A Comparison of Concurrent Programming and Cooperative Multithreading under Load Balancing Applications

by

JUSTIN TRAVIS MARIS

B.S. (University of California, Davis) 2000

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Professor Ronald A. Olsson (Chair)

Professor Matthew Bishop

Professor Matthew Farrens

Committee in Charge

2002

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The general concurrent programming execution model (CP) provides independent processes as its key abstraction. Threads execute nondeterministically and context switches can occur arbitrarily. This model of execution is found in many concurrent programming languages, e.g., Ada [9], CSP [8], Java [4], Orca [3], and SR [2]. These languages provide various synchronization mechanisms to coordinate the execution of processes (e.g., semaphores, monitors, or rendezvous).

The cooperative multithreading execution model (CM) provides threads that execute one at a time until they choose to yield the processor. Typically, threads yield the processor by executing an explicit yield statement. A more general version of CM, called parallel CM (PCM), allows multiple threads to be active simultaneously so that a CM program can run on multiple processors [11]. PCM programs can be run on a multiprocessor machine using shared memory or on a distributed network of processors. This research focuses on the distributed form of PCM style programming on a network of single processor machines.

This research utilizes the SR and Dynamic C [1] programming languages. SR supports both CP and CM styles. Dynamic C supports only CM style. These languages are used to implement dynamic load balancing versions of two classic problems: Dining Philosophers and Readers/Writers. As part of the Dynamic C implementation, an SR-like run-time system, DesCaRTeS, has been developed to provide inter-process communication. Both SR and Dynamic C implementations are run with and without load balancing to test the overall effectiveness of load balancing. Furthermore, the

SR implementations are written using both CM and CP styles so that performance comparisons can be made.

In some cases CM style programs outperform CP style programs [11]. The research presented in this thesis has shown similar results and provides insight into the tradeoffs between CM and CP. By looking at applications that use CM, we hope that this research will eventually lead to future work that reveals the benefits of CM style programming. Furthermore, this research examines load balancing under dynamic circumstances to reveal the advantages and disadvantages of load balancing during process execution.

The rest of this thesis is organized as follows. Chapter 2 discusses work related to this research. Chapter 3 describes the applications developed using load balancing models. Chapter 4 introduces DesCaRTeS (a run-time system for Dynamic C) and outlines its use and features. Chapter 5 presents the experimental performance results. Chapter 6 discusses these results. Chapter 7 presents ideas for future work and the conclusions reached through this research.

Chapter 2

Related Work

This chapter discusses work related to this research. Section 2.1 provides an overview of the SR programming language. Section 2.2 presents an overview of the Dynamic C programming language. Finally, Section 2.3 discusses load balancing research.

2.1 SR Overview

The SR programming language [2] provides support for both CP and CM programming styles. Furthermore, it provides intuitive methods to dynamically create processes and virtual machines. Inter-process communication can also be used through a variety of SR's language features.

SR uses resources as its key component. A resource is essentially a template that can be used to dynamically create instances of that resource. Each resource may also spawn more processes and/or create additional resources.

Processes in SR communicate using operations. A process may invoke these operations, sometimes with parameters, by either using a **send** or **call** statement. Invocations created with **send** are asynchronous. Thus the invoker may continue after the invocation is made. Invocations created with **call** are synchronous, and thus, cause the invoker to block until the invocation is serviced. Invocations are serviced by input statements (defined by the keyword **in**) or procedures. Input statements receive an invocation from any one of a finite number of operations. The invocation selected is determined on a FCFS basis. This selection method, however, can be modified to follow specific scheduling rules. Specifically, SR provides two clauses to accomplish this: **st** and **by**. The **st** clause causes the operation to be serviced only if the condition following the clause is evaluated to be true. An input statement can also give preference to invocations according to an ordering determined by the expression that follows the **by** clause. Once an invocation has been selected, additional code can be executed as part of the servicing of that invocation. Once that code, if any, is executed, a reply is sent back to the invoker if the invocation was made with a **call**. The invoker may then continue. If the invocation was made with a **send**, however, no such reply is sent because the invoker has already continued after the **send**. Input statements can also respond to the invoker before the entire invocation servicing code has executed by using SR's **reply** statement. This allows an invoker waiting for its invocation to be serviced to continue with its execution, using any results that accompany the **reply**. When invocations are serviced by procedures, a new process is spawned to handle the call. These procedure calls can either be local or remote. SR also provides semaphores, which are abbreviated versions of operations. The P and V operations, which are simplified input and **send** statements, respectively, are used on semaphores.

Figures 2.1 and 2.2 give examples of the Dining Philosophers problem and the Reader/Writers problem written in SR. The Dining Philosophers solution presented here (from Reference [2]) uses a centralized controller that handles the distribution of forks. Similarly, the Readers/Writers solution also provides a centralized controller that handles requests to the database. These requests are serviced with a FCFS policy, thus not starving any one process. When choosing between waiting readers or writers, the algorithm gives preference to whichever has been waiting longest. Furthermore, readers that have arrived before any waiting writer are allowed to proceed [10]. These examples illustrate many of SR's features discussed in this section.

The standard SR implementation uses an iteration-counting approach to provide support for the implicit context-switching found in CP. This approach allows a process to execute for some number of loop iterations, based on the observation that if a process is running and it does not block, then it must be looping. The SR compiler generates code to effect these implicit context switches. The SR implementation

```
resource Servant
        op getforks(id: int)  # called by Philosophers
        op relforks(id: int)
body Servant(n: int)
        process server
                var eating[1:n] := ([n] false)
                do true ->
                        in getforks(id) st not eating[(id mod n) + 1]
                           and not eating[((id-2) mod n) + 1] ->
                                eating[id] := true
                        [] relforks(id) ->
                                eating[id] := false
                        ni
                od
        end
end

resource Philosopher
        import Servant
body Philosopher(s: cap Servant; id, t: int)
        process phil
                fa i := 1 to t ->
                        s.getforks(id)
                        write("Philosopher", id, "is eating")   # eat
                        s.relforks(id)
                        write("Philosopher", id, "is thinking") # think
                af
        end
end

resource Main()
        import Philosopher, Servant
        var n:= 5, t:= 5
        # create the Servant and Philosophers
        var s: cap Servant
        s := create Servant(n)
        fa i := 1 to n ->
                create Philosopher(s, i, t)
        af
end
```

