) -> ...
[] batch(uid,...) st ?interactive = 0 -> ...
ni
```

Note that all superuser batch invocations are serviced by the **fetch** within the **rd**. The **rd** terminates when all such invocations have been serviced or no such invocations are pending. Other pending invocations are serviced by the **in**, as it originally did in Figure 2.2. Here, we use the **rd** together with **fetch** to simulate an input statement to service an invocation (see Section 6.3 below). This code is therefore equivalent to the code in Figure 3.9. However, this code is more efficient than the code in Figure 3.9 because the **rd** scans the invocation queue only once, whereas the "pre-**in** loop" in Figure 3.9 scans the invocation queue more than once if there are some superuser batch jobs pending (see Section 8.2.4).

6.2.3 Other Scheduling Scenarios

In this subsection, we revisit several examples discussed in Section 3.2.3.

6.2.3.1 Two-dimensional Scheduling

The following gives a solution using **rd** to the two-dimensional scheduling problem.

```
# service all pending invocations in matrix scheduling order
rd f(x,y) by x ->
   # read those invocations with minimum x value
   rd f(x2,y2) st x = x2 by y2 ->
      # service the least invocation currently examined
      fetch f(x,y) ->
         write(x,y) # print out parameters
      hctef
      exit
   dr
dr
```

This code is similar to that in Figure 3.10, in which nested **in** statements are used. The outer **rd** first finds the smallest `x` parameter value and then select the invocation whose `y` parameter value is the least among all pending invocations that have that `x` parameter value. The least invocation is then serviced by the **fetch** within the inner **rd**. Note that **exit** is used to terminate the inner **rd** after one iteration, i.e., the least invocation is examined. In contrast to the solution using **in** statements, this solution guarantees that a new invocation that has not been examined in the previous iteration will be considered in the next iteration of the outer **rd** as it might have the smallest `x`. In addition, this solution is more efficient because invocations are not serviced and re-invoked after being examined.

6.2.3.2 Median Scheduling

Figure 6.3 gives an SRR solution to the median arrival scheduling. This **rd** reads pending

```
do ?f > 0 ->
   median := ?f/2+1
   count := 0
   # read invocations of f in the order of x
   rd f(x) ->
      if ++count = median ->
         fetch f(x) ->
            ...
         hctef
         exit
      fi
   dr
od
```

Figure 6.3: *SRR solution to the median-arrival scheduling.*

invocations of `f`, one at a time, and then increments `count`. The iterative nature of the **rd** allows simulating an **in** with a synchronization expression. Specifically, the **rd** and the **fetch** are used to simulate the **in** statement in Figure 3.12. This solution increments `count` on each iteration of **rd** whose value represents the sequence number of the invocation. It then uses an **if** to determine whether or not the examining invocation is the median. This has exactly the same effect as the synchronization expression of the **in** in Figure 3.12, which invokes `seqno` to assign sequence numbers to invocations. In both cases, the **rd** and the **in** terminate when the median invocation is found.

Figure 6.4 shows an SRR solution to the median-value scheduling. This code uses the same approach as the solution in Figure 3.13. It examines pending invocations in sorted order (via the **by** of the **rd**) and then services the median invocation using the median-arrival scheduling approach (Figure 3.12). However, this code avoids the problem of the solution in Figure 3.13. In particular, the invocation queue is not modified and

```
do ?f > 0 ->
   median := ?f/2+1
   count := 0
   # read invocations of f in the order of x
   rd f(x) by x ->
      if ++count = median ->
         # service the invocation of f for which x is the median
         fetch f(x) ->
            ...
         hctef
         exit
      fi
   dr
od
```

Figure 6.4: *SRR solution to the median-value scheduling.*

invocations of `f` are still pending in their arrival order.

6.3 Simulating Input Statements

As discussed briefly in Section 6.2.3, an **rd** and a **fetch** can be used to simulate an **in** statement. As a simple example,

```
rd f(x) ->
   fetch f(x) ->
      # service invocation
        ...
      exit
   hctef
dr
```

The above **rd** reads the first pending invocation of `f` and then the **fetch** services the *currently* examining invocation. Thus, the executing process services the first pending invocation. If only one process services invocations of `f`, the above code is equivalent to

```
in f(x) ->
   # service invocation
     ...
ni
```

However, if more than one process services invocations of `f`, then the above **rd** and **in** statements are not equivalent because a process executing the **rd** statement might not succeed in servicing (fetching) the invocation that is currently being examined. Another process could grab that invocation while it is being examined by that process. In contrast, a process executing the **in** statement never fails to service the invocation once that invocation is selected.

An **in** statement with a synchronization expression can be simulated by

```
rd f(x) st sync_expr ->
   fetch f(x) ->
      # service invocation
        ...
      exit
   hctef
dr
```

The above **rd** examines a pending invocation whose synchronization expression is true. It terminates when the invocation is selected and serviced.

As another example, consider implementing the shortest-job-next request allocator using **rd** and **fetch**. The code in Figure 6.5 simulates the code shown in Figure 2.1 that uses **in** statement. The **rd** can replace the **do** loop because of its iterative nature. On

```
# simulate the do and in
rd request(size) st free by size ->
   fetch request(size) ->
      free := false
   hctef
[] release() ->
   fetch release() ->
      free := true
   hctef
dr
```

Figure 6.5: *SRR shortest-job-next request allocator using rd.*

each iteration of the **rd**, the process will service the invocation of `request` with the smallest `size` — but only if `free` is true — or an invocation of `release`.

An **rd** can also be used to simulate the '?' operator that returns the number of invocations currently pending for an operation. For example,

```
count := 0
rd f(x) ->
   count++
[] else ->
   exit
dr
```

The above **rd** counts the number of pending invocations of `f`, i.e., `?f`. However, this code is quite inefficient as each pending invocation has to be accessed once. On the other hand, the '?' operator needs not access any invocations; instead, it relies on the internal counter maintained by the RTS to store the number of pending invocations for each operation. Thus, it can be (and is [4, 3]) implemented much more efficiently (see Chapter 7).

6.4 Multi-way Rendezvous

A rendezvous involves two processes: an invoking process and a servicing process that handles the invocation. The invoking process invokes an operation and then waits for the servicing process to service the invocation. Synchronization and communication are required before either processes continues its execution. Charlesworth proposes the *multi-way rendezvous* [13], which is a generalization of rendezvous in which more than two processes participate. He proposes a language-independent construct — the *compact* — for expressing multi-way rendezvous and describes extensions to existing languages for supporting the compact. His proposed extension to the original version of SR [1] permits multiple operations to be specified in a single arm of an input statement. For example,

in *op1(...), op2(...)* $->$ *block* **ni**

The process executing this input statement is delayed until *op1* and *op2* each has a pending invocation.

Other than extending the language to handle multi-way rendezvous (with multiple-operation input statements), existing mechanisms in the current version of SR [2, 4] can also be used [14]. This approach employs SR's concurrent invocation statement (**co**) and rendezvous mechanism to achieve the desired functionality.

The multiple-operation input statement described above is attractive since it is the natural generalization of two-way rendezvous. Programmers should therefore have little conceptual difficulty in understanding it or in extending programs to employ it. However, extending the current version of SR with such an input statement would be difficult, as discussed in [14].

```
var f_invoked := false, g_invoked := false
rd f(x) st not f_invoked ->
   mark
   f_invoked := true
   if g_invoked -> exit fi
[] g(y,z) st not g_invoked ->
   mark
   g_invoked := true
   if f_invoked -> exit fi
dr
take f(x) ->
   take g(y,z) ->
      # servicing code
        ...
   ekat
ekat
```

Figure 6.6: *SRR multi-way rendezvous using* **rd**.

Below, we show how the multi-way rendezvous can be expressed using the new mechanisms in SRR. The **rd** allows an invocation to be read without actually removing the invocation from the operation queue; the **mark** allow invocations to be marked and later serviced (by **take**). Figure 6.6 shows how these mechanisms implement the following SR-like multiple-operation input statement.

```
in f(x), g(y,z) ->
   ...
ni
```

The **rd** in Figure 6.6 examines only the first invocation of `f` and the first invocation of `g`. The process delays until `f` and `g` each has a pending invocation. A flag is set for each operation when an invocation of the operation is examined and marked; `f_invoked` and `g_invoked` are used for `f` and `g` respectively. When `f` and `g` each has a pending invocation and has been examined, the **rd** terminates immediately via the **exit**. The process then services the marked invocations (one for `f` and one for `g`) simultaneously. One disadvantage of this code is that it is rather cumbersome especially when the number of arms is large.

```
var f_invoked := false, g_invoked := false
do true ->
   in f(x) st not f_invoked ->
      forward aux_f(x)
      f_invoked := true
      if g_invoked -> exit fi
   [] g(y,z) st not g_invoked ->
      forward aux_g(y,z)
      g_invoked := true
      if f_invoked -> exit fi
   ni
od
in aux_f(x) ->
   in aux_g(y,z) ->
      # servicing code
        ...
   ni
ni
```

Figure 6.7: *SR multi-way rendezvous using* **in**.

A style similar to that of Figure 6.6 could be used for **in** statement to express the multi-way rendezvous as shown in Figure 6.7. It requires additional operations, one for each operation arm, to which the servicing process forwards the invocation examined via **in**. This process can actually service the invocations only when `f` and `g` each has a pending invocation. Of course, it requires changes in the interface and the process actually services the invocations from the new operations.

The code in Figure 6.6 assumes that the multi-way rendezvous involves multiple invoking processes but only one servicing process. Otherwise, if some other process also services the invocations, then the **take** statements in Figure 6.6 may fail to service a marked invocation. For example, another servicing process could grab the first invocation of f just before the current servicing process executes the **take** statement. Similarly, if several processes are executing the code in Figure 6.7, they might result in deadlock. For example, suppose two processes are executing the code in Figure 6.7. One process grabs one invocation of f and waits for an invocation of g to arrive. The other process grabs one invocation of g and waits for an invocation of f to arrive. Unfortunately, no new invocations might arrive until these two invocations are serviced. These two processes then wait forever.

```
sem mutex := 1
do true ->
   P(mutex)
   if ?f > 0 and ?g > 0 ->
      in f(x) ->
         in g(x) ->
            # servicing code
              ...
         ni
      ni
   fi
   V(mutex)
od
```

Figure 6.8: *SR multi-way rendezvous with starvation.*

Figure 6.8 shows one way to use **in** to simulate the multi-way rendezvous involving several servicing processes. Similar to the multiple servicing processes problem in Section 3.1.2, this code also employs a semaphore operation, mutex, to implement mutual exclusion. Processes have to obtain permission to service an operation. Therefore, other servicing code also has to be enclosed by the P and V operations on mutex as in the above code. However, this solution does not guarantee fairness and starvation could occur. As a simple scenario, suppose invocations of f and g always arrive after the process executes V in the code in Figure 6.8. Before that process obtains the mutex again in the next iteration, one or more processes service those invocations, preventing that process from ever finding pending invocations to service. However, the starvation problem could be solved, but at greater complexity and expense. Another drawback of this code is that the process executing this code could busy wait. Note also that the above code checks whether `f` and `g` each has some pending invocations using the '?' operator before servicing. The '?' operator can be used to determine whether an operation has some pending invocations. However, it cannot be used to determine whether an operation has some pending invocations whose parameters satisfy some constraints (see Section 6.3), which is another peeking problem.

```
sem mutex := 1
var f_invoked := false, g_invoked := false
rd f(x) st not f_invoked ->
   f_invoked := true
[] g(y,z) st not g_invoked ->
   g_invoked := true
[] else ->
   if f_invoked and g_invoked ->
      P(mutex)
      if ?f > 0 and ?g > 0 ->
         in f(x) ->
            in g(x) ->
               # servicing code
                 ...
            ni
         ni
         V(mutex); exit
      [] else ->
         f_invoked := ?f > 0
         g_invoked := ?g > 0
         V(mutex)
      fi
   fi
od
```

Figure 6.9: *SRR multi-way rendezvous with multiple servicing processes.*

Figure 6.9 shows another way to simulate the multi-way rendezvous involving several servicing processes. This code services the invocations within the else command of the **rd**, using an approach similar to that presented in Figure 6.8. The executing process obtains `mutex` only when the process has examined one invocation from `f` and one from `g`. Before servicing the invocations, the process checks whether or not `f` and `g` each still has a pending invocation. If not, the flags are set appropriately to indicate whether an operation needs to be re-examined. In contrast to the code in Figure 6.8, the process executing this code does not busy wait. If the process fails to service the invocations after examining them, it then blocks waiting for a new invocation to arrive. Nevertheless, as illustrated by the applications of the multi-way rendezvous in [13], it is not common to have multiple servicing processes.

6.5 Elevator Controller Simulation

Visualizing or animating the execution of a concurrent program often requires displaying pending invocations. This section presents a realistic example of an elevator controller simulation. As discussed in Section 3.1.1, each person is represented as a process and the elevator controller is represented as a separate process. When a person process wants to take the elevator to another floor, it invokes operation `get_on` in the elevator controller; the controller services the invocation when it schedules the elevator to pick up this passenger. To visualize the execution of the simulation, the elevator examines the operation to determine which people are waiting on which floors.

6.5.1 SR Solution

Figure 6.10 shows the pseudocode for the elevator controller process using SR's existing mechanisms. The execution of the simulation is visualized by showing which people are waiting on which floors and which people are on the elevator. People waiting to get on the elevator are shown in their arrival order. The controller first examines all passengers that are waiting to get on the elevator and determines the direction that the elevator should move if it is not moving. It then discharges passengers on the current floor and picks up passengers that are waiting on the current floor to go in the same direction as the elevator. Note that after the controller examines invocations (which represent the passengers waiting to get on the elevator), they are forwarded from `get_on` to another operation `wait_geton` in order to preserve their arrival order. Therefore, the controller services invocations from `wait_geton` when it schedules the elevator to pick up a passenger. When the elevator finishes picking up passengers on the current floor, the controller determines the next floor

```
process elevator_controller
   do true ->
      # examine the newly arrived passengers
      do ?get_on > 0 or "elevator is not moving" ->
         in get_on(src,dest,...) ->
            # update new request
            # determine the moving direction if not moving
              ...
            forward wait_geton(src,dest,...)
         ni
      od
      if "new request" -> call redraw(curFloor) fi
      # discharge passengers on current floor
      do true ->
         in get_off(src,dest,...) st curFloor = dest ->
              ...
         [] else -> exit
         ni
      od
      do true ->
         in wait_geton(src,dest,...) st "elevator not full" and
              "passengers waiting on current floor and they want
               to go in same direction" ->
            # pick up this passenger
              ...
            forward get_off(src,dest,...)
         [] else ->
            # display new elevator status if some passengers got on or
            # got off the elevator
            if "status changed" -> call redraw(curFloor) fi
            # update status of elevator:
            if ?get_off = 0 ->   # elevator is empty
               # decide whether to move to next floor or
               # reverse the direction of elevator or
               # stay on current floor
                 ...
            [] else ->
               # keep moving in same direction and update curFloor
               # then display new elevator position
                 ...
            fi
            # exit the loop to examine newly arrived passengers
            exit
         ni
      od
   od
end
```

Figure 6.10: *SR pseudocode for elevator controller simulation.*

to which the elevator is to move. The current execution state of the simulation is displayed by calling the procedure `redraw`. Figure 6.11 shows the part of the procedure `redraw` that

```
procedure redraw(curFloor: int)
   # determine which people are waiting on which floor
   fa i := 1 to ?wait_geton ->
      in wait_geton(src,dest,...) ->
         ...
         forward wait_geton(src,dest,...)
      ni
   af
   # display
     ...
end
```

Figure 6.11: *Procedure* redraw *to display the elevator controller simulation.*

determines which people are waiting on which floor. The arrival order of invocations is preserved in this code because, unlike the case described for the code in Figure 3.1, the controller is the only process that invokes and services `wait_geton`.

6.5.2 SRR Solutions

This section presents three different ways to modify the SR solution to the elevator controller simulation problem presented above. The first SRR solution employs the code in Figure 6.10 for the elevator controller process but it uses another way to determine which people are waiting on which floor. Specifically, procedure `redraw` uses the following **rd**.

```
rd wait_geton(src,dest,...) ->
   ...
[] else -> exit
dr
```

This code is simpler and more concise than the code in Figure 6.11.

The code in Figure 6.10 uses an additional operation `wait_geton` in order to preserve the arrival order of invocations. The new mechanisms in SRR permit other ways to program the elevator controller without using an additional operation. Figure 6.12 shows one way using **rd** to examine all invocations of `get_on`. The rest of the code is same as in the SR solution except that the controller now services `get_on` instead of `wait_geton`.

```
process elevator_controller
   do true ->
      # examine the brand new arrived passengers
      rd get_on(src,dest,...) st "new request" ->
         # update new request
         # determine the moving direction if not moving
           ...
      [] else ->
         if "elevator is moving" -> exit fi
      dr
      /* two do loops same as in Figure 6.7 except that */
      /* the input statement services get_on instead of wait_geton */
         ...
   od
end
```

Figure 6.12: *Elevator controller simulation using* **rd** *to examine* get_on.

The third SRR solution is presented in Figure 6.13, which shows the pseudocode for the elevator controller process using **rd** and **fetch** only. Similar to Figure 6.12, no additional operation is used in this code. The else arm of the **rd** is executed when the elevator finishes loading and unloading passengers on the current floor. The controller first displays the current execution state of the simulation. It then executes the **do** loop to schedule the elevator to move to the next floor on which some passengers will get on or get off from the elevator. The controller will block on the **rd** within the **do** loop if the elevator is empty and no passengers are waiting to get on the elevator. The controller will continue its execution when new invocations arrive.

```
process elevator_controller
   rd get_off(src,dest,...) st curFloor = dest ->
      # discharge passenger on current floor
      fetch get_off(src,dest,...) ->
         ...
      hctef
   [] get_on(src,dest,...) st "elevator not full" and
        "passengers waiting on current floor and they want
         to go in same direction" ->
      fetch get_on(src,dest,...) ->
         # pick up this passenger
           ...
         forward get_off(src,dest,...)
      hctef
   [] else ->
      # display new elevator status if some passengers got on or
      # got off the elevator
      if "status changed" -> call redraw(curFloor) fi
      # keep moving the elevator or stay on current floor
      # until someone waiting to get on the elevator or
      # to get off from the elevator on current floor
      do true ->
         # update status of elevator:
         if ?get_off = 0 ->  # elevator is empty
            # decide whether to move to next floor or
            # reverse the direction of elevator or stay on current floor
              ...
         [] else ->
            # keep moving in same direction and update curFloor
            # then display new elevator position
              ...
         fi
         # examine new requests
         rd get_on(src,dest,...) st "new requests" ->
            # update new requests
            # decide the elevator direction if not moving
              ...
         [] else ->
            if "new request" -> call redraw(curFloor) fi
            if "elevator is moving" -> exit fi
         dr
         # exit the loop to continue the rd
         if "someone waiting to get on elevator on current floor" or
            "someone in elevator gets off current floor" ->
            exit
         fi
      od
      next  # force the controller to examine; not to block
   dr
end
```

Figure 6.13: *SRR pseudocode for elevator controller simulation.*

Chapter 7

SRR Implementation

SRR is implemented on top of SR (version 2.3), which consists of three major components: the compiler, linker, and run-time support (RTS) [3, 28]. The SR compiler generates C code, which is in turn passed through the C compiler to produce machine code (MC). The SR linker then produces an executable image for a specific SR program by combining its MC with the program-independent RTS. The RTS provides the environment in which the MC executes. In particular, it provides primitives that are used by the code generated by the SR compiler to access services provided by the RTS. The implementation of our new mechanisms in SRR involves extensions only to the compiler and the RTS.

This chapter first presents a brief overview of the SR implementation focusing on the mechanisms relevant to operations serviced by input statements. It shows generated code for simple forms of input statements in order to give readers a feel for the generated code. See [3] and [28] for further implementation details. This chapter then describes the SRR implementation. It begins with a description of the general approach and a discussion of the implementation difficulties due to the complicated semantics of **rd**. It describes the refinement of the SRR implementation. This chapter then describes the key SRR primitives provided by the RTS and then presents several examples and their generated codes to illustrate how these primitives for **rd** are used by the MC. Finally, this chapter gives some statistics about our effort spent on this work.

7.1 Overview of SR Implementation

7.1.1 Invocation Statements

The RTS maintains a queue of its pending invocations for an operation that is serviced by **in** statements. To invoke an operation (either by a **call** or **send** statement), the compiler generates codes for allocating an invocation block, filling in parameter values, and passing the invocation block to the RTS. When receiving the invocation block, the RTS appends the invocation onto the end of the queue associated with the operation and awakens the servicing process if it is waiting for such an invocation. For a **call** invocation, the RTS blocks the calling process until its invoked operation has been serviced. When the calling process is later awakened, it retrieves any results from the invocation block and asks the RTS to free the invocation block. For a **send** invocation, the RTS allows the invoking process to continue and will free the invocation block when the invoked operation completes.

7.1.2 Classes of Operations

We focus only on operations that are serviced by input statements. As described in Chapter 2, an operation can be serviced by multiple processes, which might compete for invocations of a shared operation. To identify and control conflicts between such competing processes, operations are grouped into *classes* based on their appearance in input statements. Operations appearing in the same input statement are grouped into a class. The transitive closure of this relation "serviced by the same input statement" forms the classes in a resource [3]. To illustrate, consider the input statements in a resource shown in Figure 7.1. Four operations `a`, `b`, `c`, and `d` are serviced by these input statements. Two classes are thus formed: {`a,b,c`} and {`d`}. As `a` and `b` are serviced by the second input statement and `b` and `c` are serviced by the third input statement, all these three operations are grouped into the same class. On the other hand, `d` is the only operation in the second class because the input statement that services `d` does not service other