Figure 2.1: Dining Philosophers written in SR

```
resource rw()

        op start_read()
        op end_read()
        op start_write()
        op end_write()

        process RW_allocator
                var nr := 0
                do true ->
                        # all input statements take advantage of FCFS servicing of invocations.
                        in start_read() ->
                                reply
                                nr++
                                do nr > 0 ->
                                        in start_read() -> nr++
                                        [] start_write() ->
                                                # don't let it proceed until all
                                                # reads have finished
                                                do nr > 0 ->
                                                        in end_read() -> nr-- ni
                                                od
                                                # nr=0, so let the write proceed; then
                                                # wait for it to finish.
                                                reply
                                                receive end_write()
                                                # at this point nr=0 so outer do nr>0 loop
                                                # will terminate too.
                                        [] end_read() -> nr--
                                        ni
                                od
                        [] start_write() ->
                                reply
                                receive end_write()
                        ni
                od
        end
        process one
                fa i := 1 to 5 ->
                        start_read()
                        # read
                        write("process one is reading")
                        end_read()
                af
        end
        process two

                start_read()
                # read
                write("process two is reading")
                end_read()

                fa i := 1 to 3 ->
                        start_write()
                        # write
                        write("process two is writing")
                        end_write()
                af
        end
end
```

Figure 2.2: Reader/Writers with fairness written in SR

```
do true ->
   # think
     ...
   # get forks
   # (use semaphore operations
   # on shared sem array fork;
   # left and right are indices
   # of neighboring philosophers)
   P(fork[left]);P(fork[right])
   # eat
     ...
   # release forks
   V(fork[left]);V(fork[right])
od
```

(a) CP-style

```
do true ->
   # think
     ...
   # get forks, by simulating:
   #   await fork[left]=1 & fork[right]=1
   do not(fork[left]=1 & fork[right]=1) ->
     nap(0) # i.e., yield
   od
   fork[left] := 0; fork[right] := 0
   # eat
     ...
   # release forks
   fork[left] := 1; fork[right] := 1
od
```

(b) CM-style

Figure 2.3: Code for a philosopher in Dining Philosophers

provides a compile-time option that inhibits this code generation, eliminating these implicit context switches. This allows the execution of CM style SR programs.

A process can use SR's **nap** function to effect the equivalent of an explicit yield statement (i.e., to force a context switch) in a CM style SR program. Passing **nap** a value of 0 simply causes the process to yield until it is scheduled to execute again. Passing **nap** a non-zero value puts the process to sleep for the given amount of time. Once that time has passed, the process is rescheduled so that it can execute again. Figure 2.3 illustrates a philosopher from Dining Philosophers written in both CP and CM styles. This version of the Dining Philosophers problem presents a decentralized solution. In the CP version, philosophers acquire forks through semaphores. In the CM version, philosophers acquire forks using shared variables. Most philosophers grab their left forks first, right forks second. To prevent deadlock due to a circular wait condition, an asymmetric philosopher is introduced. This philosopher grabs its right fork first, left fork second.

2.2 Dynamic C Overview

Dynamic C [1] extends the C language with various features to support CM. Its **costate** statement defines a block of statements, which executes as a separate thread with its own hidden instruction counter. A thread executes until it chooses to yield the processor or to wait for some event to become true. Yielding the processor is accomplished via explicit statements: **yield** and **waitfor**. **yield** context switches to another ready thread, if any, or resumes the current thread if no other thread is ready. **waitfor** evaluates the condition. If true, the thread continues; otherwise, the thread yields and will, therefore, reevaluate the condition when it runs again.

The execution of **costate** statements can be controlled by the use of their respective **CoData** structures. This structure has two relevant flags: **STOPPED** and **INIT**. The **STOPPED** flag is set when the thread is currently not scheduled to execute. The **INIT** flag is set when the thread is going to start execution at the first statement defined within the **costate** block. By default, a **costate** statement has both the **INIT** and **STOPPED** flags set. This initial state can be altered, however, by using the **init_on** flag immediately after the **costate**'s name. This flag sets **INIT** but clears **STOPPED**, causing the thread to execute when first reached by the program. In addition, there are four functions that are used to alter the **INIT** and **STOPPED** values: **CoBegin**, **CoReset**, **CoPause**, and **CoResume**.

- **void CoBegin(CoData * cd)**

  Clears the **STOPPED** flag and sets the **INIT** flag.

- **void CoReset(CoData * cd)**

  Sets both the **STOPPED** and **INIT** flags.

- **void CoPause(CoData * cd)**

  Sets the **STOPPED** flag and clears the **INIT** flag.

- **void CoResume(CoData * cd)**

  Clears both the **STOPPED** and **INIT** flag.

cofunc functions are useful when multiple **costate** statements wish to execute the same code. These are the only kind of functions that can use **yield** or **waitfor** statements. However, to allow multiple calls to a **cofunc** function, multiple instances must be declared. When the function is called, the caller must designate which instance it is using. Calling an instance of a **cofunc** function that is already in use will

```
#define T 1000
#define N 5

int lfork[N];

cofunc void philosopher[N](int right, int left){
        int i;

        for(i = 1; i <= T; i++){
                // grab forks
                while(fork[left] == 0){ yield; }
                fork[left] = 0;
                while(fork[right] == 0){ yield; }
                fork[right] = 0;

                // eat

                // release forks
                fork[left] = 1;
                fork[right] = 1;

                //think
        }
}

main(){
        int i;
        CoData philosopher0, philosopher1,
               philosopher2, philosopher3,
               philosopher4;

        // start the philosophers
        CoBegin(philosopher0);
        CoBegin(philosopher1);
        CoBegin(philosopher2);
        CoBegin(philosopher3);
        CoBegin(philosopher4);
        // initialize forks
        for(i = 0; i < N; i++){
                fork[i] = 1
        }
        // execute all of the threads
        for(;;){
                costate philosopher0{
                        wfd philosopher[0](0, 1);
                }
                costate philosopher1{
                        wfd philosopher[1](1, 2);
                }
                costate philosopher2{
                        wfd philosopher[2](2, 3);
                }
                costate philosopher3{
                        wfd philosopher[3](3, 4);
                }
                // the asymmetric philosopher
                costate philosopher4{
                        wfd philosopher[4](5, 4);
                }
        }
}
```

Figure 2.4: Dining Philosophers in Dynamic C

cause the earlier call to be terminated. Instances of **cofunc** function must be called within an **wfd** (wait for done) statement. The **wfd** statement may have multiple **cofunc** functions as parameters. This statement yields if any of the **cofunc** functions do not completed execution as a result of explicit yields. When the thread runs again, the **cofunc** functions resumes execution of all uncompleted **cofunc** functions.

Figure 2.4 illustrates a solution to the Dining Philosophers problem written in Dynamic C. It uses many features of Dynamic C, including those mentioned in this section.

2.3 Load Balancing

Load balancing is defined as spreading workload across all processors in a system with the goal of reducing overall program execution time. Many load balancing algorithms attempt to load balance as new work enters the system [14, 12]. Over time, however, the load on the system may become unbalanced due to uneven processing speeds and thus needs to be redistributed. Studies have shown that this kind of dynamic load balancing can produce favorable results [6]. Similarly, this thesis studies load balancing that is done periodically during process execution. As the state of the system becomes unbalanced, processes are moved from one processor to another.

Chapter 3

Applications

This research examines applications designed based on two classic problems. The first application revisits the Dining Philosophers problem [2]. The second application is a form of the Readers/Writers problem [2]. Using both PCM and CP styles of programming, these applications are developed to include load balancing and dynamic process creation and destruction.

3.1 Dynamic Dining Philosophers

The classic Dining Philosophers problem involves a number of philosophers gathered around a table. Each philosopher spends time either eating or thinking. In order to eat, a philosopher must grab both its left fork and its right fork. These forks, however, are shared with a philosopher's left and right neighbors. These forks must be obtained in a way that guarantees exclusive access (e.g., a semaphore). As a result, no two neighboring philosophers may eat at the same time.

This research uses an expanded version of this classic problem that allows philosophers to periodically enter or leave the system. This version of the dining philosophers is typically referred to as the Dynamic Dining Philosophers (DDP)[5].

When a philosopher attempts to enter the system, it first will designate a new fork as its left fork. Next, it is informed of its new left and right neighbors. It then must introduce itself by informing both neighbors of its arrival. The left neighbor receives additional information about its new right fork, the fork that the incoming philosophers brings to the table. The new philosopher may enter the system only after these changes have been made.

When a philosopher attempts to leave the system, it must first inform each of its neighbors of its departure. The left neighbor receives additional information about its new right fork, which is the old right fork of the departing philosopher. Then, the departing philosopher disposes of its left fork and leaves the system.

To guarantee that the system never reaches an incorrect state, simultaneous arriving and departing philosophers should be serviced sequentially. This is to protect against an incoming philosopher attempting to communicate with a recently departed philosopher. Such a situation may cause the state of the system to become erroneous or even deadlock [5, 13].

To run under the PCM model, the set of philosophers is divided into N groups. Each group lies in its own region, which represents either a real or virtual processor. Since each region runs CM style code, forks that are on the interior of the region are obtained through shared variables. Forks that lie between regions, however, are obtained through distributed semaphores. For simplicity, those philosophers that lie on the border of a region and use semaphores to grab forks are considered immortal and are not allowed to leave the system. This simplification eliminates the code needed to deal with special cases that have one or zero philosophers present in a region. Such cases can cause the region boundaries to become unclear. Figure 3.1 illustrates the general setup for DDP using two regions.

3.1.1 General Synchronization Issues

As mentioned, philosophers are allowed to leave and enter the system, which requires synchronization to meet the following requirements:

Mutual Exclusion A philosopher brings or takes a fork as it arrives or leaves from the table. Thus, a philosopher's two neighboring philosophers must be notified before it attempts to enter or leave the system. This prevents the grabbing of the same fork simultaneously or grabbing an incorrect fork. To simplify matters, such notifications are only serviced during the neighbors' non-eating phases, i.e., obtaining forks is non-preemptive.

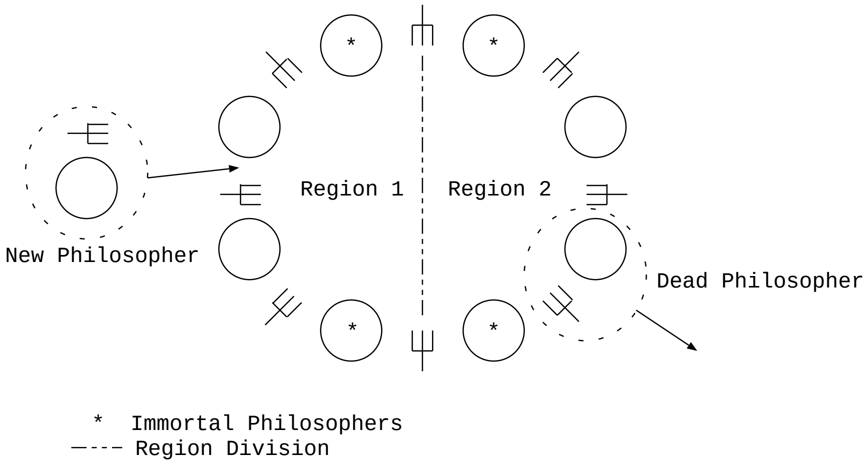

Figure 3.1: The Dynamic Dining Philosophers Model used to implement load balancing

Deadlock In the static model of Dining Philosophers, deadlock is avoided by having one asymmetric philosopher, which grabs its forks in the reverse order from other philosophers. Thus, a deadlock might occur if the asymmetric philosopher leaves the system. All remaining philosophers could grab their forks in the same order, which may lead to a circular wait. Currently, such a scenario is avoided by not allowing the asymmetric philosopher (one of the immortal ones) to leave. A less restrictive solution would be to reassign the asymmetric philosopher. To accomplish this, additional code would be needed to change fork grabbing methods of affected philosophers. This alternative, however, was not used since immortal philosophers already existed and assigning one of these philosophers as asymmetric simplified the problem.

Starvation When a philosopher attempts to enter or leave the system, its two neighboring philosophers must pause in order to synchronize. If philosophers repeatedly enter and leave the system in the same location, then the neighboring philosophers might continually be trying to synchronize and never eat. Thus, a dirty bit is introduced to force an affected philosopher to eat and think at least once before attempting to synchronize.