```
in a(x) -> ... ni

in a(x) -> ...
[] b(x,y) -> ...
ni

in b(x,y) -> ...
[] c() -> ...
ni

in d() -> ...
ni
```

Figure 7.1: *Example of classes.*

operations.

The RTS maintains a *class structure* for each class [3, 28]. A class structure contains a flag indicating whether or not some process currently has access to the class, old and new invocation lists, and a list of processes that are waiting to access the class. The invocation and process lists are used to ensure that processes obtain access to invocations in FCFS order. The old invocation list contains pending invocations of operations in the class. The new invocation list contains invocations that arrive while a process is accessing the old invocation list. The flag is used to implement the exclusive access to the pending invocations in a given class (i.e., the old invocation list[1]).

The way that the RTS maintains the invocation lists is illustrated as follows. When a new invocation arrives, the RTS first determines the class of this invocation and then checks whether or not some process currently has access to that class. If no process currently has access to the class, the invocation is appended to the old invocation list. The RTS then awakens the processes that are waiting for a new invocation to arrive (i.e., those that have searched the old invocation list but did not find a selectable invocation), if any. On the other hand, if some process is currently searching the old invocation list, the invocation is appended to the new invocation list. When that process completes its

[1]The term "invocation queue" used throughout this thesis refers to the old invocation list described in this section.

search, the new invocation list is moved to the end of the old invocation list. The RTS awakens a process that is waiting to access the pending invocations in the class, if any. It gives preference to the processes that are waiting for new invocations to arrive over those that are waiting to search the old invocation list.

7.1.3 Input Statements

The RTS provides several primitives that the MC uses for input statements. The list below summarizes the essential primitives to implement input statements.[2]

iaccess(*class, else_present***)** — Acquires exclusive access to *class*. Exclusive access is released when the process executes **rm_iop** (see below) or when it blocks trying to get an invocation. The process blocks if another process already has access to *class*. Parameter *else_present* is true if the **in** statement has an else clause and is false otherwise. Section 7.5 describes how this primitive is also used for implementing **rd** statements.

get_anyinv(*class***)** — Returns a pointer to the next uninspected invocation block in *class*. If there is no such invocation, the process blocks waiting for new invocations to arrive. If the **in** statement has an else clause, it returns a null pointer instead.

chk_myinv(*op_cap***)** — Returns a pointer to the next invocation of the operation named by *op_cap*. The RTS returns a null pointer when no invocation is found.

rm_iop(*class, invocation***)** — Removes an invocation block pointed at by *invocation* from *class* and releases access to the class.

finished_input(*invocation***)** — Informs the RTS that execution of the command body in an **in** statement has been finished. The invocation block pointed at by *invocation* is freed if it is no longer needed. If that invocation was called, the RTS passes the invocation back to the caller and awakens that process.

[2]For simplicity, we have omitted the **sr_**prefix that is common to the names of all RTS functions, and we have also omitted the non-essential arguments.

The discussion illustrates how input statements are implemented using appropriate combinations of the above primitives. (The primitives called by the MC are shown inside the parentheses.)

First, consider a simple input statement that does not use synchronization and scheduling expressions, which services an invocation pending for `f`:

```
in f(x) -> ... ni
```

Figure 7.2 presents the pseudocode of the generated code for this input statement.[3]

```
iaccess(c,0)
inv := get_anyinv(c)
rm_iop(c,inv)
# service the invocation
  ...
finished_input(inv)
```

Figure 7.2: *Generated code for a simple input statement.*

Suppose `f` is the only operation in class `c`. The MC first acquires exclusive access to the class (`iaccess`). It then asks the RTS for the first invocation in the invocation queue (`get_anyinv`). If there is such an invocation, the RTS removes it from the queue and returns it to the executing process (`rm_iop`).[4] If there is no such invocation, the RTS blocks the process; the RTS will awaken this process when such an invocation arrives and return the new invocation to the process as described above. The MC then continues execution of the input statement by executing the command body associated with the operation. When the MC completes the execution of the command body, it passes the invocation block back to the RTS (`finished_input`). The RTS replies to the invoker if the invocation was via **call** or the RTS frees the invocation block if it was via **send**, as described in Section 7.1.1.

[3]For brevity, in this chapter, generated code is presented as pseudocode in SR style, even though the compiler generates C code in the real implementation.

[4]The RTS also provides a useful abbreviation for `get_anyinv` and `rm_iop`, a `receive` primitive, that removes the first pending invocation in the class. For illustrative purposes, we do not use the `receive` primitive in Figure 7.2; instead, we show the expanded generated code.

The implementation of an input statement with a synchronization expression is similar to that of a simple input statement described above. Consider the following input statement:

```
in f(x) st x > 0 -> ... ni
```

This statement services an invocation whose parameter is positive. The generated code for this statement is shown in Figure 7.3. First, the MC acquires access to the class

```
iaccess(c,0)
do true ->
   inv := get_anyinv(c)
   if inv.x > 0 ->  # evaluate synchronization expression
      rm_iop(c,inv)
      # service the invocation
        ...
      finished_input(inv)
      exit
   fi
od
```

Figure 7.3: *Generated code for an input statement with a synchronization expression.*

(`iaccess`). The MC then searches the invocation queue for the first selectable invocation. It asks the RTS for pending invocations one at a time (`get_anyinv`). For each such invocation, the MC evaluates the synchronization expression using the parameter values in the invocation block. Note that we use the notation `inv.x` to represent the invocation parameter `x` in the invocation block `inv`. Specifically, in SR pseudocode, the invocation block can be considered a record and each invocation parameter corresponds to a field in the record. If the synchronization expression is true, the MC informs the RTS to remove the invocation (`rm_iop`) and then continues execution the input statement. If the synchronization expression is false, the MC repeats the above steps to obtain the next invocation.

For an input statement with a scheduling expression, the MC uses the same steps as described above to obtain the first selectable invocation. It then evaluates the scheduling expression using the parameter values in that invocation block. The MC searches the remaining invocation queue for the invocation with the smallest scheduling expression. It does so by obtaining the remaining invocations one at a time (`chk_myinv`) and then choosing the one whose scheduling expression is the smallest. When the MC completes its search, it then acquires that invocation and continues execution as described in the previous cases.

7.2 SRR General Implementation Approach

Classes are used in the implementation of **rd** statements. Operations that appear in an **rd** statement are grouped into a class in the same way as described in Section 7.1.2. A class can be accessed by both **rd** statements and **in** statements. Processes that examine invocations via **rd** must obtain exclusive access to the class before examination (although they do not remove or modify any invocation from the invocation queue). Otherwise, race conditions with another process servicing invocations in the queue are possible. For example, suppose a servicing process has selected an invocation to service. Before the servicing process actually removes that invocation from the invocation queue, an examining process might try to examine that invocation at the same time (as it considered that the invocation is pending).

Recall from Section 5.1 the semantics of the **rd** statement. The **rd** statement allows other invocation servicing statements, such as **in** and **fetch**, to be executed within its command body. In addition, another process is allowed to service an invocation that is currently being examined by a process via **rd**. This semantics therefore requires a finer grained control on access to the class: the process acquires access to the invocation queue at the beginning of each iteration of the **rd** (i.e., before choosing an invocation for examination) and releases access before executing the command body. (We explored other alternatives of the semantics and locking mechanism used for **rd** statements, which will be discussed in Section 9.1.)

This semantics greatly complicates the SRR implementation. One major concern is that the invocation queue could be modified by other processes during execution of an **rd**.

Specifically, new invocations might be added to and pending invocations might be removed from the invocation queue while an **rd** is examining an invocation. The RTS must be able to handle such invocation insertion and removal without affecting the execution of the **rd**. Another difficulty in implementing **rd** is that synchronization and scheduling expressions can be used to alter the order in which an **rd** examines an operation. An **rd** cannot simply examine pending invocations one by one following the invocation queue. Furthermore, an operation can be examined by several processes simultaneously. Thus, the RTS must be able to recognize which invocations that have been examined for each such process.

The remainder of this section presents some alternatives in handling these difficulties and discusses the reasons why we chose our current implementation. We want the SRR implementation not only to be efficient but also to have minimal impact on the existing SR mechanisms. First, we discuss how to handle when invocations are examined in non-FCFS order. We then discuss how to handle the situation when invocations that are currently referenced by **rd** are serviced.

7.2.1 Non-FCFS Examination Ordering

Because of the iterative nature of **rd**, it is necessary for each process to identify which invocations in the class of operations that appear in the **rd** have been examined and which invocations have not. However, the order in which the process examines invocations may not necessarily be FCFS (as synchronization and scheduling expressions can be used to alter the order in which the invocations are examined). It is not sufficient for the executing process to keep just one pointer into the invocation queue indicating how far along the process has read through the queue.

One simple way to indicate whether or not an invocation has been read is to use a flag. One flag is needed for each process executing one **rd**. This approach, however, is not feasible for the implementation of **rd** for two reasons. First, processes can be dynamically created and thus the number of processes that executes **rd** might be changed during run-time. Second, it is difficult to handle nested **rd** statements in this approach. For example, for a given process, the RTS has to determine which **rd** is executing. The RTS also has to identify which flag is used for the inner **rd** and which flag is used for the outer **rd**. In addition, this approach has another drawback: it would be quite inefficient because the process might have to revisit the examined invocations while searching for an unread invocation.

We therefore implement **rd** using an *examination queue* in the most general cases.[5] An examination queue contains pointers to all unread invocations in the invocation queue in FCFS order. On each iteration of the **rd**, the process removes the pointer to the invocation selected for examination from the examination queue. The process also updates the examination queue to include the invocations that arrived after the previous iteration and to remove some invocations that have been serviced (typically by other processes) during this iteration of **rd** but that were included in the examination queue.

The RTS must maintain two kinds of information for each process executing an **rd**: the invocation that is currently being examined and the last invocation that has been included in the examination queue. For convenience, in the rest of this thesis, we use the acronym ICBE to refer to the invocation that is currently being examined and ILAST to refer to the last invocation that has been inspected by the process in the invocation queue.[6] After choosing an invocation to examine (but before executing the command body of the operation in **rd**), the process keeps a pointer to that invocation block so that other statements, such as **mark** and **fetch**, that reference the ICBE can be executed within the command body. In addition, the process keeps another pointer into the invocation queue to record how far along it has inspected the invocation queue; i.e., it points to the last invocation in the invocation queue that has been included in the examination queue. Thus, when new invocations are appended to the invocation queue, the process executing the **rd** is able to identify which invocations have not been inspected.

[5]In some cases, the RTS can avoid building an examination queue (see Section 7.3.1 for details).

[6]Note that an invocation that has been inspected does not necessarily mean that it has been examined. An invocation is examined only when the command body associated with the **rd** is executed for that invocation.

7.2.2 Removal of Serviced Invocations

As mentioned earlier, the invocation queue can be modified by other processes while an **rd** is examining an invocation from the invocation queue. In particular, the ICBE and ILAST of an executing **rd** can be removed and freed by another process. Removing such invocations might affect the execution of the **rd**.

We considered several alternatives in handling this problem. The simplest way is to keep the serviced invocations in the invocation queue until no processes reference it. However, this method would adversely affect the performance of **in** and **rd**. For example, to service an invocation (via a simple **in**), the process would just take the first invocation in the invocation queue in the SR implementation. However, if the serviced invocations are retained in the queue, the process would have to visit them before reaching the first serviceable invocation. The cost of searching for an invocation to service would thus be increased. As operations are the key communication mechanism in concurrent programs, we chose to keep only pending invocations in the invocation queue, as in the SR implementation; serviced invocations are removed from the invocation queue. Moreover, we decided to keep serviced invocations only if necessary. Otherwise, invocations can be freed immediately after they are serviced, as the SR implementation does.

As described in Section 7.2.1, the RTS maintains two pointers into the invocation queue for each process executing an **rd** for different purposes. The RTS uses the ICBE pointer to locate the invocation block of the ICBE. The RTS uses the ILAST pointer to locate the next invocation following the ILAST. Therefore, the ILAST of an executing **rd** can be removed and freed as long as the executing process can locate the next invocation following it. Thus, we handle the removal of ICBE and ILAST separately. For the former case, we use an additional flag to indicate whether or not an invocation has been serviced and use a reference count to maintain the number of processes that reference the invocation. We also maintain a list of serviced invocations whose reference counts are greater than zero in the class structure. The RTS maintains these fields as follows. When a process selects an invocation to examine, it increments the reference count in the invocation before releasing the access. When the process finishes examining the invocation, it informs the RTS and then decrements the reference count. When an invocation is serviced, the RTS determines whether or not some processes reference it. If so, the RTS moves the invocation block to the serviced invocation list; the invocation block will be freed when its reference count becomes zero.

For the latter case (i.e., the removal of the ILAST of an executing **rd**), the RTS maintains in each invocation block a list of processes that reference the invocation as its ILAST. When an invocation is serviced, the RTS notifies each process in this list to update its ILAST pointer to the preceding invocation in the invocation queue so that the process can still find the next invocation following it. The RTS then removes the invocation from the invocation queue as described above.

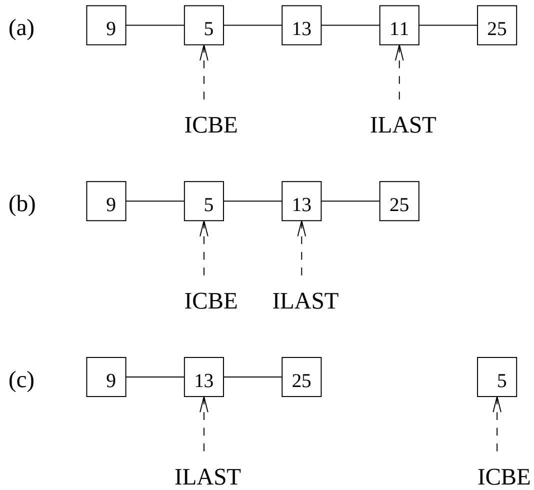

Figure 7.4: *Execution of an* **rd** — *(a) after releasing access to class (b) after* f(11) *has been serviced (c) after* f(5) *has been serviced.*

Figure 7.4 illustrates how the two pointers are updated when the ICBE and ILAST of an **rd** are serviced during an iteration of an **rd** that is examining the operation `f`. The ICBE and ILAST of the current iteration of an **rd** are invocations `f(5)` and `f(11)` respectively as shown in Figure 7.4(a). The executing **rd** has already released access to the class of `f`. No other processes are referencing the invocations of `f`; i.e., the reference count in invocation `f(11)` is 1 and the reference counts in other invocations are zero. During the execution of this iteration, invocations `f(11)` and `f(5)` are serviced by other processes. First, when invocation `f(11)` has been serviced, the RTS updates the ILAST pointer of the executing **rd** to point to the invocation that precedes invocation `f(11)` in the invocation queue, i.e., invocation `f(13)`. As the reference count in invocation `f(11)` is zero, the RTS then removes and frees that invocation immediately. Figure 7.4(b) shows the invocation queue after invocation `f(11)` has been serviced. After that, invocation `f(5)` is serviced by another process. As invocation `f(5)` is the ICBE of the executing **rd**, the RTS moves that invocation from the invocation queue to the serviced invocation list as shown in Figure 7.4(c). That invocation will be freed when the **rd** finishes examining it.

7.3 SRR Implementation Refinement

Section 7.2 described the general approach of the SRR implementation. This section focuses on the refinement of the SRR implementation. Specifically, we describe how we implement **rd** statements and nested **rd**, and **mark** and **take** statements.

7.3.1 Rd Statements

7.3.1.1 Definitions and Examples

The implementation divides **rd** statements into three disjoint types:

- *Unordered invariant* **rd**
- *Ordered invariant* **rd**
- *Variant* **rd**

An **rd** is *invariant* if the selection criteria for invocation examination (including the examination order) do not change over execution of the **rd**; i.e., for each invocation, its values of synchronization and scheduling expressions, if any, do not change over execution of the **rd**. The *unordered* invariant **rd** refers to the invariant **rd** statement that contains no scheduling expression. The *ordered* invariant **rd** refers to the invariant **rd** statement that contains one or more scheduling expressions. Note that the terms "unordered" and "ordered" indicate whether or not the programmer explicitly specifies the examination ordering using scheduling expressions in an **rd**. More precisely, the term "unordered" here does not mean that the invocations are examined randomly; instead, without scheduling expression, the **rd** examines invocations in the normal FCFS order (i.e., invocations' arrival order). The ordered invariant **rd**, on the contrary, examines invocations in other than FCFS order. In addition, for a given invocation, the value of a scheduling expression in an ordered invariant **rd** is the same across all iterations. That is, if the scheduling expression value of an invocation, say `f(x)`, is a in the first iteration of the **rd**, then the scheduling expression value of `f(x)` is a in all subsequent iterations. The **rd** statements other than invariant **rd** are therefore the *variant* **rd**.

Examples of each type of **rd** are given below. First, consider the following invariant **rd** statements:

```
rd f(x) -> ... dr

rd f(x) st x = 3 -> ... dr

rd f(x) by -x -> ... dr

rd f(x) st x > 0 by x -> ... dr
```

The first two **rd** are unordered invariant **rd** as they do not have any scheduling expression; the last two are ordered invariant **rd**. The first **rd** examines all pending invocations of `f`. It contains no synchronization expression; it simply examines all pending invocations, one at a time, in their arrival order. The second **rd** examines invocations of `f` for which parameter `x = 3`. It is an invariant **rd** because the sychronization expression references only a constant 3 besides the invocation parameter `x`. Thus, for a given invocation, the evaluation of the synchronization expression gives the same value at any time as long as the parameter value is not modified (see Section 9.5). The third **rd** examines invocations of `f` in non-increasing order of the invocation parameter `x`. The fourth **rd** examines only invocations of `f` for which parameter `x > 0` in non-decreasing order of `x`. These two **rd** statements are similar to the first two except that they examine invocations in other than FCFS order; their selection criteria for invocation examination do not change over execution of the **rd**.

Next, consider the following examples of variant **rd**:

```
rd f(x) st x > t -> ... dr

rd f(x) st x > 0 -> ...
[] g(y,z) st y = h(z) -> ...
dr

rd f(x) by x*r -> ... dr
```

The first **rd** reads invocations of `f` for which parameter `x > t`, where `t` is a variable local to the process executing this **rd** and is modified within the command body. As `t` is modified over execution of the **rd**, an invocation that was not selectable for examination over the past iterations of **rd** might become selectable in the next iteration. The second **rd** examines invocations of `f` whose parameter is positive and invocations of `g` for which `y = h(z)`. It is a variant **rd** because the synchronization expression on `g` invokes a procedure `h`. For a given input, this procedure `h` may not return the same result to the same process on any invocation. For example, `h` may use the value of a resource variable in evaluating the return value but that resource variable can be changed by other processes at any time. The third **rd** examines invocations of `f` in the order of `x*r`, where `x` is the invocation parameter and `r` is a resource variable. Other processes in the same resource could modify the value of `r` during execution of the **rd**. The scheduling expression for each invocation may therefore have different values at different points of execution.

These three types of **rd** are implemented differently. Note that the variant **rd** is the most general form. We could treat all **rd** as variant **rd** and have them implemented in the same way. However, as the implementation of variant **rd** is very expensive (see Section 8.2.1), we apply two kinds of optimizations to the unordered invariant **rd** and the ordered invariant **rd** to reduce the implementation cost (see below). Other possible optimizations will be discussed in Section 9.2.

7.3.1.2 Static Analysis

To perform optimizations on an **rd**, the behavior of the selection criteria of the **rd** must be determined. Currently, the SRR compiler employs a simple and conservative analysis to approximate such behavior statically. The compiler determines whether or not the selection criteria of an **rd** would change over its execution, i.e., whether or not the **rd** is invariant. The compiler analyzes the synchronization and scheduling expressions associated with an **rd**. It determines that the **rd** is invariant if the expressions reference only literals, constants[7], and invocation parameters and they contain no assignment statement. Otherwise, the compiler considers the **rd** to be the most general case: the variant **rd**. The optimizations are applied to **rd** statements that only specify operation names. The compiler treats **rd** statements that reference capability variables and operation arrays as variant **rd**. Clearly, this conservative analysis is inadequate for determining all optimizable cases, as defined in Section 7.3.1.1. For example, the compiler forbids the optimization of an **rd** that contains a synchronization expression referencing a variable other than constants and invocation parameters. The current compiler considers it a variant **rd**, even if that variable is not modified over execution of the **rd**. Section 9.2.2 will discuss other possible analysis that can be employed to detect other optimizable cases.

Below we describe briefly how these three types of **rd** are implemented and how the invariant **rd** can be optimized. The compiler determines whether or not an **rd** can be optimized and then generates intermediate code appropriately. Details of the implementation will be described in Section 7.5.

7.3.1.3 Variant Rd

The variant **rd** is the most general form of **rd** and has the most expensive implementation. Each process maintains an examination queue, as described in Section 7.2.1. The examination queue contains pointers to *all* unread invocations in the same order as in

[7]In SR's terminology, a literal is a value of a specified type. For example, 3 is an integer literal, 'a' is a character literal, and "abc" is a string literal. A constant, like a variable, holds a value of a specified type. However, its value remains the same for its lifetime. Constants are declared with the keyword **const**.

the invocation queue. On each iteration of **rd**, the process first updates its examination queue to include new invocations. The process then searches the examination queue for a selectable invocation. If the **rd** contains a scheduling expression, it has to search the entire examination queue for the invocation that is selectable and that also minimizes the scheduling expression. After the process completes its search, the process removes and frees the pointer to the selected invocation from the examination queue. It then informs the RTS and starts examining the selected invocation. In this most general case of **rd**, the selection criteria for examination may change during its execution. Thus, invocations that have been inspected but not yet read need to be revisited in all subsequent iterations. As a consequence, each invocation may possibly be accessed multiple times throughout the execution of the **rd**.

7.3.1.4 Ordered Invariant Rd

For the ordered invariant **rd**, each process maintains a list of pointers only to *selectable* invocations for examination. The process keeps the list ordered by the value of the scheduling expression. On each iteration of the **rd**, the executing process asks the RTS for the first invocation in this ordered list. When an invocation is examined, the pointer to it in the list is removed and freed. The cost of this type of **rd** is much less than that of variant **rd**. Each examined invocation is accessed only twice: the first access for including it into the ordered list and the second access for actual examining the invocation. We also refer to this list as an *examination queue*, although it is defined differently for the variant **rd**. Nonetheless, the examination queue serves for the same purpose in both cases.

7.3.1.5 Unordered Invariant Rd

The unordered invariant **rd** has the simplest and most efficient implementation. Each process maintains a pointer into the invocation queue indicating how far along the process has read through the queue. On each iteration of **rd**, the next invocation can be obtained simply by following the next pointer of the ILAST. Note that the ILAST pointer has a different meaning for the implementation of the unordered invariant **rd**, in which an examination queue is not used. It points to the last invocation that has been examined in the invocation queue. Invocations are examined in their arrival order and each invocation is accessed only once throughout the execution of the **rd**. No invocation revisiting is required.

7.3.2 Nested Rd Statements

We use an *rd structure* for each process to maintain all information for all types of **rd** statements. An rd structure contains the information described in Section 7.2.1. One rd structure is used for one **rd** statement. We use a stack to record all **rd** statements that are currently being executed in a process, thus reflecting the nesting structure. The top rd structure in the stack corresponds to the innermost **rd** that is executing. When an executing **rd** statement terminates, the RTS pops the first rd structure from the stack and continues the execution of the next **rd**.

7.3.3 Mark and Take Statements

Recall from Section 5.2 that the semantics of **mark** allows at most one invocation for each operation to be marked by a process. The **take** statement services a previously marked invocation of the specified operation. The RTS maintains a list of pointers to marked invocations for each process, at most one element for each operation. On marking the ICBE, the RTS first determines whether or not the operation has been marked. If the operation has been marked, the RTS just changes the pointer to the ICBE. Otherwise, the RTS appends a new pointer to the ICBE onto the marked invocation list (which requires dynamic memory allocation). When the process executes the **take**, the RTS searches the marked invocation list for the invocation of the specified operation to service.

7.4 Summary of SRR Key Primitives

The key primitives provided by the RTS to implement **rd**, **mark**, **take**, and **fetch** statements are given below.[8] We describe an **rd** as *complex* if it is an ordered invariant **rd** or a variant **rd**; i.e., the **rd**'s implementation uses an examination queue.

rd_init(*complex***)** — Initializes an rd structure for the currently executing **rd** statement. Parameter *complex* is true if the **rd** is complex and false otherwise.

rd_anyinv(*class***)** — Returns a pointer to the next uninspected invocation block in *class*. If there is no such invocation, the process blocks until new invocations arrive if the **rd** has no else clause. When a new invocation arrives, the RTS awakens the process and returns a pointer to the new invocation if the **rd** is not complex or else returns a null pointer. If it has an else clause, then the RTS returns a null pointer without blocking the process.

rdchk_inv(*op_cap***)** — Returns a pointer to the next invocation block of the operation named by *op_cap*. If there is no such invocation, the RTS returns a null pointer.

cpy_inv(*class, invocation***)** — Returns a copy of the invocation block pointed at by *invocation* in *class* and releases access to the class.

rd_else(*class***)** — Blocks the executing process until new invocations arrive. This primitive is used for implementing the else clause of an **rd** (see Section 7.5.4).

build_rdin(*class***)** — Appends a list of invocation block pointers that point to unexamined invocations in *class* to the process' examination queue and updates the ILAST pointer. This primitive can be used only if the **rd** is complex.

get_refinv(*class***)** — Returns a pointer to the next unexamined invocation block in *class* and updates the ILAST pointer. If there is no such invocation, a null pointer is returned. This primitive can be used only if the **rd** is complex and is used together with **insert_rdin**.

[8] Same as in Section 7.1.3, the **sr_**prefix and non-essential arguments are omitted.

insert_rdin(*invocation, value, sort***)** — Inserts a new pointer to *invocation* into the process's examination queue. If parameter *sort* is true, the RTS inserts the pointer to *invocation* into the queue ordered by *value*; otherwise, the RTS appends it onto the end of the queue. This primitive can be used only if the **rd** is complex and is used together with **get_refinv**.

finished_rd(*class, invocation***)** — Informs the RTS that the MC has finished executing the body of an **rd** statement.

rd_final() — Informs the RTS of the termination of an **rd** statement. The rd structure is freed if necessary.

mark(*class***)** — Marks the ICBE in *class*.

take_anyinv(*class***)** — Returns a pointer to the next invocation in *class* marked by that process. If there is no such invocation, a null pointer is returned.

grab_iop(*class, invocation***)** — Removes the invocation block pointed at by *invocation* from *class*, which was marked by the executing process. This primitive is a combination of **iaccess**(*class*) and **rm_iop**(*class, invocation*). If that marked invocation is found and removed, a pointer to that invocation block is returned; otherwise, a null pointer is returned.

fetch(*class***)** — Removes the ICBE from *class*. This primitive is similar to **grab_iop** except that the ICBE is removed rather than the marked one.

finished_take(*class, invocation***)** — Informs the RTS that the MC has finished executing the body of a **take** or **fetch** statement. The invocation block pointed at by *invocation* is freed if it is no longer needed. This primitive is similar to **finished_input**.

7.5 Examples of SRR Generated Code

Below, we present several examples and show pseudocode (in SR style) of their generated codes to illustrate how the primitives for **rd** are used by the MC. More complicated **rd** statements are implemented using appropriate combinations of the primitives. This section first presents examples for three types of **rd**: unordered invariant, ordered invariant, and variant. It then presents an example of an **rd** with an else command. That is, the optimizations of the most general form are discussed first. We present in this order because we think that it is easier for the readers to understand the generated code for the simplest form (unordered invariant **rd**) first rather than the most general form (variant **rd**).

7.5.1 Unordered Invariant Rd

As described in Section 7.3.1.5, the unordered invariant **rd** has the simplest and most efficient implementation. No examination queue is used.

To illustrate how the unordered invariant **rd** is implemented, first consider a simple **rd** statement:

```
rd f(x) -> ... dr
```

This statement reads all pending invocations of `f` and then delays the executing process until some invocation of `f` arrives. Suppose `f` is the only operation in class `c`. Figure 7.5 outlines the generated code for the above **rd** statement. Before the execution of an **rd**

```
rd_init(0)
do true ->
   iaccess(c,0)
   inv := rd_anyinv(c)
   inv := cpy_inv(c,inv)
   # command body
     ...
   finished_rd(c,inv)
od
rd_final()
```

Figure 7.5: *Generated code for a simple* **rd** *statement.*

statement, `rd_init` is executed to initialize the rd structure. When the **rd** terminates, `rd_final` is executed to cleanup the space used. An infinite loop is used to implement the iterative nature of an **rd**. For each iteration of the **rd**, the MC acquires access to class `c`, reads the next pending invocation, copies the invocation block, and executes the command body associated with `f` with parameter `x` bound to the value of `x` in the copied invocation block. When the command body terminates, the MC then frees the memory of the copied invocation block.

For brevity, in the following examples, we show only the generated code inside the loop body, i.e., the code for examining one invocation in an iteration. The initialization and finalization primitives (`rd_init` and `rd_final`) executed outside the loop are not shown.

Second, consider the following two-armed **rd** statement:

```
rd f(x) -> ...
[] g(y,z) -> ...
dr
```

This statement reads all pending invocations of either `f` or `g`. Since `f` and `g` appear in the same **rd** statement, they will be placed in the same class `c`. Figure 7.6 shows the generated code for this **rd**. For this **rd** statement, the MC first acquires access to class `c`, and then looks at each pending invocation in class `c` to determine if it is an invocation of `f` or `g`. Note that, compared to the loop body in Figure 7.5, an additional loop is used to examine each pending invocation in class `c` until an invocation of `f` or `g` is found. This is necessary because there might be additional operations in the class. If `f` and `g` are the only operations in class `c`, that loop can be removed as in the previous example. If the MC finds an invocation of `f` or `g`, it then copies the invocation block, executes the corresponding command body with the parameter values from the copy of the selected invocation block, and finally frees the copied invocation block. If the MC finds no pending invocation of `f` or `g`, the executing process blocks in `rd_anyinv` until an invocation in the class arrives. When such an invocation arrives, the RTS awakens the process, which then repeats the above steps.

Next consider another **rd** with a synchronization expression:

```
iaccess(c,0)
do true ->
   inv := rd_anyinv(c)
   if "inv is of op f" -> # generated code for operation arm f
      inv := cpy_inv(c,inv)
      # command body of f
        ...
      finished_rd(c,inv)
      exit
   fi
   if "inv is of op g" -> # generated code for operation arm g
      inv := cpy_inv(c,inv)
      # command body of g
        ...
      finished_rd(c,inv)
      exit
   fi
od
```

Figure 7.6: *Generated code for a two-armed* **rd** *statement.*

```
rd f(x) st x > 0 -> ... dr
```

This statement reads invocations of `f` for which parameter `x` is positive. Figure 7.7 shows the generated code for this **rd**. That code is very much like the code in Figure 7.6 except

```
iaccess(c,0)
do true ->
   inv := rd_anyinv(c)
   if "inv is of op f" and inv.x > 0 -> # evaluate synchronization expression
      inv := cpy_inv(c,inv)
      # command body
        ...
      finished_rd(c,inv)
      exit
   fi
od
```

Figure 7.7: *Generated code for an invariant* **rd** *with a synchronization expression.*

that only the code for the operation arm for `f` is generated and the synchronization expression is evaluated for each invocation before examination. Recall from Section 7.1.3 that we use the notation `inv.x` to represent the invocation parameter `x` in the invocation `inv`. The

MC uses the same steps as in the previous example to obtain a pending invocation of `f` in class `c`. For each such invocation, the MC evaluates the synchronization expression using the value of the parameter in the invocation block. If the synchronization expression is true, the MC examines the invocation as in the previous example. If the synchronization expression is false, the MC repeats to obtain the next invocation. Note that synchronization expressions are evaluated by the MC, not the RTS. The reasons for doing so will be discussed in the next subsection.

7.5.2 Ordered Invariant Rd

The ordered invariant **rd** is another type of **rd** that can be optimized. For each process executing an **rd**, the RTS maintains an examination queue that contains pointers only to selectable invocations for examination (see Section 7.3.1.4).

For example, consider an **rd** with a scheduling expression:

```
rd f(x) by x -> ... dr
```

This **rd** examines invocations of `f` in non-decreasing of `x`. Figure 7.8 outlines the generated code for this **rd**. That code assumes that `f` is the only operation in class `c` for simplicity. A similar approach as described for Figure 7.6 can be used to handle if operations other than `f` are in class `c`. The MC first informs the RTS that an examination queue needs to be built using the primitive `rd_init` and it then uses `iaccess` to obtain exclusive access to `c` (as it does in all cases). The MC then updates the examination queue to include all selectable invocations before selecting an invocation to examine. It does so by executing `get_refinv(c)` repeatedly to obtain invocations one at a time that have not been included into the examination queue. Note that the invocation queue is locked while the MC is updating the examination queue. Thus, no invocations would be appended on the invocation queue while updating the examination queue or searching for an invocation to examine; instead, the new invocations are appended onto the new invocation list (see Section 7.1.2). For each such invocation, the MC evaluates the scheduling expression and inserts it into the examination queue in sorted order. When all invocations in the queue

```
iaccess(c,0)
# build/update the examination queue
do (inv = get_refinv(c)) ->
   value = inv.x  # value of scheduling expression
   insert_rdin(inv,value,1)
od
do true ->
   inv := rd_anyinv(c)
   if "inv is null" ->
      # update the examination queue as above
      /* same loop as above */
   [] else ->
      inv := cpy_inv(c,inv)
      # command body of f
        ...
      finished_rd(c,inv)
      exit
   fi
od
```

Figure 7.8: *Generated code for an ordered invariant* **rd** *statement.*

are included in the examination queue (i.e., when `get_refinv(c)` returns a null pointer), the MC uses `rd_anyinv` to obtain an invocation of `f` from the examination queue with an indirect address reference. If the MC finds an unread invocation from the examination queue, the MC uses the same steps as in the previous examples to examine that invocation. If the MC finds no invocation in the examination queue, the executing process blocks in `rd_anyinv` until an invocation in the class arrives. When such invocation arrives, the RTS awakens the process and `rd_anyinv` returns a null pointer. The MC then updates the examination queue to include the new invocation and repeats the above steps.

Next consider modifying the above **rd** to add a synchronization expression:

```
rd f(x) st x > 3 by x -> ... dr
```

Like the generated code in Figure 7.8, the MC updates the examination queue before selecting an invocation to examine. Before including an invocation in the examination queue, the MC evaluates the synchronization expression for each invocation returned from `get_refinv(c)`. It inserts only invocations whose synchronization expressions are true. For each invocation, the synchronization and scheduling expressions are evaluated only *once* while updating the examination queue.

Note that an examination queue is used for the executing process for this type of **rd**. Synchronization and scheduling expressions are evaluated when maintaining the examination queue in sorted order rather than when selecting an invocation to examine. As discussed in [3, 28], synchronization and scheduling expressions are evaluated by the MC, not the RTS, for two reasons. First, these expressions can reference objects such as local variables for which the RTS would need to establish addressing if it were to execute the code that evaluates the expression. Second, these expressions can contain invocations; it would greatly complicate the RTS to handle such invocations in a way that does not cause the RTS to block itself. As a consequence, the overhead of evaluating synchronization and scheduling expressions is paid for only processes that use them in this approach.

7.5.3 Variant Rd

The variant **rd** has the most expensive implementation. As described in Section 7.3.1.3, each process executing an **rd** maintains an examination queue that contains pointers to all invocations in the invocation queue.

First, consider a variant **rd** with a synchronization expression:

```
rd f(x) st x > t -> ... dr
```

The above synchronization expression on `f` references a variable `t`. The generated code for this **rd** is shown in Figure 7.9. The MC first acquires access to class `c` and then uses the primitive `build_rdin` to update the examination queue. This primitive includes *all* invocations in the class, unlike what the ordered invariant **rd** does. The MC then uses `rd_anyinv` to obtain an invocation from the examination queue with indirect address references. For each such invocation, the MC evaluates the synchronization expression using the value of the parameter in the invocation block. If the synchronization expression is true, the MC notifies the RTS of its success to examine the invocation as it does for other **rd** statements. If the synchronization expression is false, the MC repeats the above

```
iaccess(c,0)
build_rdin(c)  # build/update the examination queue
do true ->
   inv := rd_anyinv(c)
   if "inv is null" -> # end of examination queue reached
      # new invocations arrive
      build_rdin(c)
   [] inv.x > t -> # evaluate synchronization expression
      inv := cpy_inv(c,inv)
      # command body of f
        ...
      finished_rd(c,inv)
      exit
   fi
od
```

Figure 7.9: *Generated code for a variant* **rd** *statement with a synchronization expression.*

steps to obtain the next invocation. When the MC finds no invocation in the examination queue, the executing process blocks (as described in Section 7.5.2) until an invocation in the class arrives. When such an invocation arrives, the RTS awakens the process, which then updates the examination queue to include the new invocation and repeats the above steps.

As another example, consider the following variant **rd**:

```
rd f(x) by x*r -> ... dr
```

This statement contains a scheduling expression `x*r` which references a variable `r`. Figure 7.10 shows the generated code for this **rd**. The MC uses the same steps as in the previous example to obtain the first invocation of `f`. It then evaluates the scheduling expression using the value of `x` as well as that of `r`; it saves the evaluated value and a pointer, `psave`, to the invocation block. The MC then obtains the remaining invocations one at a time by repeatedly executing the primitive `rdchk_inv(c)`. For each of these invocations, the MC evaluates the scheduling expression, compares it with the saved value, and updates the saved value and pointer if the new value is smaller. When there are no more invocations (i.e., when `rdchk_inv` returns a null pointer), `psave` points to the invocation with the smallest scheduling expression. The MC examines that invocation as in the previous examples.

Note that, unlike for the ordered invariant **rd** in Section 7.5.2, synchronization and scheduling expressions are evaluated when the MC selects an invocation to examine on each iteration.

```
iaccess(c,0)
build_rdin(c)  # build/update examination queue
do true ->
   inv := rd_anyinv(c)
   if "inv is null" -> # end of examination queue reached
      # new invocations arrive
      build_rdin(c)
   [] else ->
      # search for the invocation that minimizes scheduling expression
      psave := inv; value := inv.x*r # scheduling expression
      do (inv := rdchk_inv(c)) ->
         newvalue = inv.x*r
         if newvalue < value ->
            psave := inv; value := newvalue
         fi
      od
      inv := cpy_inv(c,psave)
      # command body of f
        ...
      finished_rd(c,inv)
      exit
   fi
od
```

Figure 7.10: *Generated code for a variant* **rd** *statement with a scheduling expression.*

7.5.4 Rd with else command

The else command of an **rd** is executed when no invocations are pending or all pending invocations have been examined (see Section 5.1). The executing process blocks immediately after executing the else command unless it executes a statement, such as **exit**, that can terminate the **rd**.

To illustrate, consider the following **rd** statement:

```
rd f(x) st x = 3 -> ...
[] else -> ...
dr
```

The above **rd** examines invocations of `f` for which parameter `x` is equal to 3. The generated for the above **rd** statement is given in Figure 7.11. The MC uses the same steps as for the

```
else_executed := false
iaccess(c,1)
do true ->
   inv := rd_anyinv(c)
   if "inv is null" ->
      if not else_executed -> # execute else command
         else_executed := true
         # else arm body
           ...
      fi
      rd_else(c) # block the process
   [] inv.x = 3 -> # evaluate synchronization expression
      inv := cpy_inv(c,inv)
      else_executed := false
      # command body of f
        ...
      finished_rd(c,inv)
      exit
   fi
od
```

Figure 7.11: *Generated code for an* **rd** *with an else command.*

rd without the else command to select an invocation to examine (see Figure 7.7). With an else command, an additional flag `else_executed` is used in the generated code to control when the else command body should be executed. When there is no pending invocation or all invocations have been examined, the `rd_anyinv` returns a null pointer. The MC then checks whether or not the else command has been executed and no invocations have been examined since then. If not, it executes the else command body and sets the flag. The MC then uses the primitive `rd_else` to block the executing process until an invocation in the class arrives. When such an invocation arrives, the RTS awakens the process, which then repeats the steps to obtain the new invocation. The MC evaluates the synchronization expression for such invocation. If the synchronization expression is true, the MC examines this invocation and sets the flag `else_executed` to false so that the else command will be executed before the process blocks the next time. If the synchronization expression is false and the MC finds no more pending invocation to be examined (i.e., `rd_anyinv` returns a null pointer), the MC then uses `rd_else` to block the process again immediately without executing the else command (as `else_executed` is true).

7.6 Effort

The design and coding of the implementation took about six man-months. We spent about one man-month to learn the current implementation of SR in the beginning. The implementation of our new mechanisms involves extensions only to the compiler and the RTS. The current SR compiler consists of approximately 20,000 lines of code. The RTS consists of about 13,000 lines of code, in which roughly 2,000 lines of code are for operation handling. Comments and whitespaces constitute about 20% of the code for the compiler and about 30% of the code for the RTS.

We extended the compiler with approximately 1,200 lines of code. We also extended the RTS with approximately 1,600 lines of code, in which 500 lines of code are written for optimizations and specialized RTS procedures. Comments and whitespaces constitute about 20% and 25% of the extended code for the compiler and the RTS respectively. Several of our new RTS procedures are similar to those in the current SR implementation, but they were separately developed for implementing different statements. In particular, the RTS procedures for implementing an input statement needed to be modified to support **rd** (see Section 7.2.2). However, instead of modifying the procedures, `sr_rm_iop` and `sr_receive`, we copied these two procedures from SR and then modified them accordingly. These two procedures consist of about 300 lines of code.

Chapter 8

Performance

In Chapters 5 and 6, we explored the programming style that results from using the new mechanisms in SRR. In this chapter, we evaluate the new mechanisms quantitatively. We use *micro-benchmarks* to measure the performance of individual mechanisms. We also use *macro-benchmarks* to measure the performance of more realistic programs that consist of many language elements.

Appendix A shows all micro-benchmark test programs and some macro-benchmark test programs. The other macro-benchmark test programs are shown in Chapter 6. Our results show that the performance of the new mechanisms is reasonable. The most general case of **rd** is necessarily expensive to implement; however, the optimized **rd** is very efficient. Simulations using **in** of **rd** are not as efficient as the **rd** statements. To examine an invocation, the **rd** takes only 39% of the execution time for its simulation using an **in** together with a **forward**. The input statement can also be simulated using **rd**. In general, the simulation using **rd** of an **in** is slower than the real **in** by 46%. However, the simulation using an optimized **rd** can outperform the real input statement in some cases when a synchronization or scheduling expression is used. Finally, the macro-benchmarks illustrate that the overall performance of some of the applications described in Chapters 3 and 6 can be improved when rewritten using the new mechanisms.

8.1 Performance Measurement

In the current SRR implementation, as described in Section 9.1.5, the RTS allocates and later releases memory for an invocation block on each iteration of the **rd** to avoid the parameters in the actual invocation block being modified by **rd** and also to avoid getting inconsistent parameter values during examining the invocation block. However, the implementation could be improved by employing more sophisticated analysis on a program to determine whether an **rd** can directly reference the actual invocation block (which is the commonly occurring case in typical programs). We did not attempt to implement this optimization because dataflow analysis is required, which is outside the scope of our work. For performance evaluation purposes, the SRR implementation allows the user to set an environment variable indicating whether or not this optimization can be applied rather than having the compiler decide by performing the real analysis.

The test programs do not modify invocation parameters within the command body of **in** or **rd**. Thus, we ran the test programs using the optimized **rd** implementation without "invocation block copying" and obtained the results reported in the chapter. To get some feel about the extra overhead due to the invocation block copying, we also measured and compared the execution times for a simple test program with and without invocation block copying separately.

The Unix *time* command was used to determine the total CPU times for the test programs, i.e., the total of CPU time the program runs in user mode and CPU time the program runs in kernel mode. We measure the total CPU time for each test program to reflect the overhead due to the memory allocation and deallocation performed by a process. The test programs were run on the four different machines listed in Table 8.1, each of which has a separate instruction cache (I) and data cache (D). All systems were lightly loaded when timing tests were run. Each test program was run ten times in a single SR virtual machine; variances between execution times were very small. The numbers presented in this chapter are the median values from the ten tests.

Machine	*Operating System*	*Memory*	*Cache*		
			Primary		*Secondary*
DEC Alpha 3000/400	OSF 3.2	64M	8K (I)	8K (D)	512K
DECstation 260	Ultrix 4.3	40M	16K (I)	16K (D)	1024K
SGI Indigo	IRIX 5.3	32M	8K (I)	8K (D)	1024K
Sun SPARCstation 5	SunOS 4.1.3	24M	16K (I)	8K (D)	—

Table 8.1: *Details of four different machines.*

Furthermore, we measured the number of execution cycles (i.e., instruction cycles) for the test programs using the utilities *pixie* and *prof* on the Alpha and the DECstation. By counting the execution cycles, the effect of caching is eliminated. Appendix B shows the statistics obtained from different machines. In the remainder of this chapter, we first discuss the performance of the new mechanisms based on the data obtained from the DEC Alpha 3000/400. We then discuss detailed effects of different architectures separately.

8.2 Micro-benchmarks

8.2.1 Cost of Rd statement

Recall from Section 7.3 that different types of **rd** — unordered invariant, ordered invariant, and variant **rd** — have different implementations. This section presents the performance results for these three types of **rd**. It also reports and compares the execution times for a simple **rd** with and without invocation block copying.

Three test programs, one for each type of **rd**, are included in this test. Each test program generates a given number of invocations and then examines them via an **rd**. Finally, it services all invocations. The test programs are named in the way to represent the form of the **rd** used as shown in Table 8.2. rd uses a simple **rd** to examine invocations in their

Type of **rd**	*Test Program*
Unordered Invariant **rd**	rd
Ordered Invariant **rd**	rd+by
Variant **rd**	rd+by+var

Table 8.2: *Types of* **rd**.

arrival order. rd+by uses an **rd** that contains a scheduling expression (`by x`) to examine invocations in increasing order of the invocation parameter value `x`. rd+by+var examines invocations in increasing order of `x*n`, where `n` is a variable with value 1. Note that our current implementation treats this **rd** as a variant **rd** conservatively although the variable `n` is not modified over the execution of the **rd**. As described in Section 7.3.1.2, the compiler considers an **rd** of variant type if its synchronization or scheduling expression contains a variable. rd+st+var+fetch and rd+by+var+fetch in Section 8.2.4.4 are considered variant **rd** statements for the same reasons. For simplicity, we used the simple expression `x*n` in the synchronization and scheduling expressions in the test programs instead of a complicated one. In fact, we could extend the range of optimizables **rd** by employing more complicated analysis, as described in Section 9.2.2.

For comparison purposes, the invocations are generated with increasing parameter values so that the **rd** in each test program results in examining invocations in their arrival order even when a scheduling expression is used. The number of invocations to be generated and examined is given as a parameter to the test programs. Each of the three test program was run with various values for this parameter from 0 to 1000. To determine the cost of invocation examination, we also measured the cost of invocation generation and service in a separate test program (which will be described in Section 8.2.2) and subtracted that cost from the execution times for each of the three test programs.

Figure 8.1 shows the performance results for each type of **rd** that examines all pending invocation once. The execution times are increasing in the order rd, rd+by, rd+by+var. In particular, the execution time for rd+by+var increases tremendously as the number of pending invocations. These differences in the execution times reflect the differences in the implementations (see Chapter 7 for details). rd uses an unordered invariant **rd**, which has the least costly implementation. The process executing the unordered invariant **rd** maintains a pointer into the invocation queue indicating how far the process has read through the queue. Each invocation in the invocation queue is accessed only *once* throughout the execution of the **rd**. Therefore, the execution time for an unordered invariant **rd** linearly

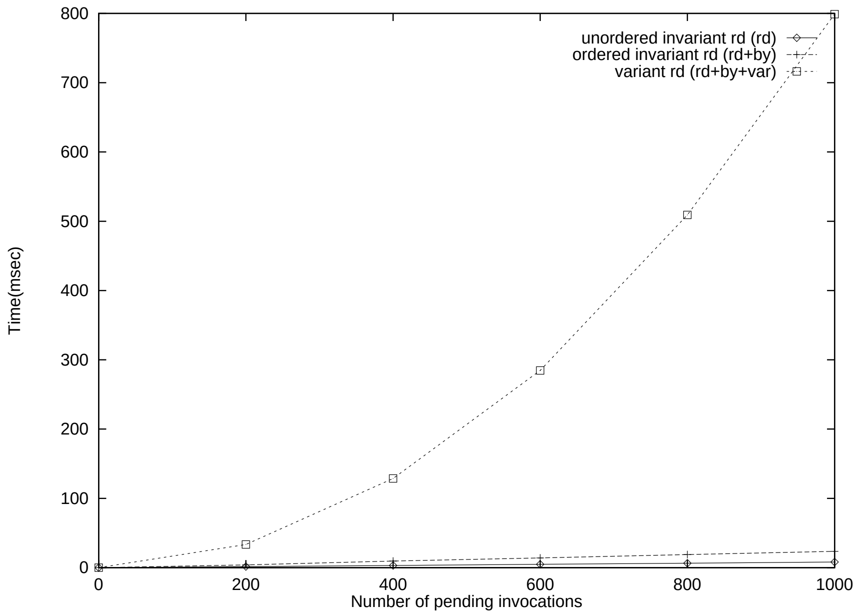

Figure 8.1: *Performance of three types of* **rd**.

increases as the number of invocations increases.

rd+by, an ordered invariant **rd**, maintains an examination queue in the scheduling expression order, i.e., in the order of the invocation parameter value. To include an invocation in the examination queue, it searches the queue from the end towards the front until it finds one with a smaller parameter value. To examine an invocation from the examination queue, it simply removes the first one. As the invocations in the invocation queue are already in order, the process just appends elements, one for each invocation, onto the end of the examination queue; hence, the searching time for including an invocation in the examination queue is negligible. (Section 8.2.4.3 shows the differences in this searching time for different sequences of pending invocations.) On each iteration of the **rd**, the process removes the first invocation from the examination queue and then examines it. Thus, the difference between the execution times for rd and for rd+by is mainly due to the additional scanning of the invocation queue as well as the memory allocation and deallocation for building the examination queue.

rd+by+var uses a variant **rd**, which has the most costly implementation. The process includes all invocations in the examination queue in their arrival order. To examine an invocation, the process searches the entire examination queue for the invocation that minimizes the scheduling expression. This queue traversal together with the memory allocation and deallocation cost for the examination queue accounts for the large differences in execution times for these test programs. The execution time for rd+by+var quadratically increases as the number of pending invocations increases. In particular, as rd+by and rd+by+var build the examination queue in a similar way, the difference between their execution times is largely due to the cost of the queue traversal (once on each iteration). rd+by+var selects an invocation to examine in the way same as an input statement with a scheduling expression that selects an invocation to service. Section 8.2.4.3 compares the performance of such an input statement and its simulation using **rd**. The queue traversing time for selection of an invocation contributes to a significant part of the cost of both the input statement and the **rd**. Compared to the cost of the input statement, the additional overhead in the **rd** results from the cost for building the examination queue (i.e., the initial scanning of the invocation queue and the memory allocation and deallocation for the examination queue).

We also ran the test program rd with invocation block copying to measure the overhead due to memory allocation and deallocation on each iteration of an **rd**. The execution times for an **rd** to examine an invocation with and without invocation block copying are shown in Table 8.3. An **rd** with invocation block copying takes about 16.640μs (247 cy-

Mechanism	*Time(μs)*	*Cycles*
rd without invocation block copying	9.743	325
rd with invocation block copying	26.383	572

Table 8.3: *Results with and without invocation block copying.*

cles) more to examine an invocation than the same **rd** without invocation block copying. Compared to the **rd** without invocation block copying, the memory allocation and deallocations for each invocation costs 1.71 times as much as the execution time for examination of an invocation. However, as the cost of invocation examination constitutes only a small fraction of the execution time of a program, the invocation block copying only slightly affects the overall performance of a program. In terms of the number of execution cycles, invocation block copying costs only 0.76 time as much as the cost for examination of an invocation. The large difference between the relative difference in the execution times and that in the execution cycles (which will also be seen in other results presented in this chapter) is presumably due to cache misses.

8.2.2 Overhead due to Rd Implementation

The implementation of **rd** introduces some overhead to existing SR mechanisms. As described in Section 7.2, the invocation block structure as well as the class structure is modified. The cost of generating an invocation has slightly increased. In addition, the cost of servicing an invocation is also affected. To illustrate, consider what happens when an invocation is serviced under the SRR implementation (see Section 7.2.2 for details). When an invocation is serviced, the RTS checks whether or not some other processes are referencing it. If not, the RTS removes the invocation block from the invocation queue and frees it, as it does in the SR implementation. If some processes are currently referencing it, the RTS moves it to the serviced invocation list maintained in the class structure. The RTS also performs some additional manipulation in the invocation block so as to guarantee that the removal of the invocation does not affect execution of other processes that are executing **rd**. In contrast to the SR implementation, the RTS performs much more work when an invocation is serviced due to the complicated semantics of the **rd**, thus introducing additional overhead to the cost of invocation service.

This section presents the impact on the performance of invocation and service due to the implementation of **rd**. A separate test program was developed to determine the cost of invocation generation and service. This test program first generates a given number of invocations and then services the invocations using an **in** statement. The test program was compiled using both the SRR compiler and the SR compiler and then run to measure the difference in their execution times. Currently, several new RTS primitives are added in the SRR implementation to implement the input statement as described above. In addition, the original RTS primitives for servicing an invocation in SR are also retained in SRR. The original RTS primitives are less costly than the new RTS primitives. The new RTS primitives are used only for servicing an operation that is serviced by an input statement and that is also examined by an **rd** statement. For servicing operations that do not appear in any **rd** statements, the original RTS primitives are used. The SRR compiler detects whether an operation that is serviced by an input statement appears in an **rd** in a program and generates code for the input statement accordingly.

Table 8.4 lists the cost for generating and servicing an invocation. Compared to

Implementation	*Time(μs)*	*Cycles*
SR	21.582	653
SRR (operation does not appears in any **rd**)	25.402	692
SRR (operation appears in an **rd**)	26.160	707

Table 8.4: *Overhead due to Rd Implementation.*

the SR implementation, the cost of invocation generation and service takes about 18% more time (6% more cycles) if the operation does not appear in any **rd** statement. If the operation appears in an **rd**, it takes 21% more time (8% more cycles).

8.2.3 Simulation of Rd Statement

This test employed two test programs to compare the performance of the real **rd** with the simulation using **in** of **rd** described in Section 3.1.1. One test program, in+forward, examines invocations without preserving their arrival order; another test program, in+forward+order, examines invocations while preserving their arrival order. Each test program generates and examines 75,000 invocations; it finally services them. The test programs were compiled using the SR compiler and run. The two test programs have the same effect as the test program rd described in Section 8.2.1.

Their execution times for examining one invocation are given in Table 8.5. They are determined in the same way as described in Section 8.2.1. As shown in Table 8.3, the

Mechanism	*Time(μs)*	*Cycles*
in+forward	24.644	675
in+forward+order	31.440	727

Table 8.5: *Results of simulation of* **rd**.

execution time for the real **rd** to examine one invocation took about 9.743μs (325 cycles). Compared to the execution time (and the execution cycles) for the real **rd**, in+forward takes about 2.53 (2.08) times as long, whereas in+forward+order takes about 3.23 (2.24) times as long. The simulation using **in** of **rd** is much less efficient than the real **rd** because each invocation being examined is serviced once and re-invoked once (see Section 3.1.1). It thus requires additional memory allocation and deallocation for each invocation block.

8.2.4 Simulation of Input Statement

As seen in Section 6.3, the input statement can be simulated using **rd** in SRR. The cost of servicing an invocation depends on what the pending invocations are and also depends on the servicing mechanism used. The test suite, therefore, is divided into 3 different parts depending on whether synchronization and scheduling expressions are used in the input statement. As an **rd** with a synchronization or scheduling expression can be optimized (see Section 7.3.1), we measured and compared the performance of simulations using both optimized and unoptimized **rd** of an input statement. As in Section 8.2.1, the servicing mechanisms are named to represent the form of the statement used. The test programs using **in** were compiled using the SR compiler whereas the test programs using **rd** were compiled using the SRR compiler. The results presented in this section are the execution times for invocation generation and service.

8.2.4.1 Simple Input Statement

This test includes an SR test program, in, which generates 75,000 invocations and then services them via an input statement. There are two ways to simulate the input statement using **rd** in SRR. The first test program, rd+mark+take, uses **mark** on each iteration of

rd to mark the invocation that is currently being examined and then services the marked invocation via **take**. The second test program, rd+fetch, uses the abbreviation **fetch** to service an invocation on each iteration of **rd**. These two test programs have the same effect as the test program in.

Table 8.6 lists the execution times of these three test programs. Compared to the

Servicing Mechanism	*Time(μs)*	*Cycles*
in	21.716	658
rd+mark+take	55.698	1467
rd+fetch	31.633	914

Table 8.6: *Results from simulation of a simple* **in**.

execution time (and the execution cycles) for in, rd+mark+take takes about 2.56 (2.23) times as long, whereas rd+fetch takes about 1.46 (1.39) times as long. A **mark** together with a **take** used for servicing an invocation is more expensive than a **fetch** statement. As described in Section 7.3.3, the implementation of the process executing **mark** and **take** statements maintains a list of marked invocations, one for each marked operation. On marking an invocation, the process searches the list to check whether or not an invocation of the operation has previously been marked but not yet been taken. If no such invocation is found, the RTS allocates memory for an element of the marked invocation list. When the process services a marked invocation via **take**, it also searches the list for an appropriate invocation to service. On the other hand, the **fetch** only services the invocation that is currently being examined via **rd**. One major factor contributing to the difference between their execution times is the cost of memory allocation and deallocation for elements in the marked invocation list. Another factor is the searching on the marked invocation list done on each execution of **mark** and **take**. (In fact, this searching time costs very little in this test because the **rd** statements used have only one operation arm.)

The simulation using **rd** and **fetch** takes 46% more time (39% more cycles) than the input statement. As **rd** and **fetch** are two separate mechanisms, the SRR compiler generates several procedure calls to the RTS for implementing them. In contrast to the implementation of the input statement, the simulation of the input statement costs a few more procedure calls to the RTS to service an invocation. Another factor that contributes to the difference in their execution times is the overhead due to the implementation of **rd**, which costs almost half of the difference (see Section 8.2.2).

8.2.4.2 Input Statement with Synchronization Expression

In this test, we wanted to measure the performance of an input statement with a synchronization expression and its simulation using **rd** and **fetch**. In addition, we wanted to see how their performances are affected when different sequences of invocations are serviced. We employed only an optimized **rd** (i.e., an unordered invariant **rd**) to simulate an input statement in this test; simulation using an unoptimized **rd** will be discussed in Section 8.2.4.4.

Two servicing mechanisms were employed in this test. One mechanism, in+st, is the real input statement and the other, rd+st+fetch, is its simulation using **rd** and **fetch**. rd+st+fetch and in+st each contains a synchronization expression `x > 0`, where `x` is the invocation parameter; i.e., they service invocations whose parameter is positive. Two invocation sequences were serviced via these mechanisms. Each sequence contains 1000 invocations with one parameter of integer type. SEQ generates all invocations with positive parameter values. ALT generates invocations with alternating positive and negative parameter values.

Table 8.7 summarizes the cost for an input statement and its simulation using **rd** to service these two sequences of invocations. The execution time and the execution cycles shown represents the average cost of generating and servicing one invocation in μs. The execution times and execution cycles for in+st to service the two sequence of

Servicing Mechanism	*Invocation Sequence*	*Time(μs)*	*Cycles*
in+st	SEQ	22.918	641
in+st	ALT	291.118	17572
rd+st+fetch	SEQ	32.333	920
rd+st+fetch	ALT	41.500	1144

Table 8.7: *Results from simulation of* **in** *with synchronization expression.*

invocations — SEQ and ALT — vary considerably (1170% difference in execution times and 2641% difference in execution cycles). The differences in their execution times reflect the implementation of the synchronization expression in **in** statements. To service an invocation, the process searches the invocation queue until it finds the first serviceable invocation. This searching time increases as the number of non-serviceable invocations pending before the first serviceable one increases. For example, the execution time for servicing SEQ is the shortest because all invocations are serviceable and the process just takes the first invocation in the invocation queue.

In contrast, the execution times (and the execution cycles) for rd+st+fetch to service invocations in SEQ and ALT differ in 28% (24%). The relatively small differences (compared with the differences in the execution times for in+st) indicate that the distribution of the non-serviceable invocations in the invocation queue does not affect the performance of the invariant **rd**. The **rd** in rd+st is an unordered invariant **rd** and implemented in the same ways as a simple **rd**. As described in Section 7.3.1.5, each invocation in the invocation queue (either serviceable or non-serviceable) is accessed only once throughout execution of the **rd**.

Another sequence of invocations, NEG, were also tested to further illustrate the difference in the implementation of the input statement and its simulation using an invariant **rd**. NEG generates 1000 invocations, in which a given number of invocations have negative parameter values and the rest have positive parameter values. The two test programs were run with varying number of invocations with negative parameters from 5 to 25. Figure 8.2 shows the cost of generating and servicing one invocation with different number of non-serviceable invocations. As expected, the execution time for rd+st+fetch is independent of the number of invocations with negative parameter values. On the other hand, the execution time for in+st to service an invocation increases as the number of non-serviceable invocations increases. Note also that if the number of non-serviceable invocations is greater than 17, the cost of the input statement becomes more expensive than that of its simulation using **rd**.

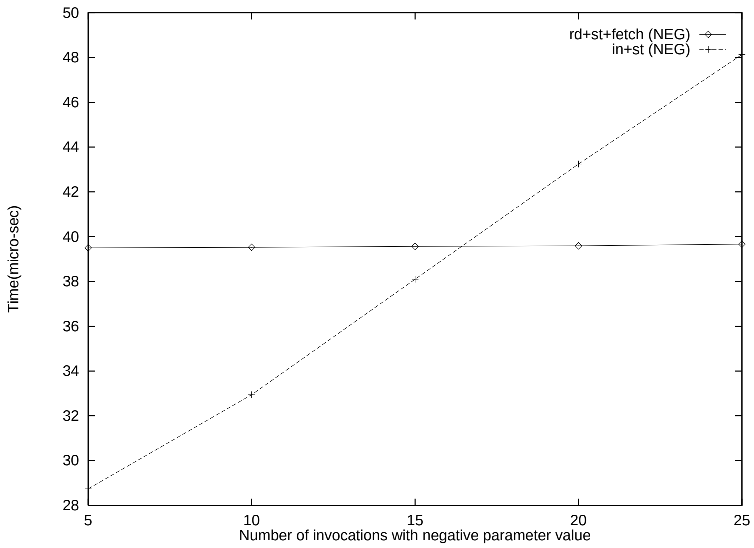

Figure 8.2: *Results from simulation of* **in** *with synchronization expression servicing NEG.*

Compared the execution times for rd+st+fetch to those for in+st, the simulation using optimized **rd** can outperform the real **in** statements in some cases depending on the distribution of the non-serviceable invocations in the invocation queue.

8.2.4.3 Input Statement with Scheduling Expression

Similar to the previous test, we measured the execution times of an input statement with a scheduling expression and its simulation using **rd** and **fetch**. in+by and rd+by+fetch service invocations in ascending order of the invocation parameter. Two different sequences of invocations were serviced in these test programs. ASCEND generates invocations with ascending parameter values. DESCEND generates invocations with descending parameter values.

Each of the two test programs was run with varying number of pending invocations. Figure 8.3 shows the performance of in+by and rd+by+fetch that services invocations in these two different sequences of invocations. Executions of rd+by+fetch to service invocations in both ASCEND and DESCEND are much faster than those of in+by. rd+by+fetch takes about 6–27% of the execution time for in+by to service invocations in ASCEND; it

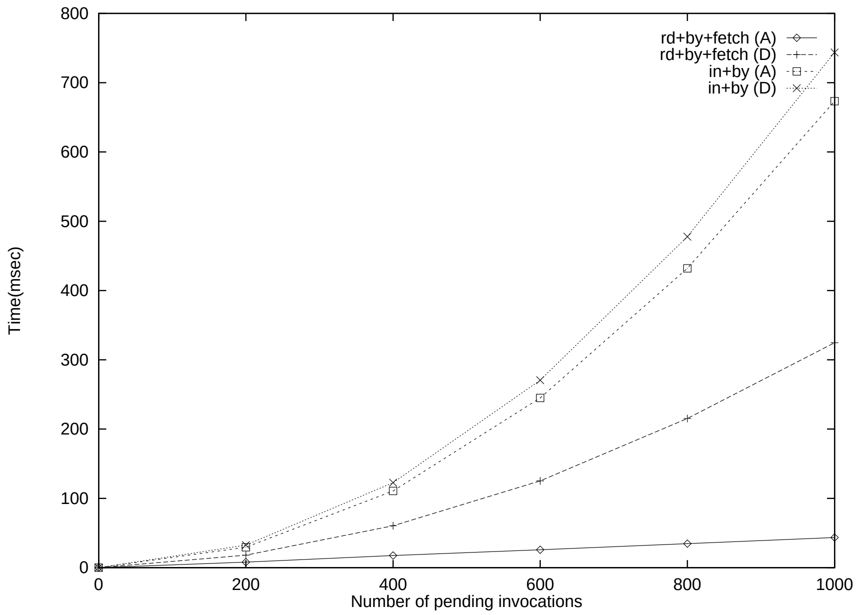

Figure 8.3: *Results from simulation of* **in** *with scheduling expression.*

takes about 44–54% of the execution time for in+by to service invocations in DESCEND. The differences increase as the number of pending invocations increases. As described in Section 8.2.1, the process executing rd+by maintains invocations in the examination queue in the scheduling expression order. To select an invocation, the process just takes the first invocation from the examination queue. On the other hand, the process executing in+by searches the entire invocation queue for the invocation that minimizes the scheduling expression to service. Despite the cost for building the examination queue required by rd+by (see Section 8.2.1), the simulation using ordered invariant **rd** of **in** is much more efficient than the real **in** with a scheduling expression.

Note that the **rd** can take advantage of the arrival order of the pending invocations whereas the **in** cannot. As shown in Figure 8.3, the ordered invariant **rd** is more efficient in servicing invocations in ASCEND than those in DESCEND. The invocations in ASCEND are already in order in the invocation queue. Thus, as described in Section 8.2.1, the searching time to include an invocation in the examination queue is very small because the process just appends elements onto the end of the examination queue, one at a time.

However, the cost for building the examination queue for invocations in DESCEND (which is the worst case invocation ordering) is more expensive because the process searches the entire examination queue before finding an appropriate place to include an invocation.

8.2.4.4 Simulation using Variant Rd

As discussed in Section 7.3.1, two kinds of optimizations can be applied to **rd** to improve its execution cost. Sections 8.2.4.2 and 8.2.4.3 presented the performance of the optimized **rd**. This section presents the performance of the most general form of **rd**, i.e., the variant **rd**.