3.1.2 Load Balancing Models

To introduce load balancing, an efficient algorithm was designed that will reduce overhead yet provide the necessary synchronization. Initially, moving philosophers from one region to the next was restricted to those philosophers that lie on the border. Thus, load balancing only considered whether moving a philosopher on the border of one region to the border of the other region would balance the workload. A more flexible approach allowed groups of consecutive philosophers to move across this border. But, unfortunately for both cases, the overhead of performing such a move proved too costly. Thus, we use the DDP model as the basis for implementing load balancing, utilizing its death and creation abilities to move a philosopher from one region to another. This model also allows the load balancer to move any mortal philosopher, adding to the algorithm's flexibility. Furthermore, synchronization for load balancing was provided in large part by DDP. One process on a separate processor handles all deaths, creations, and load balancing. The philosophers are run on two separate virtual machines (one for each region), each running on a separate processor.

A balanced system is determined by whether or not workloads on each region can be redistributed better by moving a philosopher. This information is gathered by having the balancing process send load request messages to each processor in the system. These messages are serviced by processes running on each processor. Once the balancer process has gathered system load information from each processor, load balancing potential is evaluated. Other research has shown this load information gathering scheme to be an effective method to achieve load balancing [6].

Two different load balancing models for DDP have been implemented. The first model balances by moving the philosopher with the smallest amount of work. This model will be referred to as **DPLBsmall**. The second spends more time analyzing the status of each processor and determines the largest philosopher that could be moved to yield the most balanced system. This load balancing model will be referred to as **DPLBlarge**. The non-load balancing model will be referred to as **DPNLB**.

3.1.2.1 Synchronization Issues for Load Balancing

To ensure that the system never reaches a bad state due to load balancing, some synchronization issues must be examined.

Mutual Exclusion A fork cannot be grabbed by two philosophers simultaneously due to load balancing since the algorithm uses DDP's death and creation procedures.

Deadlock Currently, the four philosophers aligned along the border never die or move. Since one of these is the asymmetric philosopher, this avoids the potential deadlock situation as described in Section 3.1.1. It also means that how a given philosopher grabs its fork is fixed (always by semaphores or always by using shared variables). Specifically, border philosophers grab their interior forks using shared variables and their forks shared with philosophers in other regions using semaphores. Interior philosophers always use shared variables to acquire forks.

Starvation A philosopher could be starved if it is moved from one region to the next repeatedly such that it never gets to eat. To avoid this, we take advantage of the dirty bit so that a philosopher cannot change regions until it cycles through at least one iteration of eating and thinking.

3.2 Readers/Writers

The classic Readers/Writers problem involves a number of readers and writers that all attempt to periodically access a centralized database. However, certain restrictions apply. Any number of readers may access the database simultaneously, provided no writers currently have access. Each writer, on the other hand, must have exclusive access. Thus, no other reader or writer may be accessing the database while a writer has access.

The research here slightly modifies this classic problem to allow readers or writers to periodically enter and leave the system. More specifically, a process enters the system either as a reader or a writer and performs a certain number of iterations of

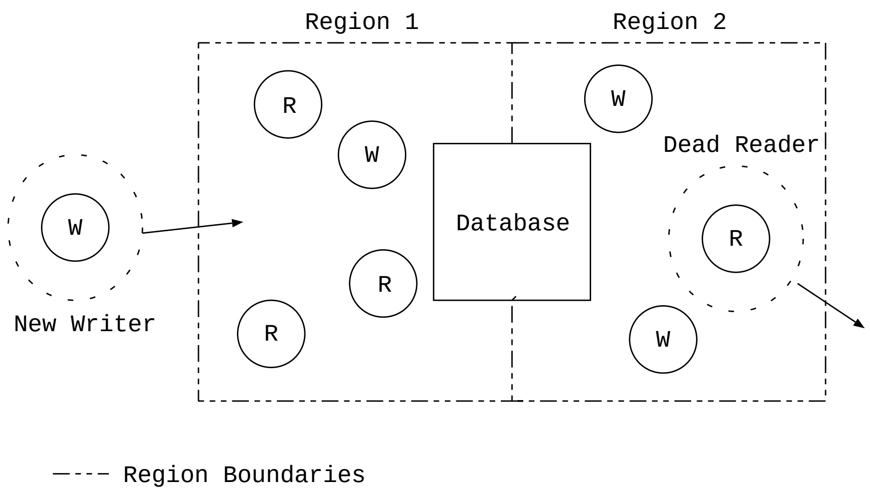

Figure 3.2: The Dynamic Readers/Writers Model used to implement load balancing

work. Each iteration includes some independent work as well as work requiring access to the database. Once all of this work is completed, the process then terminates. This thesis will refer to this version of Readers/Writers as Dynamic Readers/Writers (DRW). Figure 3.2 illustrates DRW for two regions.

The Readers/Writers solution is implemented using the FCFS policy discussed in Section 2.1. Specifically, the DRW models used for this research use the **RW_allocator** process shown in Figure 2.2.

3.2.1 General Synchronization Issues

The following synchronization issues must be addressed for the DRW problem.

Mutual Exclusion Requests for the database are handled by only one process. This process regulates the number of readers or writers in the database. Thus, a writer can only access the database if it will be the only process with that access.

Deadlock Deadlock would occur if all processes blocked indefinitely. Since processes only block while waiting for access to the database, deadlock could only happen if some process accessing the database failed to exit and release the database. However, processes accessing the database do not block indefinitely, and thus continue execution and eventually release access to the database. Any processes waiting to access the database will then be allowed to continue.

Starvation A process would be starved if its request to access the database is never serviced. This cannot occur since requests are serviced by a FCFS policy and processes access the database for finite time.

3.2.2 Load Balancing Models

Two load balancing models were explored in this research. One model considers the number and type of processes active in each region. The other considers the total workload in each region. This section describes these two models as well as other potential but untested solutions.

The first model attempts to balance the system such that an equal number of readers exist in each region and an equal number of writers exist in each region. These processes, randomly placed when first introduced to the system, can be moved to another region as the numbers of readers and writers on each region becomes uneven. Information about the state of the system is gathered in the same manner as described in Section 3.1.2. This model will be referred to as **RWLBbynum**.

The second model attempts to balance the system by trying to allocate equal amounts of work to each processor. The model disregards whether or not a process is a reader or writer. Thus, this load balancing technique is in many ways similar to the DDP load balancing model described in Section 3.1.2. Unlike the DDP model, however, processes do not have neighboring processes. Thus, DRW does not need to have readers or writers synchronize with other readers or writers when they enter or leave the system. Information about the load on each processor is gathered in the same manner as described in Section 3.1.2. This model will be referred to as **RWLBbywork**. The non-load balancing model will be referred to as **RWNLB**.