This test employed two variant **rd** statements to simulate input statements with a synchronization expression or a scheduling expression. rd+st+var+fetch and in+st+var each contains a synchronization expression `x*n > 0`; rd+by+var+fetch and in+by+var each contains a scheduling expression `x*n`, where `x` is the invocation parameter and `n` is a variable with value 1. Each test program generates and services a given number of invocations with ascending parameter values, which are all positive.

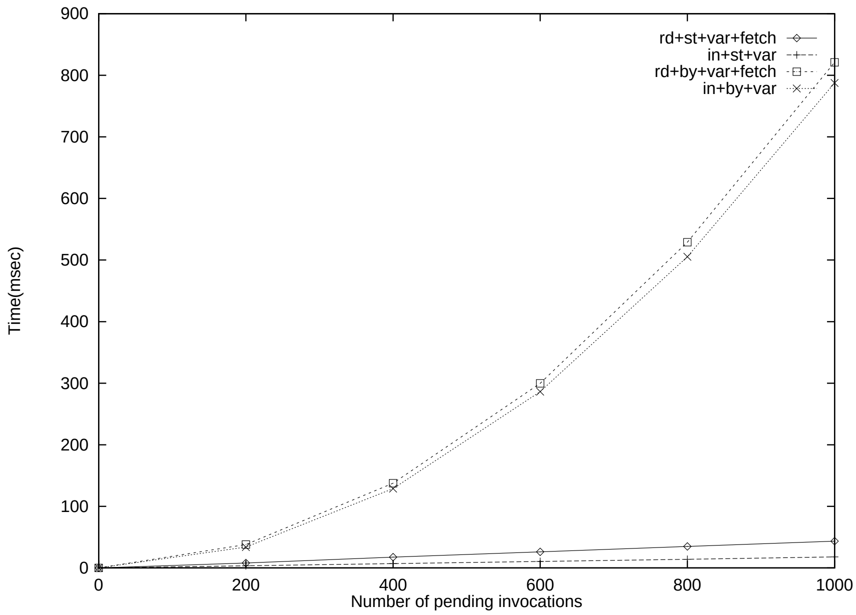

Figure 8.4: *Results from simulation using variant* **rd** *of* **in**.

Figure 8.4 shows the execution times for the four test programs with varying number of pending invocations. The performance of the simulation using variant **rd** of **in** is worse than that of the real **in**, as can be seen in the differences in the execution times for the input statements and their corresponding simulation using variant **rd**. Compared with the execution times for the input statement, rd+st+var+fetch takes 127–150% more time than in+st+var to service invocations; on the other hand, rd+by+var+fetch takes 5–15% more time than in+by+var.

The variant **rd** has a very costly implementation. The executing process scans the entire invocation queue once to build an examination queue before examination. On each iteration of the **rd**, it then searches the examination queue in the same way as the **in** does. For in+st+var and rd+st+var+fetch, the process searches the examination queue for the first invocation whose synchronization expression is true. For in+by+var and rd+by+var+fetch, the process searches the entire examination queue to select the invocation whose scheduling expression is the smallest. The difference between the execution times for an input statement and its simulation using variant **rd** to service an invocation is independent of the number of invocations pending for the operation. It can be attributed to the following three factors required for the simulation. First, the invocation queue is traversed once for building the examination queue. Second, the memory for an element is dynamically allocated when including each invocation in the examination queue; in addition, the memory is later released. Third, the simulation using **rd** and **fetch** is more expensive than the real **in** (see Section 8.2.4.1). Nevertheless, the difference is relatively small if the searching time for selecting an invocation is large, for example, when using a scheduling expression (5–15%).

8.2.5 Simulation of '?' Operator

A separate test employed two other test programs to compare the performance of the '?' operator and its simulation using **rd**. Each test program consists of a single SR process; it first generates a given number of invocations and iterates to service invocations, one at a time, until the number of pending invocations becomes zero (i.e., until all invocations have been serviced). To obtain the number of pending invocations, the SRR program uses an **rd** to read through the invocation queue to count the number of pending invocations. On the other hand, the SR program uses the '?' operator, which returns the number of pending invocations that is implicitly maintained by the RTS for each operation (see also Section 6.3).

Note: these two graphs use different scales on y-axis.

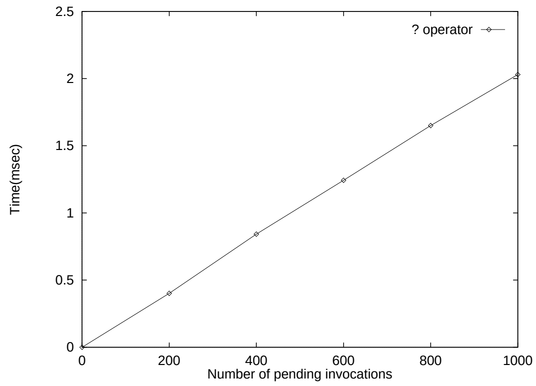

Figure 8.5: *Results from '?' operator.*

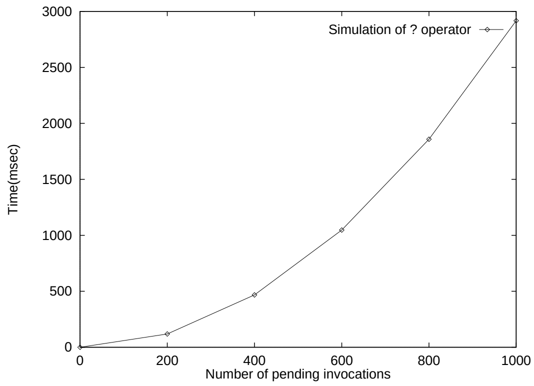

Figure 8.6: *Results from simulation using* **rd** *of '?' operator.*

Figures 8.5 and 8.6 show the performance of the '?' operator and its simulation using **rd** respectively. The times shown in these figures represent the cost of executing the '?' operator or its simulation using **rd** various number of times (i.e., same as the number of pending invocations). The cost of executing the '?' operator takes about 2μs, independent of the number of pending invocations. On the other hand, the cost of its simulation using **rd** increases as the number of pending invocations increases. Clearly, the simulation using **rd** is much more costly than the '?' operator. For example, if the number of pending invocations is 200, the time for executing the simulation 200 times is about 120 ms. Hence, the simulation takes about $120/200$ ms $= 600\mu$s, which is 300 times the execution time of the '?' operator. The difference in their execution times is linearly proportional to the number of pending invocations.

8.3 Macro-benchmarks

8.3.1 Problems Revisited

We measured the performance of the solutions to three problems discussed in Chapter 3. For each problem, the SRR solution using **rd** and the SR solution using **in** to these problems were developed and tested.

Test program	*Time(μs)*	*Cycles(in millions)*
SR multiple servicing processes (MSP)	13.543	390.649
SRR multiple servicing processes (MSP)	10.590	349.850
SR two-dimensional scheduler	11.042	351.714
SRR two-dimensional scheduler	6.187	173.622

Table 8.8: *Results from macro-benchmarks.*

Table 8.8 shows the execution times for the first two problems. The first problem is the multiple servicing processes problem presented in Section 3.1.2. Each test program consists of three processes accessing the same operation concurrently: an invoking process, an examining process, and a servicing process. The invoking process generates 150,000 invocations. The examining process repeatedly examines all pending invocations. The servicing process services all invocations and then terminates the program. We employ the critical section approach for the SR solution to this problem presented in Section 3.1.2. Note that we use the `nap` function to control the execution order of these processes so that the concurrent execution of these processes is deterministic and thus reproducible. Our results indicate that the SRR solution is simpler and more efficient than the SR solution. The SR solution takes 28% more time (12% more cycles) than the SRR solution. The difference in execution times is mainly due to the servicing and re-invoking of invocations in the examiner process (see Section 8.2.3) and also due to the additional cost of handling mutual exclusion via semaphore operations.

The second test problem is the two-dimensional scheduling problem in Section 3.2.3.

We employed the code in Figure 3.11 that preserves the arrival order of invocations for servicing invocations in the SR test program. We employed the SRR solution to this problem presented in Section 6.2.3.1 in the SRR test program. The results show that the SR solution takes about 78% more time (103% more cycles) than the SRR solution. As discussed in Section 3.2.3, the invocation queue is sorted explicitly by servicing all invocations and forwarding them to another operation. The sorting time (including the cost for servicing and re-invoking the invocations) accounts for the large difference between their execution times.

The third test problem is the LIFO scheduling problem discussed in Section 3.2.3. This test employed three test programs: one SR solution and two SRR solutions to this problem. The SR solution assigns a sequence number to each of the pending invocations using a scheduling expression in an input statement and then services the invocation with the highest sequence number (see Section 3.2.3). One of the SRR solutions uses the same approach as in the SR solution, but it simulates the input statement using **rd** and **fetch**. Another SRR solution examines all pending invocations via **rd**, and then mark the last one. It then services the marked invocation via **take** (see Section 5.2).

Each of the three test programs first generates a given number of invocations and then services them using the solution described above. Figure 8.7 shows the results from the three solutions to the LIFO scheduling problem with varying number of pending invocations. Executions of the two SRR solutions to this problem are slower than that of the SR solution. The SRR program using rd+fetch uses a variant **rd** (which has the most costly implementation). Therefore, the difference between the execution times for the SR program and the SRR program using rd+fetch, which is about 36% of the execution time for the SR program, is attributed to the same reasons as discussed for the simulation using a variant **rd** of an input statement in Section 8.2.4.3. The SRR program using rd+mark+take is much costly and takes about 270% more time than the SR program. One factor contributing to the large difference between their execution times is that this SRR test program explicitly scans the invocation queue via **rd** to reach the invocation

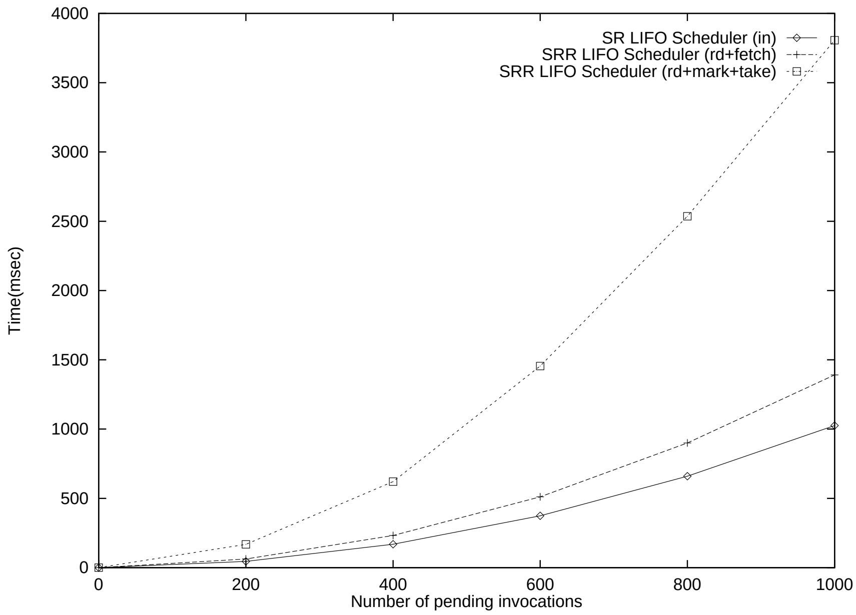

Figure 8.7: *Results from the three solutions to the LIFO scheduling problem.*

that arrives the queue the most recently. It therefore traverses the invocation queue and examines every pending invocation once when servicing an invocation. This queue traversal is very expensive due to additional procedure calls made to the RTS on examining each invocation. Another factor is due to the manipulation cost of the marked invocation list in the implementation of **mark** and **take** (see Section 8.2.4.1).

8.3.2 Elevator Controller Simulation

Section 6.5 described several solutions to the elevator controller simulation problem. In particular, one SR solution and three SRR solutions were given. This section presents the execution costs for these four different solutions. The four programs are:

ELEV_SR: the SR solution employs an additional operation `wait_geton`; i.e., the elevator controller process from Figure 6.10 using procedure `redraw` from Figure 6.11 that examines `wait_geton` using **in**.

VIEW_RD: the SRR solution employs an additional operation `wait_geton` but examines it using **rd**; i.e., the elevator controller process from Figure 6.10 using procedure `redraw` that examines `wait_geton` using **rd** shown in Section 6.5.2.

COMBINED: the SRR solution does not employ an additional operation and it examines `get_on` using **rd** and services invocations using **in**; i.e., the elevator controller process from Figure 6.12.

ELEV_RD: the SRR solution does not employ an additional operation and it examines `get_on` using **rd** and services invocations using **rd** and **fetch**; i.e., the elevator controller process from Figure 6.13.

Each of the four programs was run with various values for three of its parameters: the number of person processes, the number of floors in the building, and the elevator capacity. All of these three parameters affect the number of persons waiting to board the elevator. Figure 8.8 shows the results from the elevator controller simulation on a 5-story building with an elevator capacity of 5 persons. The number of person processes varies from 10 to 120. We also ran the simulation with different numbers of floors in the building and with different elevator capacities. The results obtained follow the same relative order as those reported in Figure 8.8.

The execution times are increasing in the order VIEW_RD, COMBINED, ELEV_SR, ELEV_RD. The differences vary with the number of person processes. Compared with the execution times for ELEV_SR, execution of VIEW_RD is faster by 1–10% and that of COMBINED is faster by 1–4%. On the other hand, execution of ELEV_SR is slower than that of ELEV_SR by 1–6%.

Clearly, execution of VIEW_RD is faster than that of ELEV_SR because the cost of **rd** for examining invocations of `wait_geton` is much less than that of using **in** in ELEV_SR (see Section 8.2.3). The **rd** used in VIEW_RD is an unordered invariant **rd**, i.e., an optimized **rd**. COMBINED contains two **rd** statements. One **rd** is an unordered invariant **rd** and the other is a variant **rd** whose synchronization expression references a variable. As

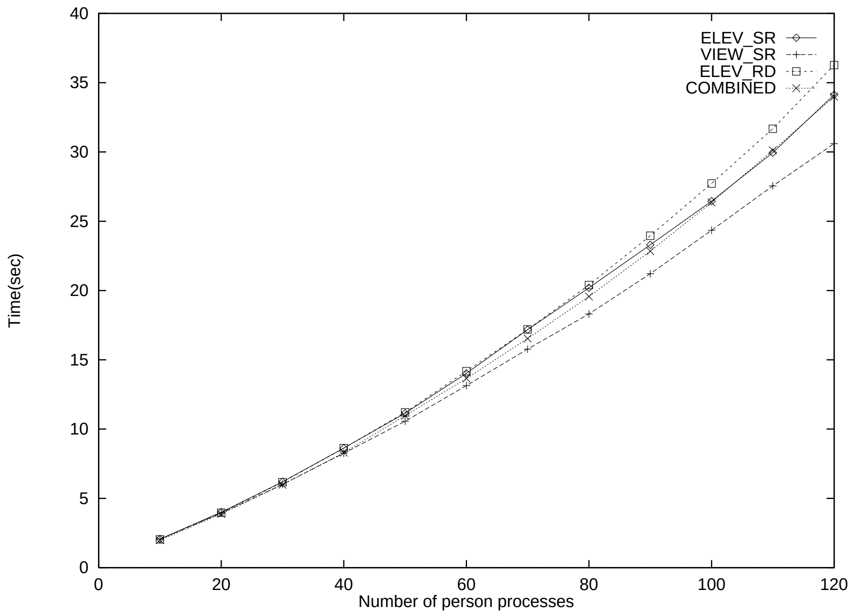

Figure 8.8: *Results from elevator controller simulation.*

described in Section 9.2.2, this variant **rd** could also be optimized if the compiler employs more sophisticated static analysis to identify the optimizable cases. Similarly, ELEV_RD contains three variant **rd** statements, in which two of them are also optimizable. Although an unoptimized **rd** is used in COMBINED, its execution is still faster than the execution of ELEV_SR, in which invocations of `wait_geton` and `get_on` are actually serviced and re-invoked when they are being examined. ELEV_RD is the slowest among the four test programs for three reasons. First, three unoptimized **rd** statements are used in ELEV_RD. The RTS must allocate and later release memory for elements in the examination queues. Second, the operations `get_on` and `get_off` appear on one of the three **rd** and are thus in the same class. All invocations in the class are therefore included in each examination queue (see Section 7.3.1). Although the other two **rd** statements examine only invocations of `get_on`, they might access invocations of `get_off` while searching for an invocation of `get_on` to examine. These two **rd** statements are thus less efficient than the one used in COMBINED for examining invocations of `get_on`. Third, the controller process in ELEV_RD services invocations using **rd** and **fetch**, which is less efficient than **in** (see

Section 8.2.4).

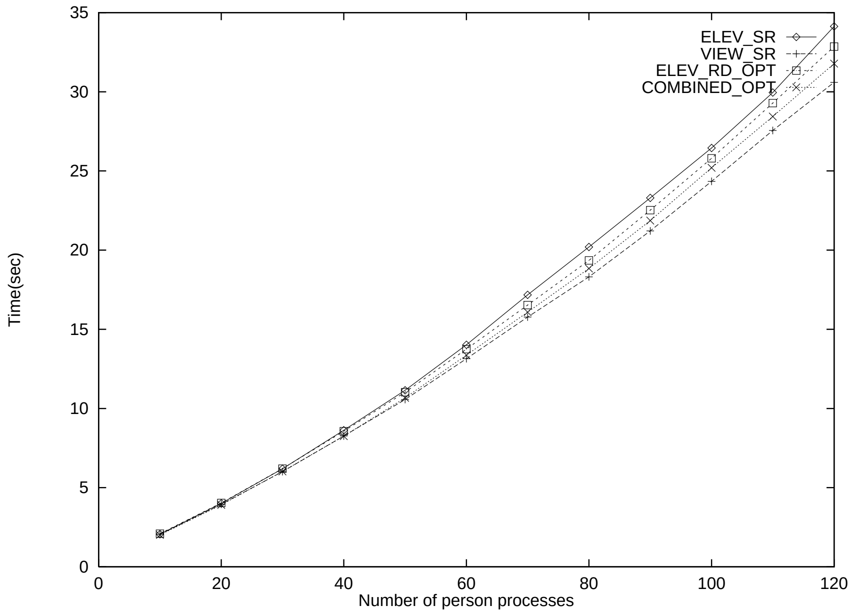

Figure 8.9: *Results from elevator controller simulation.*

To see how the optimizations of **rd** affect the performance of these test programs, we hand-optimized the optimizable **rd** statements in COMBINED and ELEV_RD so that the implementation of these **rd** statements do not use an examination queue. We then ran these optimized programs and measured their execution times. We denote the optimized COMBINED as COMBINED_OPT and the optimized ELEV_RD as ELEV_RD_OPT. Figure 8.9 shows the results from the two optimized programs running with the same input parameters as in Figure 8.8. The execution times for ELEV_SR and VIEW_RD are also shown in this figure for comparison. Compared with the execution times for ELEV_SR, execution of COMBINED_OPT takes 1–7% less time while that of ELEV_RD_OPT takes 1–4% less time. The results illustrate that the optimizations of **rd** improve the performance of both COMBINED and ELEV_RD, reflecting the high cost for building the examination queues. Executions of the three SRR solutions are faster than that of ELEV_SR.

8.4 Other Architectures

The timing tests were performed on four different machines listed in Table 8.1. To get some feel about the relative computational speed of these machines, the execution times for performing one 'add" operation on these machines were measured. Table 8.9 shows the cost of one "add" operation on the four machines and the ratios of their execution times to that on on the Alpha. We used two C programs to measure such cost. The first program

Machine	*Time(ns)*	*Ratio*
DEC Alpha 3000/400	7.639	1.00
DECstation 260	8.410	1.10
SGI Indigo	10.065	1.32
Sun SPARCstation 5	14.450	1.89

Table 8.9: *Execution times for one "add" operation on different machines.*

contains a loop of 1 billion iterations, each of which performs one "add" operation; the second program is similar to the first one, except that two "add" operations are performed on each iteration. We then determined the execution time for an "add" operation from the difference in the execution times for these two programs on each of the four machines.[1]

Table 8.10 shows some representative results obtained from the four machines and the differences between these results and those obtained from the Alpha, i.e., the data presented in the previous sections. (The complete results obtained from the four machines are given in Appendix B.) The first column lists the test programs. The second column shows the parameters used when the test program ran. Each test program contains two input parameters: the number of pending invocations and the number of iterations of the outer loop. The resource body of each test program is enclosed with an outer loop. We use the outer loop to make the test program run long enough to get reliable statistics. The last four columns report the execution time, in seconds, for the test programs run on four different machines. They also report the difference between the execution time for the test program run on one machine and that on the Alpha.

[1]These two programs were compiled with -O flag that invokes the ucode optimizer. We inspected the generated code to ensure that the same kind of loop is used in both programs.

Test program	Invs	Iter	Alpha 3000/400	DECstation 260		SGI Indigo		Sun SPARC 5	
rd+mark+take	75000	3	12.532	12.858	(3%)	17.781	(42%)	19.83	(58%)
rd+fetch	75000	4	9.490	12.263	(29%)	14.127	(49%)	12.62	(33%)
in	75000	6	9.772	13.678	(40%)	16.631	(70%)	13.11	(34%)
in+st (SEQ)	75000	6	10.313	13.444	(30%)	15.664	(52%)	12.98	(26%)
in+st (ALT)	1000	80	12.342	17.295	(40%)	19.411	(57%)	28.98	(135%)
rd+st+fetch (SEQ)	75000	4	9.700	12.338	(27%)	15.326	(58%)	12.72	(31%)
rd+st+fetch (ALT)	75000	4	10.149	13.115	(29%)	15.425	(52%)	1609.15	(16K%)
SR MSP	150000	1	13.543	8.651	(-36%)	19.658	(45%)	38.64	(185%)
SRR MSP	150000	1	10.590	7.745	(-27%)	14.269	(35%)	35.33	(234%)

Table 8.10: *Representative results obtained from different machines.*

From these statistics, we observed that the differences in the execution times are very inconsistent on the DECstation and the SPARC 5. We also observed several seemingly impossible results. For example, executions of some test programs are faster on the DECstation than on the Alpha. Moreover, compared with the results obtained on the Alpha, execution of rd+st+fetch (ALT) on the SPARC is 160 times slower, but executions of other test programs are only 0.3–0.6 times slower. When the test programs were run with different parameter values, the results and the differences changed considerably. Two reasonable explanations of those seemingly impossible results are different cost of context switching between SR processes and different cost of memory allocation and deallocation on different machines.

8.4.1 Context Switching

One explanation to the differences in the test programs that involve several processes is that the cost of context switching between SR processes is different on different machines. Table 8.11 shows the execution time for a context switch performed on the four machines.

Machine	*Time(μs)*
DEC Alpha 3000/400	0.879
DECstation 260	0.753
SGI Indigo	0.890
Sun SPARCstation 5	19.640

Table 8.11: *Cost of context switching in different machines.*

As the test programs for the multiple servicing processes problem and the elevator controller simulation perform a lot of context switching (SR MSP and SRR MSP in Table 8.10), the cost of context switching dominates the overall cost of the test programs.

8.4.2 Memory Allocation

Another factor affecting the results is the different memory allocation behavior in each kind of machine. A separate C program was developed to determine the behavior of memory allocation and deallocation in different machines. This C program basically builds a doubly-linked list and then traverses the list a given number of times; finally, it removes all elements and frees the allocated memory. This C program also has an outer loop to make the program run multiple times so that memory allocation and deallocation are performed repeatedly. Table 8.12 shows the execution times for this C program with various values of its parameters: the number of blocks (elements) in the linked list, the number of iterations of the outer loop, the block size, and the number of times to traverse the list.

Blocks	Iter	Block size	Times traverse	Alpha 3000/400	DECstation 260		SGI Indigo		Sun SPARC 5	
1000	1000	84	0	3.077	1.679	(-45%)	7.837	(155%)	7.37	(140%)
1000	1000	144	0	2.868	2.636	(-8%)	7.806	(172%)	7.24	(152%)
100	10000	84	0	2.474	1.303	(-47%)	7.296	(195%)	6.34	(156%)
100	10000	144	0	2.451	1.588	(-35%)	7.446	(204%)	6.69	(173%)
10000	100	84	0	3.680	4.995	(36%)	7.969	(117%)	7.43	(102%)
10000	100	144	0	3.964	6.166	(56%)	9.829	(148%)	7.38	(86%)
100	1	84	100000	1.059	1.398	(32%)	1.919	(81%)	1.72	(62%)
100	1	144	100000	1.099	1.987	(81%)	1.912	(74%)	1.70	(55%)
50000	1	84	10	0.496	1.686	(240%)	1.361	(174%)	1.15	(132%)
50000	2	84	10	0.904	2.659	(194%)	2.462	(172%)	1.66	(84%)
50000	3	84	10	1.313	3.674	(180%)	3.582	(173%)	2.21	(68%)
50000	10	84	10	4.279	10.576	(147%)	11.319	(165%)	5.87	(37%)
50000	20	84	10	8.399	20.536	(145%)	22.378	(166%)	11.13	(33%)

Table 8.12: *Memory allocation behavior with different combination of parameter values.*

Memory allocation and deallocation perform differently on different machines. We observed that the system call (*sbrk*) to change the amount of space allocated for a process is relatively more efficient on the Alpha than other machines. However, memory allocation performs relatively slower on the Alpha than the DECstation when some space was allocated but has previously been freed. In other words, memory allocation on the Alpha uses a less efficient algorithm to search for the freed memory to reuse. In addition, the block size is also another factor affecting the performance of memory allocation. The size of an invocation block with no parameters in the SRR implementation on the Alpha is 144 bytes, whereas that on the other three machines is 84 bytes. The size of an invocation block on the Alpha is larger because the size of a pointer as well as that of a long integer on the Alpha is 64 bits whereas that on the other three machines is 32 bits.

To identify the reasons why execution of the test program rd+st+fetch on the SPARC is extremely slow to service the invocation sequence ALT (which contains invocations with alternating positive and negative parameter values), another C program was also developed to simulate that test program. Table 8.13 summarizes the execution times for this C program with various values of its parameters. The execution times on the SPARC

Blocks	Iter	Block size	Alpha 3000/400	DECstation 260		SGI Indigo		Sun SPARC 5	
10000	1	72	0.070	0.218	(211%)	0.171	(144%)	0.41	(486%)
10000	1	84	0.076	0.218	(187%)	0.195	(157%)	7.36	(9584%)
10000	1	120	0.090	0.218	(142%)	0.225	(150%)	7.36	(8078%)
10000	1	144	0.091	0.347	(281%)	0.248	(173%)	0.58	(537%)

Table 8.13: *Results from C programs simulating* in+st *and* rd+st+fetch *to service ALT.*

vary dramatically with different block sizes. In particular, for the block size of 120 and 144 bytes, the program ran on the SPARC almost 100 times slower than it ran on the Alpha. It was determined that most of the execution time was spent on memory deallocation. The cost for deallocating memory that was used as a block at the beginning or at the end of the list is reasonable. However, memory deallocation is extremely expensive to free a block in the middle of the list.

From these observations, the execution times for the test programs are highly dependent of the time spent on allocating new space for the executing process and the time spent on searching for the deallocated space to reuse. We believe that the different behavior of memory allocation and deallocation on different machines can directly explain the strange patterns observed from the statistics.

8.5 Summary

We ran the timing tests using a number of benchmark programs (listed in Appendix B with the complete statistics). The performance results obtained from the DEC Alpha 3000/400 is summarized below. Table 8.14 shows some representative results for invocation examination as well as invocation generation and service using simple input statements in SR and using simple **rd** statements in SRR. The last column gives the ratio of the

Mechanism	SRR		SR		SRR to SR Ratio	
	Time	*(Cycles)*	*Time*	*(Cycles)*		
Examination of an invocation	9.743	(325)	24.644	(675)	39%	(48%)
Invocation+service	31.633	(914)	21.716	(658)	146%	(139%)

Table 8.14: *Summary of performance results.*

execution time (the execution cycles) for the mechanism using **rd** in SRR to that using **in** in SR. Compared with the execution times (and the execution cycles) for the input statement in SR, a simple **rd** in SRR only takes 0.39 (0.48) time as long to examine an invocation. On the other hand, the simulation using **rd** and **fetch** in SRR takes 1.46 (1.39) times as long to generate and service an invocation.

In fact, the optimizations of an **rd** that contains a synchronization or scheduling expression can greatly improve its performance. Our results indicate that the simulation using **rd** of an input statement can outperform the real input statement in SR in some cases when a synchronization or scheduling expression is used. Table 8.15 gives a range of ratios of the execution times for the simulation using **rd** (optimized and unoptimized) in SRR to those for the input statement in SR.

Several realistic applications were rewritten using the new mechanisms in SRR. Our results show that the overall performance of some applications can be improved by 1–44%. However, the new mechanisms can slow executions of some applications down by 36–270%.

Invocation+Service	*Simulation using* **rd**	*SRR to SR Ratio*
Synchronization expression	optimized **rd**	14–141%
	unoptimized **rd**	227–249%
Scheduling expression	optimized **rd**	6–54%
	unoptimized **rd**	105–115%

Table 8.15: *Ratios of the cost for simulation of* **rd** *in SRR to that for* **in** *in SR.*

We also conducted the timing tests on three other architectures, including a DECstation 260, a SGI Indigo, and a Sun SPARC 5. The results obtained from these machines deviate considerably from those obtained from the DEC Alpha in some cases. After additional testing, it was determined that the relative cost of memory allocation and deallocation on each kind of machine varies depending on whether memory that has been freed is available, thus explaining those deviations and the seemingly impossible results. Different cost of context switching is also another factor affecting the results obtained from different machines.

Benchmarking is difficult in general. Without exception, we faced a lot of difficulties in performing our benchmarking on various platforms. It is not easy to find good benchmarks to evaluate the performance of the new mechanisms on various platforms. Nevertheless, the results for **rd** we obtained are reasonable.

Chapter 9

Discussion

SR is a powerful and flexible language for concurrent programming. Chapter 3 discussed the limitations of invocation handling in SR and illustrated how those limitations make it difficult to write programs in the language. Chapter 5 presented the new mechanisms in SRR that provide additional support for invocation handling to overcome these difficulties and Chapter 7 described their implementation.

This chapter first explores various design alternatives and justifies our decisions. This chapter then describes several shortcomings of the current implementation and identifies their possible optimizations. Some of these optimizations of **rd** are also applicable to some SR's existing mechanisms. This chapter discusses how those optimizations can be applied to **forward** and **in** statements. It also discusses other issues such as possible side effects of **rd** and modification of invocation parameters in **rd**. Finally, this chapter discusses the applicability of our work to other languages.

9.1 Design Alternatives

This section discusses various design alternatives for the following:

- **General approaches:** We considered two approaches to making invocation handling more flexible — *statement-based* approach and *type-based* approach. We discuss the advantages as well as disadvantages of these approaches and discuss why we chose the former and decided against the latter.

- **Semantics of rd:** The **rd** statement is an iterator that allows examination of invocations. In addition, it has an optional else command with special semantics. We explore other reasonable semantics of **rd** and justify why we chose the particular semantics.

- **Class locking:** Locking is necessary for preventing concurrent access to an operation by several processes. We discuss when to acquire and release mutually exclusive access to an operation for examination of invocations via **rd**.

- **Invocation parameters:** As the current semantics of **rd** releases access to a class before examining an invocation, we explore several ways to avoid modification of the parameters of the invocation that is being examined via **rd**.

9.1.1 General Approaches

As discussed in Chapter 3, existing SR mechanisms do not permit straightforward solutions to a number of problems. As seen there and in Chapter 8, solutions can be programmed using existing mechanisms, however these solutions are cumbersome and inefficient. We considered two approaches to making invocation handling more flexible. As seen in Chapters 5 and 6, we employ the *statement-based* approach to use several new statements to deal with invocations — **rd**, **mark**, **take**, and **fetch**. The **rd** statement allows an invocation to be read without actually removing the invocation from the invocation queue. The **mark** and **take** statements allow invocations to be marked and later serviced. In addition, the **fetch** statement is a useful abbreviation for **mark** and **take** statements within an **rd**. These statements are similar in appearance to the existing mechanisms in SR. Most importantly, they have simple meanings and thus should be easy to learn and use.

Another approach we considered was the *type-based* approach, which employs a new type as well as some special operations on this type. This type is essentially a pointer to an invocation and the operations provide means to manipulate invocations. Below we present this approach and discuss why we decided against it. Note that the semantics and syntaxes of the primitives for this approach are tentative and have not been fully defined.

The type-based approach employs a new type, `inv`, to represent invocations as well as some special operations on this type. Variables of type `inv` can point to invocations that match their declared parameterization. Various primitives would apply to variables of this type, such as `getinv`, `grabinv`, `qlock`, and `qunlock`. The `getinv(f)` primitive returns a pointer to the next invocation of the specified operation `f`. The `grabinv(r)` primitive removes the invocation block pointed at by `r` from the invocation queue and returns it to the calling process. The `qlock` primitive locks the invocation queue associated with the specified operation whereas the `qunlock` primitive releases the lock.

The following code, for example, prints out all pending invocations of operation `f`, as in the code in Figure 3.1, but without disturbing their order.[1]

```
var r: inv f
do (r := getinv(f)) != null -> # set r to point at next invocation of f
   write( r.x ) # print out parameters of invocation
od
```

As another example, consider how to locate and service the invocation of `batch` with a zero `uid`. The following code is an initial attempt.

```
var r: inv batch
do (r := getinv(batch)) != null and r.uid != 0 ->
   # do nothing
od
if r != null ->
   # service invocation
   # actually remove the invocation from the batch's invocation queue.
   grabinv(r)
      # actually service it and possibly send back reply
      ...
fi
```

Note that the above code is equivalent to the non-blocking **in**

[1] Similar to the generated code in Chapter 7, we use the notation `r.x` to represent the parameter `x` in the invocation pointed at by `r`, which is a variable of type `inv`.

```
in batch(uid) st uid = 0 ->
   # actually service it and possibly send back reply
     ...
[] else ->
   # do nothing
ni
```

For the `inv` code to be blocking (i.e., equivalent to the **in** code without the **else**), a "delay until next invocation arrives" primitive is needed; that primitive should allow waiting for an invocation of one of several operations.

Both of the above examples using the `inv` type require locking the invocation queue explicitly. Otherwise, race conditions with another process accessing the queue are possible. In the two examples above, a `qlock(f)` and `qunlock(f)` pair could surround each code fragment, ensuring that the invocation queue does not change during execution of the fragment. Alternatively, finer grained control, closer to **in**'s semantics, could be obtained by locking just around each `getinv` and `grabinv`.

These examples illustrate some key points about this approach. Clearly, it is lower-level than the statement-based approach or the current SR approach. This approach represents an "opening up" of the current SR implementation. It has more direct mapping of the language primitives to their implementation, thus offering better efficiency. As seen above, the new language primitives are somewhat like the RTS primitives invoked by the MC (see Section 7.1.3). However, we decided against this approach for the following reasons. First, as seen in the above examples, the `inv` code is harder to understand and maintain than the code using the statement-based approach. The `inv` code looks similar to the generated code for the statement-based approach (see Section 7.5).

Second, programs written using this approach are more error-prone. The programmer has to obtain mutually exclusive access to operations before examining an invocation queue using the `qlock` and `qunlock` primitives. Race conditions with another process accessing an invocation queue would occur if the user forgets to obtain the lock before accessing an operation. Similarly, deadlock would also happen if the user forgets to release the lock after accessing an operation.

So far, we have considered examples using only one operation. Using this approach, it would be very difficult to examine invocations of more than one operation. To permit examining invocations of several operations in their arrival order, we could change the `getinv` primitive (or add a new primitive) to get an invocation of a class of operations rather than of an operation. Currently, programmers need to know that invocations of operations appearing in an input statement are queued implicitly in their arrival order in a class. This new approach, however, would require programmers to know exactly to which class an operation belongs. In addition, they would need to know that the class may contain invocations that are pending for operations other than those appearing in an input statement. This would further complicate the language. Furthermore, several other semantic problems can still occur. Suppose `f` and `g` are two operations in a class. It would be difficult for the compiler to perform type-checking on variables of type `inv` because the operation of the invocation returned from the primitive might not be known until runtime. The programmer would also need a way to determine the operation for which the invocation pointed at by a variable of type `inv` is pending. If `f` and `g` have different parameterizations, it would also be difficult to ensure that the MC accesses the invocation parameters of the right operation of an invocation block. For example, if the MC accesses invocation parameters of `g` from an invocation block of `f`, it would introduce a potential source of error.

9.1.2 Iterative Nature of Rd

The **rd** statement is an iterator that reads all pending invocations, one per iteration, for operations appearing in the **rd**. It treats the collection of invocations as a group rather than individually, as **in** does.

We considered a non-iterative **rd** statement, denoted here as **ni_rd** (pronounced as "nerd"), that reads only one invocation of the operations appearing in the **ni_rd**. More specifically, it reads the oldest pending invocation that has not been examined by the executing process. An implicit pointer is set to indicate how far along the invocation queue the process has examined; it is advanced to the next invocation of the specified operation on each execution of **ni_rd**.

We chose not to use this non-iterative semantics mainly due to its significant loss of functionality. For example, consider the following code that prints out all pending invocations of operation f, as in the code in Figure 3.1.

```
do true ->
   ni_rd f(x) -> # non-iterative rd
      write(x)
   [] else ->
      exit
   dr_in
od
```

The first execution of the above code has the same effect as the iterative **rd**

```
rd f(x) ->
   write(x)
[] else ->
   exit
dr
```

When the first execution of the **ni_rd** code terminates via **exit** in the else arm, all pending invocation of f have been examined. Therefore, the implicit pointer points to the end of the invocation queue. Figure 9.1 illustrates the effect of execution of the **ni_rd** code. Figure 9.1(a) shows two pending invocations of f. These invocations have not been

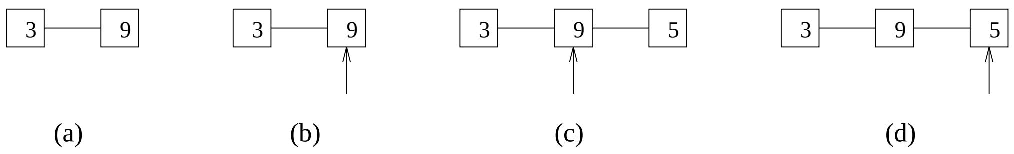

Figure 9.1: *Examining operation* f *using* **ni_rd** *(a) Pending invocations of* f *(b) first execution of the* **ni_rd** *code (c) invocation* f(5) *arrives (d) second execution of the* **ni_rd** *code.*

examined by the process before executing the **ni_rd** code. Each iteration of the loop (i.e., each execution of the **ni_rd**) reads one pending invocation of f. The first execution of the code thus examines invocations `f(3)` and `f(9)` and outputs 3 and 9. After the code terminates, the implicit pointer points to `f(9)` in the invocation queue as shown in Figure 9.1(b). The next execution of the **ni_rd** code examines only invocations that arrive after the last execution of **ni_rd** is completed; i.e., those invocations follow the invocation pointed at by the implicit pointer. Suppose a new invocation `f(5)` arrives as shown in Figure 9.1(c). The second execution of the **ni_rd** code then reads the invocation `f(5)` only and prints 5. As a result, each invocation in the invocation queue is examined only *once* by each process via **ni_rd**. In contrast to the **rd** statement, the RTS does not keep any information about the previous execution of the **rd**. Every execution of the **rd** examines all pending invocations of `f`, including those invocations that have been examined in its previous execution (if they have not been serviced). Therefore, using the same example in Figure 9.1, the first execution of the **rd** prints 3 and 9 and the second execution prints 3, 9, and 5.

We also considered providing a new statement, `first`, whose semantics specify re-examination of an operation's invocation queue from the beginning. More precisely, it resets the implicit pointer in a process so that the next execution of **ni_rd** would examine the first pending invocation of the specified operation. For example, the following code is equivalent to the above **rd** statement after adding a `first` statement in the **ni_rd** code.

```
first f
do true ->
   ni_rd f(x) -> # non-iterative rd
      write(x)
   [] else ->
      exit
   dr_in
od
```

However, even with the `first` statement, some of the examples discussed in Chapter 3 could not be solved easily, such as the threshold order peeking in Section 3.1.3 and the two-dimensional scheduling in Section 3.2.3. Moreover, a number of semantic issues arise when considering the `first` statement. Three main problems are discussed below.

First, a nested **rd** could not be simulated using **ni_rd** without using additional operations. (Examples in Section 6.2.3 showed that the nested **rd** is useful in solving some scheduling problems.) Consider, for example, the following nested **rd**, which prints all pending invocations of `f` and for each of them also prints all invocations whose parameters are smaller than its parameter value.

```
rd f(x) ->
   write(x)
   rd f(y) ->
      if x > y ->
         write(y)
      fi
   [] else ->
      exit
   dr
[] else ->
   exit
dr
```

The above code could not be simulated easily using **ni_rd**. In particular, note that the following code is *not* equivalent to the above nested **rd**.

```
do true ->
   ni_rd f(x) ->
      write(x)
      first f  # reset to examine the first pending invocation
      do true ->
         ni_rd f(y) ->
            if x > y ->
               write(y)
            fi
         [] else ->
            exit
         dr_in
      od
   [] else ->
      exit
   dr_in
od
```

The difficulty is that one implicit pointer is maintained for `f` in the process executing this code. Once `first f` is executed, the implicit pointer is reset so that the **do** loop following it in the above code examines all pending invocations of `f`. After the inner **do** terminates, the outer **ni_rd** finds no invocation and then exits the outer loop as the inner **ni_rd** has already reached the end of the invocation queue. Adding a `first` statement following the loop would only reset the pointer to the first invocation, thus making the outer **ni_rd** always examine the first invocation.

Second, it is hard to decide whether **ni_rd** can have synchronization and scheduling expressions. As discussed in Section 7.2, these expressions can be used to alter the order in which an **rd** examines invocations. In the above code, we assumed that synchronization and scheduling expressions could not be used. If we allow **ni_rd** to contain synchronization and scheduling expressions, the implementation cost of **ni_rd** would be very expensive. A single implicit pointer is not sufficient when implementing the **ni_rd** containing synchronization and scheduling expressions. The **ni_rd** could be implemented by maintaining a list of invocations that have been examined in each process so that execution of the **ni_rd** would be guaranteed to examine an unexamined invocation. Note that this list is different from an examination queue discussed in Chapter 7, which instead contains pointers to invocations that have not been examined. The **ni_rd** could also be implemented using the examination queue. Owing to the non-iterative nature of the **ni_rd**, the cost of building an examination queue would seem to be wasteful in many cases. In addition, optimizations of the **rd** could not be used for optimizing **ni_rd** easily. Because different **ni_rd** statements could have different synchronization and scheduling expressions associated with the same operations, it would be very difficult to analyze a program to determine the optimizable cases. On the contrary, if we do not allow **ni_rd** to contain synchronization and scheduling expressions, the expressive power of that statement would be reduced and the programmer has to use other means to achieve similar effects obtained from these expressions (as seen in the above example).

Third, the semantics of `first` and **ni_rd** are not well-defined when the specified operation is not the only operation in its class. For example, suppose `f` and `g` are two operations in a class and some invocations of `f` and `g` have been examined. Consider the following code that contains a `first` statement followed by a two-armed **ni_rd**.

```
first f
ni_rd f(x) ->
   ...
[] g(y,z) ->
   ...
dr_in
```

One semantic issue is whether the `first` should reset the implicit pointer to the first invocation of `f` or whether it should reset the implicit pointer to the first invocation in the class (i.e., has effect on both `f` and `g`). Furthermore, it is very difficult to implement a multi-armed **ni_rd** such as the above **ni_rd**. The RTS could maintain one implicit pointer in each process for each operation but the RTS would still have difficulty determining which invocation to examine next.

9.1.3 Blocking Else Command of Rd

We allow the last arm of an **rd** to be an else command. Unlike the **in** statement, the **rd** does not terminate after executing the else command unless it executes a statement, such as **exit**, that terminates an iterative statement. The executing process blocks immediately after executing the else command and wakes up when new invocations arrive. This blocking behavior allows the programmer to change the selection criteria of an **rd** in its else command without terminating the **rd**. Otherwise, invocations that have not been read in the previous iterations would not be re-examined again, even if they satisfy the modified selection criteria.

To illustrate, consider the following **rd** statement:

```
var n := 10
rd f(x) st x > n ->
   ...
[] else ->
   n := 0
dr
```

This **rd** first examines invocations of `f` for which parameter `x` is greater than 10. When all such invocations are examined, the process executes the statement `n := 0` and then blocks waiting for a new invocation to arrive. When a new invocation arrives, the process wakes up and then examines invocations of `f` for which parameter `x` is positive. Although the parameter values of the examined invocations are also positive, the process examines only those that have not been examined. If the **rd** terminated immediately after executing the else command, additional operations would have been used to separate invocations that have been examined from those that have not. The threshold order peeking example in Section 5.1.4 illustrates the use of this behavior. Furthermore, this blocking behavior avoids busy waiting for a selectable invocation to arrive.

Nevertheless, one problem associated with the above example is that the process might block forever. After the else command has been executed, the process blocks waiting for new invocations to arrive. If no invocations arrive, the process would wait forever even if some pending invocations in the invocation queue are selectable for examination. This problem can be solved by using the **next** statement. Recall from Section 5.1.4 that the **next** statement forces the **rd** to continue execution of the next iteration instead of blocking the process.

We also considered two other alternatives in solving the problem described above. First, we could change the semantics of **rd** so that it checks whether invocations that have not been examined in the invocation queue are selectable for examination before blocking (as execution of the else command might change the selection criteria). If such invocations are pending, the process continues execution of the next iteration of **rd**. If there are no such invocations, the process blocks waiting for new invocations to arrive. The implementation of this modified semantics would be rather inefficient, though, because the process would need to inspect all unread invocations every time the else command is executed.

The second alternative we considered was to use a new statement (e.g., `retry`) to force the executing process to continue execution of the next iteration of **rd** without blocking. This gives the programmer explicit control over when to block the process and when not to. The `retry` statement differs from the **next** statement in that the else command of an **rd** would be executed even if no invocation is examined in the following iteration of the **rd** after `retry` has been executed; thus, the executing process would busy wait for an invocation to arrive. In contrast to the case when a **next** statement is executed in the else command of an **rd**, the process blocks immediately without executing the else command again if no invocation is examined in the following iteration of the **rd**. However, it is not clear whether this semantics is useful in real applications.

9.1.4 Class Locking

As described in Section 7.2, a process must obtain access to a class before examination via **rd**. Two approaches to locking a class were considered. The first approach is to lock before examining a class and to release the lock when the **rd** terminates. This is the simplest approach and would have minimal impact on the current RTS implementation. The RTS is certain that the invocation queue would not be modified over execution of the **rd**. However, this approach has several disadvantages. First, this approach does not allow **rd** nested within other statements, such as **in** and **rd**, that access the same class. Otherwise, the executing process would deadlock. Second, programmers need to understand the concept of classes and the implementation details, which makes the language less expressive and harder to learn and use. For example, suppose `f` and `g` belong to the same class. Consider two processes each executing an **rd** statement; one examines `f` and the other examines `g`. As these two processes need to acquire access to the same class, they are force to execute sequentially; thus reducing the level of concurrency. Moreover, if the executing **rd** does not terminate, the other process would wait forever.

The second approach is to lock the class at the beginning of each iteration (i.e., before searching for an invocation to examine) and to release the lock before executing the command body associated with the corresponding operation. We chose this approach because it allows greater expressive power. The **rd** statement can be nested with any **in** or **rd**. As seen in Chapter 6, it is useful for many applications to have servicing statements executed within an **rd**. However, as seen in Section 7.2, this locking mechanism greatly complicates the implementation. In particular, manipulating the invocation queue is more complicated because the queue can be modified while a process is accessing it. The RTS must also handle several processes accessing the same invocation at about the same time.

9.1.5 Invocation Block Copying

Recall from Section 5.1 that the **rd** allows examination of invocations. To avoid modification of invocation parameters within **rd**, we could define a rule prohibiting such modification. In other words, the invocation parameters would not be treated as local variables within a command body of an **rd**. However, with such a restriction, it is not safe for an **rd** to reference the invocation parameters directly from the actual invocation block. Consider, for example, a process that executes an **in** statement to service the invocation that is currently being examined by another process. As the servicing process can modify the invocation parameters in the invocation block, the examining process is not guaranteed to get the same parameter values while examining the same invocation block.

We chose to examine a copy of an invocation block rather than to access the actual invocation block. The programmer can treat the invocation parameters within a command body of an **rd** as local variables, as in an **in** statement. On each iteration of an **rd**, the RTS must allocate (and later release) memory for the copy of the invocation block and return the copied invocation block to the executing process for examination. This method has one disadvantage, though: it is quite inefficient because of the cost of dynamic memory allocation and deallocation for an invocation block on each iteration of the **rd** (see Section 8.2.1). The implementation could be improved to reduce its cost. The compiler could analyze the program to determine whether any servicing code or examining code would modify any of the invocation parameter values. If there is no such code, the RTS returns the actual invocation block to the examining process. This requirement is relatively easy to check.

Another alternative is to allow modification of invocation parameters, which will be discussed in Section 9.4. We expect that this semantics would be useful.

9.2 Shortcomings and Optimizations

9.2.1 Semaphore Optimization

The SR compiler optimizes special uses of operations. One such optimization is that operations that are used like semaphores are implemented directly by the RTS's semaphores rather than by the general operation mechanisms (see [3, 28] for details). In the SRR implementation, we further constrain the criteria that an operation must satisfy to apply this optimization. An operation is semaphore optimizable if it is invoked using **send** and has no parameters or return value. Additionally, it must be serviced by an input statement that has no synchronization or scheduling expressions and that services no other operations. In addition to these current criteria for a semaphore optimizable operation, the operation must not appear in an **rd** statement. This restriction is, however, not too significant because realistic programs, in our experience thus far, rarely contain **rd** statements that examine a parameterless operation (which, as noted above, is one of the criteria that an operation must satisfy to be implemented by semaphores).

We could modify this constraint to extend the range of semaphore optimizable cases. If an operation satisfies all the main criteria and it is also examined by an **rd** that has no synchronization or scheduling expressions and that examines no other operations, it can also be implemented by semaphores. An **rd** that examines a semaphore operation could access the implicit count maintained for the semaphore instead of the invocation queue. On each iteration of the **rd**, the generated code compares the number of invocations that have been examined to the semaphore's implicit count. As mentioned above, such optimizable cases are unlikely to occur in realistic programs. It is therefore not worth implementing this optimization.

9.2.2 Static Analysis

As described in Section 7.3.1, the implementation divides **rd** into three types: unordered invariant, ordered invariant, and variant. Two kinds of optimizations can be applied to the unordered invariant **rd** and the ordered invariant **rd** because of the invariant selection criteria behavior. Currently, we employ a simple and conservative approach to approximating such behavior of the **rd** statically. However, our current analysis cannot detect many commonly occurring cases that are optimizable (see Section 7.3.1.2).

In fact, the compiler can employ more sophisticated analysis to obtain better approximations of the behavior of the selection criteria of **rd**, thus detecting more optimizable cases. As defined in Section 7.3.1.1, an **rd** is optimizable if the following information does not change over all iterations of its execution:

- Values of synchronization and scheduling expressions of invocations
- Invocation parameter values
- Operations that are examined in the **rd**

Dataflow analysis, interprocess analysis, and interprocedure analysis are required to determine whether or not the above information would change over execution of an **rd**. Note that the last behavior is concerned with the **rd** that uses capability variables or arrays of operations. Dataflow analysis extended in [24, 25, 26] could be used to determine all possible values for capability variables.

Two other possible optimizations could also be used to extend the range of optimizable cases. One such optimization is to include only invocations of operations that appear in an **rd** in the examination queue, if known at compile time. Recall from Section 7.3.1.3 that an examination queue maintained for a variant **rd** contains all invocations in the class of operations that appear in the **rd**. If the class contains operations other than those that appear in the **rd**, invocations of these operations are also included in the queue and inspected. The implementation would lead to unnecessary inefficiencies in that case. For example, suppose `f` and `g` are the two operations in class `c`. A variant **rd** that examines only invocations of `f` accesses class `c`. Although the process is interested in only invocations of `f`, it might need to access invocations of `g` while searching for an invocation of `f` to examine. A more efficient approach would be for the compiler to determine which operations would be examined by an **rd** and generate code to include only invocations of these operations in the examination queue. This optimization would reduce the overhead due to the search for an invocation to examine on each iteration. Note that if capability variables are used in the **rd**, dataflow analysis as described above must be used to determine their possible values.

The second optimization is to apply the optimizations of **rd** on specific arms of an **rd** and to treat synchronization and scheduling expressions independently. More precisely, for each operation arm of a variant **rd**, if the synchronization expression is invariant over execution of the **rd**, we can include only invocations whose synchronization expression is true in the examination queue. Similarly, if the scheduling expression is invariant over execution of the **rd**, we can maintain invocations of the operation associated with the arm in the scheduling expression order in the examination queue. For other arms in the variant **rd** that cannot be optimized (if any), their invocations are included in their arrival order in the examination queue as in the current implementation. Note that this optimization cannot be applied when synchronization and scheduling expressions in an **rd** can modify the invocation parameters (see Section 9.5 and also related discussion in Section 9.4). Clearly, this optimization requires significant modification of the compiler's code generator. It also requires several new or modified RTS primitives. The approach for optimizing scheduling expressions is similar to the optimization of input statements proposed in [26, 27]: for an operation that is always serviced with **in** statements that have the same scheduling expression, the RTS can keep the invocation queue ordered by the scheduling expression value. Both optimizations aim at reducing the traversal time of the invocation queue required to find the invocation that minimizes the scheduling expression. However, our approach could also be applied even if the same operation is examined with **rd** statements that contain different scheduling expressions.

9.2.3 Examination Queue

The current implementation of **rd** using an examination queue has three shortcomings, but it could be further optimized. First, the RTS must allocate and later release memory for each element of the examination queue. The cost of such dynamic memory allocation and deallocation constitutes a significant part of the execution cost of **rd** (see Section 8.2.1). One possible optimization to reduce the execution cost would be as follows. Each process can have pre-allocated memory for the examination queue. The compiler analyzes the program to identify which processes need to maintain an examination queue. For each such process, the RTS allocates memory for a certain number of elements, say 10, for the examination queue during the process's initialization. Each such process also maintains a count on the number of available pre-allocated elements. When including an invocation in the examination queue, the RTS uses the pre-allocated element if available and updates the count. When the element has been examined, the RTS increments the count to indicate that a pre-allocated element is available to use. If no pre-allocated elements are available, the RTS then allocates and later releases memory for an element as it originally does.

Second, the examination queue might contain serviced invocations. During execution of an **rd**, invocations that have been included in the examination queue are allowed to be serviced by other processes (see Section 7.2). Thus, the process executing the **rd** might have to visit some serviced invocations in the examination queue while searching for an invocation to examine, which would incur extra overhead. We minimize such overhead as follows. Whenever the RTS finds an invocation that has been serviced in an examination queue, the RTS removes the pointer to that serviced invocation from the queue immediately. The RTS thus visits a serviced invocation in an examination queue at most once.

Third, to examine only one or few invocations via **rd** would be more costly than than to examine all or most of the pending invocations (see Chapter 8). Before examining an invocation, the examination queue is updated to include all unread invocations in the invocation queue. The cost of such an update includes the cost of dynamic memory allocation for new elements in the examination queue as well as the traversal time of the invocation queue. This cost would be substantial and wasteful if only one or few invocations are examined. One possible way to handle this case is to provide compiler directives for the programmer to explicitly specify the expected number of invocations to be examined via an **rd**. The compiler could determine whether or not to use an examination queue for implementing an **rd** based on the information supplied by the programmer. If no examination queue is used, the executing process would search the invocation queue directly for an appropriate invocation on each iteration of the **rd**, similar to the implementation of an input statement. Each process would also need to keep a separate list to record the invocations that have been examined. When selecting an invocation for examination, the process must check against the list to make sure that the invocation has not already been examined.

9.2.4 Marked Invocation List

The **mark** and **take** statements are currently implemented by maintaining a list of marked invocations in each process (see Section 7.3.3). As seen in Section 8.2.4.1, the current implementation of **mark** and **take** is inefficient due to the cost of dynamic memory allocation and deallocation for each element in the marked invocation list. Similar to the optimization discussed in Section 9.2.3, we could improve the current implementation by maintaining pre-allocated elements for the marked invocation list for each process. The compiler can analyze the program and determine which processes contain **mark** statements and also determine the maximum number of operation arms among those **rd** statements that contain **mark** statements. The compiler can then generate code that statically allocates that number of elements for the marked invocation list, thus saving the cost of dynamic allocation and deallocation of those elements.

9.3 Optimizations of SR's Existing Mechanisms

During the development of our new mechanisms, we realized that some optimizations of **rd** are applicable to SR's existing **forward** and **in** statements. Chapter 8 showed that the optimized **rd** performs much better than the unoptimized one. This section discusses how optimizations similar to those discussed in Sections 7.3.1 and 9.2 can be applied to **forward** and **in** statements. We expect that those optimizations would also greatly improve the performance of these statements.

9.3.1 Forward Statements

The current implementation of **forward** is expensive; it requires allocating an invocation block, copying parameter values from the invocation block that is currently being serviced, sending the new invocation block to the RTS. After the forwarded invocation has been serviced, the RTS frees the invocation block but does not reply to the invoker.

One of the optimizations of **rd** described in Section 9.1.5 can be applied to **forward** statements. For example, consider the following **in** statement, that contains one **forward** statement at the end of its command body:

```
in f(x) ->
   ...
   forward g(x)
ni
```

This **forward** statement is also the only **forward** in the command body (which is the simplest and most commonly occurring case). Clearly, it is not necessary for the above **forward** statement to allocate an invocation block because the invocation block that is currently being serviced would not be used after the **forward** is executed. Thus, a **forward** statement that appears at the end of a command body of an **in** statement could be optimized so that it just sends the old invocation block (i.e., the one that is being removed from the invocation queue) to the forwarded operation without copying the invocation block. Additionally, the RTS would not free the invocation block after it has been serviced. As a result, this optimization would save the cost of dynamic allocation of a new invocation block as well as the cost of deallocation of the old invocation block.

The above example shows the simplest requirement for the optimization of **forward**, which is relatively easy to check. Before techniques are developed for for optimizing the cases where some other statements follow the **forward** in the command body of an **in**, further study is required.

```
in f(x) ->
   ...
   forward g(x)
   # some code follows forward
     ...
ni
```

In particular, it is more complicated if another process grabs the forwarded invocation immediately after the **forward** statement is executed. The same invocation block could then be accessed by two processes simultaneously.

9.3.2 Input Statements

As seen in Section 8.2.4, it is more expensive to implement an input statement with synchronization and scheduling expressions than to implement a simple input statement. Currently, the implementation of synchronization and scheduling expressions requires searching the invocation queue for a serviceable invocation.

As mentioned in Section 9.2.2, the optimization proposed in [26, 27] could eliminate the searching time for implementing scheduling expressions for some cases. Specifically, if an operation is always serviced with **in** statements that have the same scheduling expression, the RTS could maintain pending invocations of the operation in the scheduling expression order rather than in FCFS order. Thus, the **in** statement would just service the first pending invocation of the operation.

The implementation of synchronization expressions could also be optimized using an approach similar to that presented in Section 7.3.1 for optimizing **rd**. As the optimization of **rd** takes advantage of the iterative nature of **rd**, this optimization can be applied only to input statements that are used in implementing servers. A server is typically implemented with an infinite loop containing an input statement to service all pending invocations of one or more operations (see Figure 2.1 for example). Thus, if the selection criteria for invocation servicing do not change over execution of the server, invocations that are unselectable can be searched only once.

9.4 Modification of Invocation Parameters

As defined in Chapter 5, the **rd** statement allows examination of invocation parameters. It would also be useful to allow *modification* of the parameters (in which case **rd** would be a misnomer). For example, consider fair share [23] or aging schedulers that periodically update processes' priorities. Requests for service would naturally be represented as invocations to an operation with a parameter indicating priority. All such invocations could be scanned and the priority parameter updated via a simple **rd** statement. To illustrate, consider the following possible syntax:

```
rd f(x) ->  # no change to invocation parameter
   x := 19
[] [wr] g(x,y) -> # wr means write back changes
   x := 5
   y := 11
dr
```

The first operation arm on `f` allows only examination of invocation parameters; i.e., the parameters of the examined invocations of `f` remain unchanged after executing the **rd**. The second operation arm on `g` allows modification of invocation parameters; i.e., the `(x,y)` parameter pairs of all examined invocations of `g` become (5,11). In both cases, the compiler must analyze the program to determine whether the **rd** needs to copy an invocation block before referencing the invocation parameters, as described in Section 9.1.5; otherwise, other processes that reference the same invocation block might obtain inconsistent parameter values. Furthermore, the write-back of invocation parameters must be done atomically at the end of the command body; otherwise, race conditions could happen if one process is accessing the invocation parameters while another is writing back the modified parameter values (e.g., a process accesses an invocation of `g` with parameter values (5,0) when another process has not finished writing back changes to the invocation parameters (5,11)).

Another alternative, analogous to the modes of parameter passing in SR, would be to allow the programmer to specify before executing a command body whether an invocation block is copied in and/or copied out. For example, consider the following code:

```
rd [val] f(x) -> # copy in
   x := 19
[] [var] g(x) -> # copy in and copy out
   x := 5
[] [ref] h(y,z) -> # no copy
   ...
dr
```

We can consider the invocation selected on each iteration of an **rd** as a parameter passed to its operation arm. A `val` invocation is copied in, a `var` invocation is copied both in and out, and a `ref` invocation is passed by address. In the above **rd**, the first and second operation arms have the same effect as the previous example. An invocation block of `f` is copied before executing `x := 19`. Changes to the invocation parameter `x` are not written back to the actual invocation; i.e., the **rd** reads invocations of `f`. An invocation block of `g` is also copied before executing `x := 5`. When the **rd** finishes examining the invocation, changes to the invocation parameters would be copied back atomically to the actual invocation. Therefore, the parameters of invocations of `g` are modified to 5 after being examined. (We use the copy-in/copy-out approach because the current semantics of **rd** allows other processes to access the same invocation concurrently. This approach can guarantee that modified parameters values are only available when the modification of the invocation is completed.) The last operation arm of the above **rd** references actual invocation blocks of `h` in the invocation queue.

Note that when the `ref` mode is used in an operation arm, invocation blocks of the operation are not copied; instead, the actual invocation blocks are accessed directly. We must define a rule prohibiting modification of invocation parameters within the command body. This rule is used to guarantee that other processes would not obtain different parameter values from the same invocation block at different execution times.

9.5 Possible Side Effects

In SR's **in** statements, the evaluation of synchronization and scheduling expressions can have side effects [28]. As we have implemented the **rd** in a way similar to the implementation of SR's **in** statement, the evaluation of synchronization and scheduling expressions in **rd** statements can also have side effects. For example, consider the following **rd** statement, where `t` is a variable local to the process containing this **rd** and `r` is a resource variable.

```
rd f(x) st p(x,t) by q(x,r) ->
   ...
dr
```

Possible side effects include:

- Modification of variables local to the process containing this **rd** statement or modification of resource variables. For example, `t` might be modified in evaluation of `p` if the second formal parameter of `p` is a variable parameter or a result parameter. Similarly, `r` might be modified in the evaluation of `q`.

- Modification of variables local to the process servicing the synchronization expression or the scheduling expression, for example, `p`.

- Generation of new invocations. For example, servicing `p` might invoke other operations.

- Modification of parameters in the invocation. For example, `x` might be modified in the evaluation of `p` or `q` if the first parameter is a variable parameter.

This last side effect can have an undesirable consequence: the parameter in an invocation can be modified, which violates the semantics of **rd**. This side effect may also cause the **rd** not to examine some invocations that should be examined. For example, consider the following **rd** statement:

```
rd f(x) st ++x > 1 ->
   ...
dr
```

This **rd** examines only invocations whose parameter is positive. Suppose two invocations for `f` are pending in arrival order: `f(0)` and `f(3)`. During the first iteration of the **rd**, the synchronization expressions for these two invocations are evaluated (the synchronization expression of `f(0)` is false) and the invocation `f(3)` is examined. The evaluation of `++x` causes the parameter in each invocation incremented by 1. This side effect makes the invocation `f(0)` become selectable in the next iteration (as its invocation parameter has been modified to be 1).

As proposed in [28], the semantics could define a rule prohibiting the evaluation of a synchronization expression or scheduling expression to modify the parameters of the invocation. However, even if such a rule were defined, some of the other side effects would still be possible. These side effects would occur in both **in** and **rd** statements. In addition, most of these side effects are unlikely to occur in typical programs; the above example is just contrived to illustrate our point. Thus, consistent with the **in** statement, we do not impose such a restriction on **rd**.

9.6 Applicability to Other Languages

As discussed in Chapter 4, the invocation handling mechanisms in other concurrent programming languages, such as Ada, Concurrent C, and Linda, have shortcomings similar to those of SR. It is difficult for the programmer to solve common programming situations encountered in program visualization and debugging as well as a number of more sophisticated scheduling scenarios.

Our work is applicable to other concurrent programming languages. In particular, the **rd** mechanism could be very useful. The **rd** gives the programmer the ability to examine invocations without actually servicing them. The mechanisms — **rd**, **mark**, **take**, and **fetch** — are defined, in this thesis, to be consistent with the existing constructs in SR. To apply our work in each of the other languages, we would need to define the mechanisms precisely. For example, in Ada, the `select` statement services an invocation from one of several entries, which are chosen nondeterministically. In that case, the **rd** mechanism should be defined to have similar nondeterministic behavior.

A primitive, `forall`, is proposed for Linda but rejected in [11]. This primitive allows iteration through elements of a tuple space, similar to our **rd** mechanism. The authors of [11] rejected `forall` due to its expected complicated semantics. They raised a number of semantic problems when considering `forall`, which are not further discussed in [11]. Specifically, the semantics of `forall` is not well-defined if the tuple space is being modified by other process. If "locking" is used, the `forall` could lock the tuple space against modifications until it has terminated, or it could just lock all tuples matching the given template. On the other hand, if "locking" is not used, the semantics of `forall` must specify what happen if a tuple is removed or added during execution of the `forall`. As noted above, `forall` is very similar in spirit to the **rd** statement. We have fully specified the semantics of **rd** and resolved those semantic issues (see Chapters 5 and 9). In addition, we have implemented the **rd**. Our results show that the semantics and performance of the implementation of our **rd** are reasonable. Based on our results, a primitive for examination of tuples in Linda, such as the `forall` primitive, is worthy of further investigation.

Chapter 10

Conclusion

This chapter summarizes our work and discusses possible future research.

10.1 Summary

This research has studied how to provide better support for handling invocations in concurrent programming languages. In Chapter 3, we discussed the two limitations in SR for invocation handling. First, invocations can be examined only by actually servicing the invocation. Second, when selecting which invocation to service, only one invocation at a time can be considered within a synchronization or scheduling expression. These limitations make it difficult to solve common programming situations encountered in program visualization and debugging as well as a number of more sophisticated scheduling scenarios. Solutions to these problems using current mechanisms often result in cumbersome and inefficient code. In Chapter 4, we described that similar limitations are also presented in other languages and systems.

We then presented the new language mechanisms — **rd**, **mark**, **take**, and **fetch** — that improve invocation handling and overcome these difficulties. These four new statements are similar in appearance to the existing mechanisms in SR. Moreover, they have simple meanings and are expressive. In particular, we showed how the input statement and the '?' operator can be simulated using the new mechanisms. Many small examples and several larger, realistic ones were also given to illustrate their use and show the expressiveness and flexibility provided by these new mechanisms.

In Chapter 7, we described our implementation of these new mechanisms. The implementation was designed to make common cases efficient. The most general case of **rd** is necessarily more expensive to implement. However, the implementation divides the **rd** into three different types — unordered invariant **rd**, ordered invariant **rd**, and variant **rd** so that the performance of simpler cases is not affected by more complicated cases. We then described the key primitives provided by the RTS and showed the generated code for implementing the new mechanisms. In particular, we showed how different kinds of **rd** statements, some with synchronization or scheduling expressions, are implemented.

In Chapter 8, we evaluated our new mechanisms quantitatively and presented the performance results. We measured the performance of individual mechanisms as well as the performance of some realistic programs that consist of many language elements. Our results showed that the performance of **rd** is efficient in simple cases. In particular, the simulation using **rd** of the **in** statement with a synchronization or scheduling expression is more efficient than the real **in** statement in some cases. Several common applications were rewritten using the new mechanisms in SRR. Some of them have better performance than their SR versions.

In Chapter 9, we explored various design alternatives and described why we chose the particular semantics for our new mechanisms. We identified several optimization possibilities to improve the current implementation. We also described how some optimizations similar to those of **rd** can be applied to **forward** and **in** statements. Finally, we described the applicability of our work to other languages.

10.2 Future Research

Currently, the compiler uses a simple and conservative approach to determine whether or not an **rd** can be optimized and thus can identify only a small range of optimizable cases. We want to enhance the compiler to perform dataflow analysis to obtain a better approximation of program behavior so that additional optimizable cases can be detected. We also want to investigate the application of optimizations similar to those for **rd** to **forward** and input statements to improve their efficiency. In addition, we will further explore how to modify the semantics as well as the implementation of **rd** to allow modification of invocation parameters, which will be useful to several applications.

We designed and implemented our new mechanisms as an extension to SR. The development of our new mechanisms was highly influenced by the current SR implementation due to the presence of the input statement. We are interested in exploring how to design and implement our new mechanisms and the input statement together independent of the existing implementation of the input statement. In particular, as seen in Chapter 6, the input statement can be simulated using our new mechanisms. It is interesting to consider whether the input statement should be eliminated in SR and whether doing so would improve the performance of our new mechanisms. On the other hand, although the '?' operator can also be simulated, it might be a good idea keeping it even if the **rd** statement exists.

Appendix A

Performance Test Programs

This chapter shows the test programs used in our timing tests. The names for the test programs shown here are those used in Chapter 8. The resource body in each of the test programs is run multiple times by enclosing it with a nested loop, making the program run long enough to get reliable statistics. For brevity, the loop that forces the resource body to run multiple times is not shown.

Note that, as seen in the SR solution to the multiple servicing processes problem, the `nap` function is used to control the execution order of several processes so that the concurrent execution of the processes is deterministic and thus reproducible.