A number of other load balancing models exists. The balancer could attempt to place all writers in one region while placing all readers in the other. Similarly, readers could be distributed randomly while writers are relocated to a designated region. Of course, any hybrid of the mentioned load balancing models could also be used. However, this research examines only the **RWLBbynum** and **RWLBbywork** algorithms.

3.2.2.1 Synchronization Issues for Load Balancing

Load balancing introduces new synchronization concerns for the DRW problem.

Mutual Exclusion Load balancing does not affect mutual exclusion. Operations on the database are independent from operations involving the movement of processes.

Deadlock Processes are not allowed to move while accessing or attempting to access the database. This ensures that the process being moved does not tie up the database while it is in transit. Doing so could cause the system to become deadlocked since a process may be granted access during or after transfer. This would cause the database to try and fulfill a request for a process that no longer exists in a particular region.

Starvation Processes must complete at least one iteration of work before being moved across regions. This prevents the potential continual movement of a process from one region to another.

Miscellaneous In order to guarantee that load balancing is performed using correct information about the state of the system, readers and writers must check in and check out. This ensures that processes that are requested to move are still active and that the number of processes active on all processors is accurate.

Chapter 4

Dynamic C Run-time System (DesCaRTeS)

To implement the Dynamic Dining Philosophers and Readers/Writers algorithms in Dynamic C, a run-time system (RTS) needed to be implemented. This RTS, called DesCaRTeS, mimics the run-time system of SR [2] and provides many features of the SR language (see Section 2.1). These features include operations, capabilities, semaphores, inter-process communication, and process creation.

4.1 DesCaRTeS Overview

DesCaRTeS is used to provide programs written in Dynamic C with SR-like features. It does this by using data structures and algorithms that provide similar semantics to those of SR. This section provides an overview of DesCaRTeS. It provides a general template for developing programs and sample code to illustrate the library's usage.

DesCaRTeS uses the TCP/IP library available with the Dynamic C distribution to provide inter-process communication across a network. This library allows a programmer to open and close connections, read and write from sockets, and use the many other features typically found in a TCP/IP library. DesCaRTeS uses this library to also establish connections with other processors. This is semantically equivalent to SR's virtual machine creation. Each connection established is stored in a table. Processes communicating with another machine use an index into that table to signify message destinations.

The primary form of communication between processes is through special variables called operations. These operations are stored within a table internal to DesCaRTeS. Each entry in the table has an invocation queue and a blocking queue. Whenever a process sends an invocation of an operation, that invocation is placed in the invocation queue of that operation. When a process receives from an operation, one of two things can occur. If an invocation for the operation exists, the process grabs the invocation and proceeds. If no such invocation is present, the process places itself on the blocking queue for the operation and blocks. Once an invocation becomes available, the process is then restarted.

Operations often have parameter lists. DesCaRTeS represents these lists as strings. Strings were used since the number of arguments for any particular message is unknown at compile time. Thus, the correctness of building and parsing messages is the responsibility of the programmer, with help from functions in the DesCaRTeS library.

4.1.1 DesCaRTeS Template

Programs intending to use DesCaRTeS must follow a specific format to ensure correct functionality of the library. Figure 4.1 provides a basic template for writing programs using DesCaRTeS. Lines 1 through 25 illustrate header code that should be included. This includes any redefinitions of macros used by DesCaRTeS. All global constants, forward process declarations, and global variables should be declared here. Lines 27 through 51 illustrate how an SR resource should be written using Dynamic C and DesCaRTeS. Lines 52 through 84 show how **main** should be written. The **for** loop inside **main** contains multiple costatements, which represent all of the potential resource instances and processes, including the RTS. The RTS costatement should always use the **init_on** flag (see Section 2.2). This starts the RTS as soon as the program is started. The only other process that should use this flag is the main resource. Much of this template is optional, depending on the structure of the program, but certain parts are required. Generally, those parts that are required appear as code in the template. Parts not required appear as comments.

```
#memmap xmem

// Change these from the TCP/IP library
// defaults.
#define MY_IP_ADDRESS "10.1.1.1"
#define MY_NETMASK "255.255.255.248"

// Change this if this machine will be
// communicating with more than 4 other
// machines.  The default is 4.
 #define MAX_SOCKETS 4

// declare global constants

// import the rts library
#use "rts.lib"

// Forward declarations of CoData structures.
// These represent all possible resource
// instances as well as any additional
// processes which may be created.
CoData process1;
...
CoData processn;

// declaration of global variables

// resource instances
cofunc void proc_cofunc[n](CoData * p){
     // Special variables needed for RTS
     // should be declared, ie: buffers

     // declare local operations, variables
     // and semaphores

     // This should/must be called after all
     // variable declarations
     RTSSTART_PROCESS(p);

     // Grab and Parse any creation
     // parameters

     // Call RTSOpDelare and RTSSemInit on
     // all ops and sems

     // Initialize local variables

     // process code

     // Finish process
     RTSBODY_END;
}
main(){
     // Special variables needed for RTS
     // should be declared, ie: buffers

     // Initialize the RTS
     RTSInit()

     // Register resource instances in the RTS
     RTSRegCoData(&process1, "process")
     ...
     RTSRegCoData(&processn, "process")

     // Global initialization code

     // This loop contains costates for all
     // potential resource instances and
     // processes, including the RTS.  A
     // process other than RTSprocess should
     // be init_on only if this is the
     // main machine.
     for(;;){
             costate RTSprocess init_on{
                 wfd RTS();
             }
             costate process1{
                 wfd proc_cofunc[1](&process1);
             }
             ...
             costate processn{
                 wfd proc_cofunc[n](&processn);
             }
     }
}
```

Figure 4.1: Template for using DesCaRTeS

4.1.2 Process Communication

Communication between processes is achieved in the same fashion as in SR. Processes can declare operations and send invocations to and receive invocations from those operations. Capabilities to operations can be passed to other processes as well.

Figure 4.2 provides some sample code that illustrates how to use operations in DesCaRTeS. Process **A** declares an operation **a** and sends the capability to a newly created process **B**. As discussed in Section 4.2.4, DesCaRTeS does not provide resource capabilities. Instead, capabilities to operations of a newly created process are sent back to the creator after the operations have been declared. These capabilities are then explicitly parsed. Thus, process **B** returns the capability to its operation **b** back to process **A**. Now both processes **A** and **B** can use both operations **a** and **b**. Process **A** creates invocations using **RTSSend** and services invocations using **RTSReceive**. Process **B** creates invocations using **RTSCall** and services invocations using DesCaRTeS's functions that simulate input statements.

Figure 4.3 provides code for the SR equivalent. The only real difference is the use of resource capabilities as mentioned earlier.

4.1.3 Procedure Calls

Generally, simulating procedure calls using operations is not difficult. Standard procedure calls without any implicit or explicit yields can be written as Dynamic C functions. Procedure calls with yields, however, require a different approach. Typically the solution is to just write the procedure code in-line. Remote procedure calls, however, require more work. Figure 4.4 illustrates how a remote procedure call, with the limitations described later in Section 4.2.4, can be simulated. A procedure is a process, explicitly created by its parent process, which contains an infinite loop that services two operations. The first operation is the one that handles the remote procedure call. The second operation is invoked when the process that created the procedure is terminating. This second operation signals the remote procedure handling process to terminate as well. Of course, the programmer must supply the explicit call to this terminating operation. With these key elements in place, a program using DesCaRTeS can support remote procedure calls like those in SR.

```
cofunc void A(CoData * p){
        char params[RTS_MAX_STRING_LENGTH];
        char rtn[RTS_MAX_STRING_LENGTH];
        // A's operation
        OP a;
        // capability to B's operation
        CAP b;
        int i;
        RTSOpDeclare(a, CALL);
        RTSSTART_PROCESS(p);
        // build the parameter list to B
        RTSStartParams(params);
        RTSAddCap(params, a);
        // create process_B
        RTSCreate("process_B", params, rtn, -1);
        // parse return parameters, a capability to B's operation
        RTSParseCap(rtn, &b);
        // create an invocation of b
        RTSSend(b, NULL);
        // receive an invocation of a
        RTSReceive(a, params);
        RTSParseInt(params, &i);
        RTSEND_PROCESS();
}

cofunc void B(CoData * p){
        char params[RTS_MAX_STRING_LENGTH];
        char rtn[RTS_MAX_STRING_LENGTH
        CAP caparray[2];
        // B's operation
        OP b;
        // capability to B's operation
        CAP a;
        int i;
        int j;
        // declare operations and send them back to A
        RTSOpDeclare(b, SENDCALL);
        RTSSTART_PROCESS(p);
        // grab and parse the creation parameters
        RTSGetParams(params);
        RTSParseCap(params, &a);
        RTSReply();
        i = 0;
        caparray[0] = a;
        caparray[1] = b;
        while(TRUE){
                RTSInBegin(caparray, 2);
                        RTSInArmBegin(a, i == 0, params, caparray, 2);
                         // service an invocation of a, provided i == 0
                                RTSParseInt(params, &j);
                                printf("%d", j);
                        RTSInArmEnd(a, NULL);
                        RTSInArmBegin(b, TRUE, NULL, caparray, 2);
                         // service an invocation of b
                                RTSStartParams(params);
                                RTSAddInt(params, 2);
                                RTSCall(a, params, NULL);
                        RTSInArmEnd(b, NULL);
                RTSInEnd(caparray, 2);
        }
        RTSEND_PROCESS();
}
```

Figure 4.2: Sample code using operations with DesCaRTeS

```
resource A()
	import B

	op a(int) {call}

	var resB : cap B
	var i : int

	# create a resource instance of B
	resB := create B(a)

	# create an invocation of B.b
	send resB.b();

	# service an invocation of a
	receive a(i);
end
```

```
resource B
	op b()
body B(a : cap(int) {call})
	reply

	var i := 0

	do true ->
		in a(j) st i = 0 ->
		 # service an invocation of a
			write(j)
		[] b() ->
		 # service an invocation of b
			call a(2);
		ni
	od
end
```

Figure 4.3: SR code functionally equivalent to Dynamic C code in Figure 4.2

4.2 DesCaRTeS Library

This section discusses DesCaRTeS in detail. It discusses datatypes, introduces important constants, and describes the specific details of the library's functions.

The following sections give definitions of DesCaRTeS library components. Section 4.2.1 discusses new datatypes and their uses. Section 4.2.2 lists relevant constants that should be used in a program using DesCaRTeS. Section 4.2.3 describes both external and internal functions of DesCaRTeS.

4.2.1 Datatypes

DesCaRTeS offers four new datatypes that provide functionality similar to their SR equivalent [2].

- **VM**

  Used to store the index into the virtual machine table after a call to **RTSCreateVM**. This type is equivalent to an integer and is provided to maintain symmetry with SR's **vm** type.

- **OP**

  Variables of type **OP** are structures that contain three fields. The first field records the virtual machine index into the virtual machine table. This value indicates where an **OP** was declared. The second field records the physical

```
// this cofunction is the resource instance which creates the procedure
cofunc void parent_cofunc(CoData * p){
        char rtn[RTS_MAX_STRING_LENGTH];
        ...
        // remote procedure calls can be made using the proc capability
        CAP proc;
        CAP procdie;
        ...
        RTSSTART_PROCESS(p);
        ...
        // create the process to handle remote procedure calls to proc
        RTSCreate("proc_handler", NULL, rtn, -1);
        RTSParseCap(rtn, &proc);
        RTSParseCap(rtn, &procdie);
        ...
        // explicitly kill the process handling the remote procedure calls
        RTSCall(procdie, NULL, NULL);

        RTSEND_PROCESS();
}

// this cofunction handles procedure calls to proc
cofunc void proc_handler(CoData * p){
        char params[RTS_MAX_STRING_LENGTH];
        char rtn[RTS_MAX_STRING_LENGTH];
        CAP caparray[2];
        int loopstop;
        OP proc;
        OP procdie;

        // declare operations and send them back to the parent
        RTSOpDeclare(proc, SENDCALL);
        RTSOpDeclare(procdie, CALL);
        RTSSTART_PROCESS(p);
        caparray[0] = proc;
        caparray[1] = procdie;
        loopstop = FALSE;
        while(!loopstop){
                RTSInBegin(caparray, 2);
                        RTSInArmBegin(proc, TRUE, params, caparray, 2);
                                // procedure code
                                // the rtn below is any parameters
                                // the procedure may return
                        RTSInArmEnd(proc, rtn);
                        RTSInArmBegin(procdie, TRUE, NULL, caparray, 2);
                                // kill this process
                                loopstop = TRUE;
                        RTSInArmEnd(procdie, NULL);
                RTSInEnd(caparray, 2);
        }

        RTSEND_PROCESS();
}
```

Figure 4.4: Template for remote procedure calls using DesCaRTeS

address where the **OP** is stored on its respective machine. The third field records the sequence number. This number is used to determine whether or not a reference to this operation is valid.