```
# rd
resource A()
   op f(int)
   var invs := 1000; getarg(1,invs)
   # generate invocations
   fa i := 1 to invs -> send f(i) af
   # examine invocations of f
   rd f(x) ->
   [] else -> exit
   dr
   # service invocations
   fa i := 1 to invs ->
      in f(x) -> ni
   af
end
```

```
# rd+by
resource A()
   op f(int)
   var invs := 1000; getarg(1,invs)
   # generate invocations
   fa i := 1 to invs -> send f(i) af
   # examine invocations of f
   rd f(x) by x ->
   [] else -> exit
   dr
   # service invocations
   fa i := 1 to invs ->
      in f(x) -> ni
   af
end
```

rd+by+var
```
resource A()
   op f(int)
   var invs := 1000; getarg(1,invs)
   # generate invocations
   fa i := 1 to 10000 -> send f(i) af
   # examine invocations of f
   var n := 1
   rd f(x) by x*n ->
   [] else -> exit
   dr
   # service invocations
   fa i := 1 to invs ->
      in f(x) -> ni
   af
end
```

measurement of overhead due to **rd**
```
resource A()
   op f(int)
   var invs := 1000; getarg(1,invs)
   # generate invocations
   fa i := 1 to invs -> send f(i) af
   # service invocations
   fa i := 1 to invs ->
      in f(x) -> ni
   af
end
```

in+forward
```
resource A()
   op f(int)
   var invs := 1000; getarg(1,invs)
   # generate invocations
   fa i := 1 to invs -> send f(i) af
   # examine all invocations of f
   # without preserving arrival order
   var count := ?f
   fa i := 1 to count ->
      in f(x) -> forward f(x) ni
   af
   # service invocations
   fa i := 1 to invs ->
      in f(x) -> ni
   af
end
```

in+forward+order
```
resource B()
   op f(int), aux_f(int)
   var invs := 1000; getarg(1,invs)
   # generate invocations
   fa i := 1 to invs -> send f(i) af
   # examine all invocations of f
   # while preserving arrival order
   var count := ?aux_f
   fa i:= 1 to count ->
      in aux_f() -> forward aux_f() ni
   af
   do ?f > 0 ->
      in f() -> forward aux_f() ni
   od
   # service invocations
   fa i := 1 to invs ->
      in aux_f(x) -> ni
   af
end
```

\# rd+mark+take

```
resource A()
   op f(int)
   var invs := 1000; getarg(1,invs)
   # generate invocations
   fa i := 1 to invs -> send f(i) af
   # service all invocations of f
   rd f(x) ->
      mark
      take f(x) -> ekat
   [] else -> exit
   dr
end
```

\# rd+fetch

```
resource B()
   op f(int)
   var invs := 1000; getarg(1,invs)
   # generate invocations
   fa i := 1 to invs -> send f(i) af
   # service all invocations of f
   rd f(x) ->
      fetch f(x) -> hctef
   [] else -> exit
   dr
end
```

\# rd+st+fetch

```
resource A()
   op f(int)
   # generate invocations
     ...
   # service all invocations of f
   rd f(x) st x > 0 ->
      fetch f(x) -> hctef
   [] else -> exit
   dr
end
```

\# in+st

```
resource B()
   op f(int)
   # generate invocations
     ...
   # service all invocations of f
   do true ->
      in f(x) st x > 0 ->
      [] else -> exit
      ni
   od
end
```

\# rd+st+var+fetch

```
resource A()
   op f(int)
   # generate invocations
     ...
   # service all invocations of f
   var n := 1
   rd f(x) st x*n > 0 ->
      fetch f(x) -> hctef
   [] else -> exit
   dr
end
```

\# in+st+var

```
resource B()
   op f(int)
   # generate invocations
     ...
   # service all invocations of f
   var n := 1
   do true ->
      in f(x) st x*n > 0 ->
      [] else -> exit
      ni
   od
end
```

```
# measurement of simulation of
# ? operator using rd
resource A
   op f()
body A()
   var invs := 1000; getarg(1,invs)
   # generate invocations
   fa i := 1 to invs -> send f() af
   # service all pending invocations
   var cnt := 0
   rd f() -> cnt++  # simulate ?f
   [] else -> exit
   dr
   do cnt > 0 ->
      receive f()
      # count no. of pending invocations
      cnt := 0
      rd f() -> cnt++
      [] else -> exit
      dr
   od
end
```

```
# use of ? operator
resource B
   op f()
body B()
   var invs := 1000; getarg(1,invs)
   # generate invocations
   fa i := 1 to invs -> send f() af
   # service all pending invocations
   do ?f > 0 ->
      receive f()
   od
end
```

```
# SRR solution to multiple servicing
# processes problem
resource A
   op f()
body A()
   process sender
      fa i := 1 to 100000 ->
        send f(); nap(0)
      af
   end
   process examiner
      do true ->
        rd f() ->
        [] else -> exit
        dr
        nap(0)
      od
   end
   process server
      fa i := 1 to 100000 ->
         in f() ->
         ni
         nap(0)
      af
      stop(0)
   end
end
```

```
# SR solution to multiple servicing
# processes problem
resource B
   op f(), aux_f()
body B()
   # semaphore to control that only
   # one process is examining or
   # servicing the operation f
   sem mutex := 1
   process sender
      fa i := 1 to 100000 ->
         send f(); nap(0)
      af
   end
   process examiner
      var count := 0
      do true ->
         P(mutex) # lock the queue
         # need to preserve order
         # as sender might invoke f
         # while examining
         count := ?aux_f
         fa i := 1 to count ->
            in aux_f() ->
               forward aux_f()
            ni
         af
         do ?f > 0 ->
            in f() ->
               forward aux_f()
            ni
         od
         V(mutex) # release
         nap(0)
      od
   end
   process server
      var n := 0
      do n < 100000 ->
         P(mutex) # lock the queue
         in aux_f() -> n++
         [] else ->
         ni
         V(mutex) # release
         nap(0)
      od
      stop(0)
   end
end
```

```
# SRR two-dimensional scheduler
resource A()
   op f(int;int)
   var invs := 1000; getarg(1,invs)
   # generate invocations
   fa i := 1 to invs ->
      send f(i,i+1)
   af
   process scheduler
      # find invocation with
      # the smallest x
      rd f(x,y) by x ->
         # find invocation with
         # the smallest y
         rd f(x1,y1) st x = x1 by y1 ->
            fetch f(x,y) -> hctef
            exit
         [] else -> exit
         dr
      [] else -> exit
      dr
   end
end
```

```
# SR two-dimensional scheduler
resource B()
   op f(int;int), aux_f(int;int)
   var invs := 1000; getarg(1,invs)
   # generate invocations
   fa i := 1 to invs -> send f(i,i+1) af
   process scheduler
     const MAX := invs+100
     var x1,y1: int
     var count := 0
     fa i := 1 to invs ->
        count := ?aux_f
        x1 := MAX; y1 := MAX
        # find the smallest invocation
        fa i := 1 to count ->
           in aux_f(x,y) ->
              if x < x1 or
                (x = x1 and y < y1) ->
                 x1 := x; y1 := y
              fi
              forward aux_f(x,y)
           ni
        af
        do ?f > 0 ->
           in f(x,y) ->
              if x < x1 or
                (x = x1 and y < y1) ->
                 x1 := x; y1 := y
              fi
              forward aux_f(x,y)
           ni
        od
        # service the selected invocation
        in aux_f(x,y) st x = x1
           and y = y1 ->
        ni
     af
   end
end
```

```
# SRR LIFO scheduler (rd+fetch)
resource A()
   op f(int)
   var invs := 1000; getarg(1,invs)
   # generate invocations
   fa i := 1 to invs -> send f(i) af
   # assign sequence number
   var count := 0
   procedure seqno() returns index:int
      count++
      index := -count
   end
   process scheduler
      # find invocation with the
      # smallest sequence number
      rd f(x) by seqno() ->
         fetch f(x) -> hctef
         count := 0
      [] else -> exit
      dr
   end
end
```

```
# SR LIFO scheduler
resource B()
   op f(int)
   var invs := 1000; getarg(1,invs)
   # generate invocations
   fa i := 1 to invs -> send f(i) af
   # assign sequence number
   var count := 0
   procedure seqno() returns index:int
      count++
      index := -count
   end
   process scheduler
      # find invocation with the
      # smallest sequence number
      fa i := 1 to invs ->
         count := 0
         in f(x) by seqno() -> ni
      af
   end
end
```

```
# SRR LIFO scheduler (rd+mark+fetch)
resource A()
   op f(int)
   var invs := 1000; getarg(1,invs)
   # generate invocations
   fa i := 1 to invs -> send f(i) af
   process scheduler
      fa i := 1 to invs ->
         rd f(x)  -> mark
         [] else -> exit
         dr
         take f(x) -> ekat
      af
   end
end
```

Appendix B

Statistics

This appendix presents the statistics obtained from different machines. The names of the test programs shown here are those used in Chapter 8.

B.1 Execution Times from Different Machines

Test program	Invs (in K)	Iter	Alpha 3000/400	DECstation 260		SGI Indigo		Sun SPARC 5	
in+forward	75	3	10.401	13.010	(25%)	14.630	(41%)	15.20	(46%)
in+forward+order	75	3	11.930	13.041	(9%)	15.221	(28%)	15.76	(32%)
SR invocation+service	75	6	9.712	13.384	(38%)	16.930	(74%)	14.30	(47%)
SRR old primitives	75	6	11.431	14.732	(29%)	17.525	(53%)	14.83	(30%)
SRR new primitives	75	6	11.772	13.963	(19%)	18.690	(59%)	16.70	(42%)
rd	75	4	10.771	13.147	(22%)	14.915	(38%)	14.63	(36%)
rd+by	75	3	11.785	14.845	(26%)	17.884	(52%)	16.62	(41%)
rd+by+var	1	10	8.227	11.666	(42%)	11.243	(37%)	13.52	(64%)
rd+mark+take	75	3	12.532	12.858	(3%)	17.781	(42%)	19.83	(58%)
rd+fetch	75	4	9.499	12.263	(29%)	14.127	(49%)	12.62	(33%)
in	75	6	9.772	13.678	(40%)	16.631	(70%)	13.11	(34%)
in+st (SEQ)	75	6	10.313	13.444	(30%)	15.664	(52%)	12.98	(26%)
in+st (ALT)	1	80	12.342	17.295	(40%)	19.411	(57%)	28.98	(135%)
in+st+var	75	6	10.114	13.475	(33%)	15.962	(58%)	15.03	(49%)
rd+st+fetch (SEQ)	75	4	9.700	12.338	(27%)	15.326	(58%)	12.72	(31%)
rd+st+fetch (ALT)	75	4	10.149	13.115	(29%)	15.425	(52%)	1609.15	(16K%)
rd+st+var+fetch	75	4	14.621	17.115	(17%)	20.885	(43%)	19.95	(36%)
in+by (DESCEND)	1	15	11.299	15.264	(35%)	14.453	(28%)	15.63	(38%)
in+by (ASCEND)	1	15	10.162	13.760	(35%)	13.482	(33%)	13.75	(35%)
in+by+var	1	15	11.888	15.854	(33%)	14.584	(23%)	17.53	(47%)
rd+by (DESCEND)	1	30	9.880	14.170	(43%)	11.005	(11%)	15.43	(56%)
rd+by (ASCEND)	75	6	21.534	29.899	(39%)	33.430	(55%)	29.26	(36%)
rd+by+var	1	10	8.338	12.908	(55%)	11.211	(34%)	13.56	(63%)
LIFO (in)	2	3	12.311	16.775	(36%)	17.408	(41%)	19.25	(56%)
LIFO (rd+fetch)	2	3	17.005	19.814	(17%)	22.329	(31%)	21.52	(27%)
LIFO (rd+mark+take)	1	3	11.461	12.311	(7%)	18.488	(61%)	18.66	(63%)
'?' operator	75	6	10.928	13.194	(21%)	17.013	(56%)	13.79	(26%)
Sim. of '?' operator	1	5	14.712	15.424	(5%)	21.576	(47%)	19.53	(33%)
2-D scheduler (SR)	1	1	11.042	8.909	(-19%)	15.779	(43%)	19.91	(80%)
2-D scheduler (SRR)	1	2	12.373	10.217	(-17%)	21.470	(74%)	17.45	(41%)
SR MSP	150	1	13.543	8.651	(-36%)	19.658	(45%)	38.64	(185%)
SRR MSP	150	1	10.590	7.745	(-27%)	14.269	(35%)	35.33	(234%)
					-36–55%		11–74%		26–16K%

B.2 Execution Cycles from Different Machines

The utilities *pixie* and *prof* were used to measure the number of execution cycles (i.e., instruction cycles) for the test programs run on the Alpha and the DECstation. The table below shows the number of execution cycles, in millions, for the test programs with the same parameter values as presented in the previous section.

Test program	Invs (in K)	Iter	Alpha 3000/400	DECstation 260	Difference	
in+forward	75	3	298.844	415.179	116.335	(39%)
in+forward+order	75	3	310.549	430.035	119.486	(38%)
SR invocation+service	75	6	293.894	414.729	120.835	(41%)
SRR old primitives	75	6	311.444	432.732	121.288	(39%)
SRR new primitives	75	6	318.194	442.182	123.988	(39%)
rd	75	4	309.646	410.234	100.588	(32%)
rd+by	75	3	326.070	449.597	123.527	(38%)
rd+by+var	1	10	368.824	499.074	130.250	(35%)
rd+mark+take	75	3	330.120	446.233	116.113	(35%)
rd+fetch	75	4	274.246	367.634	93.388	(34%)
in	75	6	296.154	419.240	123.086	(42%)
in+st (SEQ)	75	6	288.504	412.040	123.536	(43%)
in+st (ALT)	1	80	729.000	962.529	233.529	(32%)
in+st+var	75	6	289.854	412.940	123.086	(42%)
rd+st+fetch (SEQ)	75	4	276.046	370.034	93.988	(34%)
rd+st+fetch (ALT)	75	4	277.696	378.884	101.188	(36%)
rd+st+var+fetch	75	4	374.146	522.273	148.127	(40%)
in+by (DESCEND)	1	15	474.659	621.282	146.623	(31%)
in+by (ASCEND)	1	15	444.734	591.402	146.668	(33%)
in+by+var	1	15	467.257	606.432	139.175	(30%)
rd+by+fetch (DESCEND)	1	30	325.271	431.486	106.215	(33%)
rd+by+fetch (ASCEND)	75	6	598.097	835.023	236.926	(40%)
rd+by+var+fetch	1	10	367.624	497.674	130.050	(35%)
LIFO (in)	2	3	802.115	983.857	181.742	(23%)
LIFO (rd+fetch)	2	3	890.005	1121.220	231.215	(26%)
LIFO (rd+mark+take)	1	3	672.235	805.923	133.688	(20%)
'?' operator	75	6	315.953	440.389	124.436	(39%)
Sim. of '?' operator	1	5	826.436	978.340	151.904	(18%)
2-D scheduler (SR)	1	1	351.714	478.219	126.505	(36%)
2-D scheduler (SRR)	1	2	347.244	534.784	187.540	(54%)
SR MSP	150	1	390.649	493.802	103.153	(26%)
SRR MSP	150	1	349.850	425.555	75.705	(22%)
						18–54%

B.3 Execution Times for C programs

We developed several C programs to simulate the behavior of several test programs. In each C program, a doubly-linked list is created to simulate an invocation queue. We use 84 bytes as the block size, which is the size of an invocation block with no parameters in the SRR implementation on three of the machines except the Alpha. An invocation block on the Alpha is larger and the size is 144 bytes (see Section 8.4.2 for reasons). We ran these C programs on the four machines with various values of its parameters and measured their execution times. Below we use the names used for their corresponding test programs.

C program	Invs (in K)	Iter	Alpha 3000/400	DECstation 260	SGI Indigo	Sun SPARC 5
in+forward	50	1	0.432	1.222 (183%)	0.938 (117%)	1.27 (194%)
rd	50	1	0.324	1.093 (237%)	0.801 (147%)	1.00 (209%)
in	50	1	0.299	1.143 (282%)	0.747 (150%)	0.99 (231%)
rd+mark+take	50	1	0.375	1.054 (181%)	0.907 (142%)	1.25 (233%)
in+st (ALT)	10	1	3.303	10.018 (203%)	3.386 (3%)	12.49 (278%)
rd+st+fetch (ALT)	50	1	0.342	1.069 (213%)	0.922 (170%)	176.45 (516%)
rd+by	50	1	0.600	2.127 (255%)	1.460 (143%)	1.94 (223%)
rd+by+var	10	1	13.392	41.098 (207%)	15.911 (19%)	16.14 (21%)
rd+by+var	10	2	24.366	85.614 (251%)	31.916 (31%)	31.90 (31%)
rd+by+fetch (DESCEND)	10	1	11.884	41.852 (252%)	16.301 (37%)	16.59 (40%)
rd+by+fetch (ASCEND)	50	1	0.608	2.132 (251%)	1.562 (157%)	1.99 (227%)
rd+by+var+fetch	10	1	13.292	42.102 (217%)	16.173 (22%)	16.12 (21%)
in+by (ASCEND)	10	1	10.566	41.504 (293%)	15.896 (50%)	16.49 (56%)
in+by (DESCEND)	10	1	13.134	41.898 (219%)	16.121 (23%)	19.41 (48%)
in+forward	50	30	10.508	16.447 (57%)	20.455 (95%)	20.02 (91%)
rd	50	30	7.075	12.471 (76%)	16.671 (136%)	12.10 (71%)
in	50	30	6.553	10.638 (62%)	14.803 (126%)	11.62 (77%)
rd+mark+take	50	30	8.639	11.260 (30%)	19.710 (128%)	19.80 (129%)
in+st (ALT)	10	3	8.022	33.645 (319%)	10.553 (32%)	37.27 (365%)
rd+by	50	20	10.073	16.076 (60%)	19.790 (96%)	15.83 (57%)
rd+by+fetch (DESCEND)	10	2	20.565	81.556 (297%)	33.216 (62%)	32.87 (60%)
rd+by+fetch (ASCEND)	50	20	9.186	17.444 (90%)	21.973 (139%)	16.42 (79%)
rd+by+var+fetch	10	2	24.048	83.380 (247%)	31.853 (32%)	32.11 (34%)
				30–319%	3–170%	21–516%

Bibliography

[1] G.R. Andrews. "Synchronizing Resources". *ACM Transactions on Programming Languages and Systems*, 3(4):405–430, 1981.

[2] G.R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1991.

[3] G.R. Andrews and R.A. Olsson. *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1993.

[4] 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.

[5] M.S. Atkins. "Experiments in SR with Different Upcall Program Structures". *ACM Transactions on Computer Systems*, 6(9):365–392, November 1988.

[6] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. "Orca: A Language for Parallel Programming of Distributed Systems". *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[7] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. "Parallel Programming in SR". In *Proceedings of the 1992 International Conference on Computer Languages*, pages 310–319, Oakland, CA, April 1992.

[8] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. "Programming Languages for Distributed Computing Systems". *ACM Computing Surveys*, 21(3):261–322, September 1989.

[9] A. Burns. *Programming in Occam.* Addison Wesley, 1988.

[10] A. Burns, A.M. Lister, and A.J. Wellings. *A Review of Ada Tasking*, volume 262 of *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

[11] P. Butcher, A. Wood, and M. Atkins. "Global Synchronisation in Linda". *Concurrency: Practice and Experience*, 6(6):505–516, September 1994.

[12] N. Carriero and D. Gelernter. "Linda in Context". *Communications of the ACM*, 32(4):444–458, April 1989.

[13] A. Charlesworth. "The Multiway Rendezvous". *ACM Transactions on Programming Languages and Systems*, 9(2):350–366, February 1987.

[14] M. Coffin and R. A. Olsson. "An SR Approach to Multiway Rendezvous". *Computer Languages*, 14(4):255–262, 1989.

[15] W. A. Crow. *Inter-Entry Selection — Implementation and Performance Analysis.* Master's thesis, Dept. of Computer Science, California State University, Sacramento, May 1995.

[16] T. Elrad and F. Maymir-Ducharme. "Distributed Languages Design: Constructs for Controlling Preferences". In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 176–183, St. Charles, Illinois, August 1986.

[17] T. Elrad and F. Maymir-Ducharme. "Satisfying Emergency Communication Requirements with Dynamic Preference Control". In *Proceedings of Sixth Annual National Conference on Ada Technology*, March 14-17 1988.

[18] N. Gehani. *UNIX Ada Programming.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987.

[19] N. Gehani and W.D. Roome. *The Concurrent C Programming Language.* Silicon Press, Summit, NJ, 1989.

[20] D. Gelernter. "Generative Communication in Linda". *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[21] S.J. Hartley. *Operating Systems Programming: The SR Programming Language.* Oxford University Press, New York, NY, 1995.

[22] C.A.R. Hoare. "Communicating Sequential Processes". *Communications ACM*, 21(8):666–667, August 1978.

[23] J. Kay and P. Lauder. "A Fair Share Scheduler". *Communications ACM*, 31(1):44–55, January 1988.

[24] C. M. McNamee and R. A. Olsson. "Transformations for Optimizing Interprocess Communication and Synchronization Mechanisms". *International Journal of Parallel Programming*, 19:357–387, October 1990.

[25] C. M. McNamee and R. A. Olsson. "An Attribute Grammar Approach to Compiler Optimization of Intra-Module Interprocess Communication". *International Journal of Parallel Programming*, 20(3), June 1991.

[26] C.M. McNamee. *Compiler Optimization of Interprocess Communication and Synchronization Mechanisms.* PhD thesis, University of California, Davis, June 1991.

[27] R. A. Olsson. "Using SR for Discrete Event Simulation: A Study in Concurrent Programming". *SOFTWARE — Practice and Experience*, 20(12):1187–1208, December 1990.

[28] R.A. Olsson. *Issues in Distributed Programming Languages: The Evolution of SR.* PhD thesis, Dept. of Computer Science, The University of Arizona, August 1986.

[29] R.A. Olsson and C.M. McNamee. "Inter-Entry Section: Nondeterminism and Explicit Control Mechanisms". *Computer Languages*, 17(4):269–282, 1992.

[30] R.A. Olsson and C.M. McNamee. "Tools for Teaching CCRs, Monitors, and CSP Concurrent Programming Concepts". *SIGCSE Bulletin*, 27(2):31–40, June 1995.

[31] T. Purdin, R. Hayes, G.R. Andrews, and R. Schlichting. "The Design of the Saguaro Distributed Operating System". *IEEE Transactions on Software Engineering*, 13(1):104–118, January 1987.

[32] PVM. *Parallel Virtual Machine System (PVM) Version 3 Manual Pages*, 1992.

[33] C.A. Waldspurger and W.E. Weihl. "Lottery Scheduling: Flexible Proportional-Share Resource Management". In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 1–11, Monterey, California, November 1994. USENIX.