- **CAP**

  Variables of type **CAP** are structurally equivalent to **OP** variables. This type is provided to maintain symmetry with SR's **cap** type.

- **SEM**

  Variables of type **SEM** are structurally equivalent to **OP** variables. This type is provided to maintain symmetry with SR's **sem** type. In SR, semaphores are handled as abbreviated versions of operations (see Section 2.1). DesCaRTeS, however, does no special handling of these kind of variables.

4.2.2 Constants

DesCaRTeS provides the following constants.

- **SEND**

  Used to restrict an operation to invocations using send only. This restriction is made when the operation is declared using **RTSOpDeclare**.

- **CALL**

  Used to restrict an operation to invocations using call only. This restriction is made when the operation is declared using **RTSOpDeclare**.

- **SENDCALL**

  Used to allow unrestricted invocations of an operation. This unrestricted property is assigned when the operation is declared using **RTSOpDeclare**

- **RTS_MAX_STRING_LENGTH**

  Strings used with DesCaRTeS must be less than or equal to this value. Currently set to 300, this value could be changed if enough memory is present (see Section 6.3).

- **MAX_SOCKETS**

  A macro defined within in the TCP library, this value limits the number of ethernet connections that can be made. Currently, this value is set to 4.

4.2.3 Functions

DesCaRTeS provides a number of functions that provide semantics similar to their SR equivalents.

4.2.3.1 General

DesCaRTeS itself relies on the continuous execution of a cofunction that handles communication with other machines. As illustrated in Figure 4.1, this cofunction must be explicitly called within a costatement. Furthermore, it should run parallel with all other processes on that machine.

- **void RTSInit()**

  Initialize global variables within DesCaRTeS.

- **cofunc void RTS()**

  Cofunction that is the core of DesCaRTeS. It performs a number of tasks including listening for incoming connections and processing messages from existing connections.

4.2.3.2 Virtual Machine Creation and Destruction

These functions create and destroy virtual machines.

- **VM RTSCreateVM(char * ip)**

  Establishes an ethernet connection with another machine at an address designated by **ip**. Returns an index into the array of existing connections. If no connection already exists, a new one is established. If the maximum number of connections has been reached, this function returns -1.

- **void RTSDestroyVM(VM v)**

  Terminates the virtual machine **v** and closes the ethernet connection. The machine being destroyed subsequently destroys all of the virtual machines created by local processes.

4.2.3.3 Process Handling

These functions are involved in the handling of processes, including creation and structure maintenance.

- **void RTSRegCoData(CoData * p, char * name)**

  Stores (registers) a process named **name** that is controlled by a costatement using the CoData variable referenced by **p** in a table. This information is used to simulate dynamic process creation.

- **void RTSCreate(char * name, char * params, char * rtn, VM v)**

  Creates a process registered as **name** with the parameters **params** on virtual machine **v**. Return parameters are copied to **rtn**.

- **void RTSSTART_PROCESS(CoData * p)**

  Initializes the process controlled by the costatement using the CoData variable referenced by **p**.

- **void RTSEND_PROCESS()**

  Terminates a process. Calling this functions allows the process to be created again later.

4.2.3.4 Operations and Process Communication

The functions used to implement this communication are described below.

- **void RTSOpDeclare(OP o, int call)**

  Declares the operation designated by **o** by storing it in an operation table maintained by DesCaRTeS. The invocations of **o** are restricted according to the value given by **call**. **call** should be given one of the constants **SEND**, **CALL**, or **SENDCALL**.

- **void RTSSend(CAP c, char * params)**

  Sends to an operation designated by the capability **c** the parameters in the string **params**.

- **void RTSCall(CAP c, char * params, char * rtn)**

  Performs a call to an operation designated by the capability **c** using the string **params** as its parameters. Any return parameters will be stored in the string **rtn**. Limitations to the function are the same as those described for **RTSSend**.

- **void RTSReceive(CAP c, char * params)**

  Receives an invocation made on an operation referenced by capability **c**. All parameters for that invocation are copied to the string **params**.

- **void RTSInBegin(CAP * caparray, int numcaps)**

  Designates the beginning of an input statement. **caparray** is an array of capabilities that are serviced by this input statement. **numcaps** is the number of capabilities in the array.

- **void RTSInArmBegin(CAP c, int clause, char * params, CAP * caparray, int numcaps)**

  Designates the beginning of an arm in an input statement. **c** is the capability serviced by this arm. **clause** is a boolean valued expression. This arm is only serviced if **clause** evaluates to true. All parameters for the invocation received are copied to the string **params**. **caparray** and **numcaps** are the same as described above.

- **void RTSInArmEnd(CAP c, char * rtn)**

  Designates the end of an arm in an input statement. This function terminates the service to a capability **c** and replies to the invoker with the parameters in the string **rtn**.

- **void RTSInElseBegin()**

  Designates the beginning of an optional else arm in an input statement. If all arms of the input statement cannot be serviced, then this arm executes.

- **void RTSInElseEnd()**

  Designates the end of an optional else arm in an input statement.

- **void RTSInEnd(CAP * caparray, int numcaps)**

  Designates the end of an input statement. **caparray** and **numcaps** are the same as described above.

- **void RTSReply(char * rtn)**

  Replies to the process on top of the reply stack. This routine can be used to reply to parent processes or to the invoker of an operation being serviced by an input statement. **rtn** holds any parameters being passed back with the reply.

4.2.3.5 Message Processing

These functions provide a clean interface for message processing.

- **void RTSStartParams(char * params)**

  Initializes the string designated by **params**. It just clears the string by setting it to the empty string.

- **void RTSGetParams(char * params)**

  Retrieves the parameters passed to a process after it was created. These parameters are stored in the string designated by **params**.

- **void RTSAddInt(char * params, int i)**

  Appends a string representation of an integer **i** to the end of the parameter list designated by the string **params**. Appears as "_INT(i)" in the string.

- **void RTSAddReal(char * params, float f)**

  Appends a string representation of a real **f** to the end of the parameter list designated by the string **params**. Appears as "_REAL(f)" in the string.

- **void RTSAddChar(char * params, char c)**

  Appends a string representation of a character **c** to the end of the parameter list designated by the string **params**. Appears as "_CHAR(c)" in the string.

- **void RTSAddString(char * params, char * s)**

  Appends a string representation of a string **s** to the end of the parameter list designated by the string **params**. Appears as "_STRING(s)" in the string.

- **void RTSAddLongInt(char * params, long int l)**

  Appends a string representation of a long integer **l** to the end of the parameter list designated by the string **params**. Appears as "_LONGINT(l)" in the string.

- **void RTSAddCap(char * params, CAP c)**

  Appends a string representation of a capability **c** to the end of the parameter list designated by the string **params**. Appears as "_CAP(c)" in the string.

- **void RTSParseInt(char * params, int * i)**

  Parses an integer from the front of the parameter list designated by the string **params**. It first ensures that the front of the **params** begins with "_INT(", then parses up to the first ')' found. This value is converted and stored in the integer pointed to by **i**.

- **void RTSParseReal(char * params, float * f)**

  Parses a real from the front of the parameter list designated by the string **params**. It first ensures that the front of the **params** begins with "_REAL(", then parses up to the first ')' found. This value is converted and stored in the real pointed to by **f**.

- **void RTSParseChar(char * params, char * c)**

  Parses a character from the front of the parameter list designated by the string **params**. It first ensures that the front of the **params** begins with "_CHAR(", then parses up to the first ')' found. This value is converted and stored in the character pointed to by **c**.

- **void RTSParseString(char * params, char * s)**

  Parses a string from the front of the parameter list designated by the string **params**. It first ensures that the front of the **params** begins with "_STRING(", then parses up to the first ')' found. This value is stored in the string pointed to by **s**. See Section 4.2.4 for limitations on **s**.

- **void RTSParseLongInt(char * params, long int * l)**

  Parses a long integer from the front of the parameter list designated by the string **params**. It first ensures that the front of the **params** begins with "_LONGINT(", then parses up to the first ')' found. This value is converted and stored in the long integer pointed to by **l**.

- **void RTSParseCap(char * params, CAP * c)**

  Parses a capability from the front of the parameter list designated by the string **params**. It first ensures that the front of the **params** begins with "_CAP(", then parses up to the first ')' found. This value is converted and stored in the capability pointed to by **c**.

4.2.3.6 Semaphore Operations

Semaphores and their related operations are also provided by DesCaRTeS.

- **RTSSemInit(SEM s, int val)**

  Initializes a semaphore **s** to the value designated by **val**. This function must be called for each semaphore.

- **RTSP(SEM s)**

  Performs a **P** operation on a semaphore **s**.

- **RTSV(SEM s)**

  Performs a **V** operation on a semaphore **s**.

4.2.3.7 Memory Management

Dynamic C does not provide a method to dynamically allocate and deallocate memory. Thus, versions of malloc and free were implemented for use by DesCaRTeS. Both of these functions were developed using a First Fit algorithm designed at Princeton [7]. This algorithm has the advantages of being both small and simple, thus helping to minimize the amount of memory used for DesCaRTeS.

- **XMEM_ADDR RTSmalloc(unsigned nbytes)**

  Allocates the number of bytes of memory designated by **nbytes**. Returns the physical address of the chunk of memory that was allocated. This function uses the Dynamic C memory allocator **xalloc**(**xalloc** is not used directly since it behaves more like the system call **sbreak** and thus leaves no method for deallocating memory). **xalloc** is only called when the amount of memory requested is currently unavailable to **RTSmalloc**. This occurs when there is no memory currently allocated to **RTSmalloc** or all free blocks managed by **RTSmalloc** are too small.

- **void RTSfree(XMEM_ADDR ptr)**

  Deallocates the memory pointed to by **ptr**. It also checks to see if the newly deallocated chuck of memory is next to one or more unallocated blocks. If so, these blocks of memory are combined into a single block.

4.2.3.8 Internal Functions

These functions are used by DesCaRTeS internally. They provide a wide range of processing including invocation queue handling, blocking queue handling, and process id look-up.

- **void RTSClearParams(XMEM_ADDR, paddr)**

  Clears the parameter field of the process designated by **paddr**. Sets the value of that field to the empty string.

- **XMEM_ADDR RTSGetpid(CoData * p)**

Searches the process table for the process designated by **p**. Once found, returns the physical address of that entry in the table. This address represents the process id of **p**.

- **void RTSOrderInv(CAP * caparray, int numcaps, XMEM_ADDR paddr)**

  Sorts **caparray** by timestamps of invocations of all the operations referenced by the capabilities in **caparray**. **numcaps** designates the number of capabilities in **caparray**. **paddr** is the process id of the process waiting for an invocation. If only one operation is referenced, then the operation may be remote. Remote operations are sent a request by this function for the oldest invocation available. The process designated by **paddr** is then blocked until a reply has been received.

- **void RTSRemoveInv(CAP * caparray, char * rtn, XMEM_ADDR paddr, int numcaps)**

  Removes the invocation designated by **RTSGetInv**. This is called once DesCaRTeS confirms that the invocation will be serviced. **caparray** designates the operations serviced by the input statement. **rtn** points to the string to which the parameters of an invocation should be copied. **paddr** is the process id of the process servicing the invocation. **numcaps** is the number of operations serviced by the input statement.

- **void RTSAddBlocking(CAP * caparray, int numcaps, XMEM_ADDR paddr)**

  Adds the process designated by **paddr** to the blocking queues of all operations referred to by **caparray**. **numcaps** designates the number of operations in **caparray**.

- **void RTSRemoveBlocking(CAP * caparray, int numcaps, XMEM_ADDR paddr)**

  Removes the process designated by **paddr** from the blocking queues of all operations referred to by **caparray**. **numcaps** designates the number of operations in **caparray**. If more invocations for a particular operation exist, any blocking processes for those operations are restarted.

- **void RTSStartNextBlocking(CAP * c, XMEM_ADDR paddr)**

  For a particular operation designated by **c**, restarts the next blocked process on the blocking queue after the process designed by **paddr**. This is called when an invocation of **c** goes unserviced due to a false such-that clause in the arm of an input statement. Used internally by **RTSInArmEnd**, this procedure ensures that at least one process blocking on **c** is active when one or more invocations exists for **c**.

- **void RTSCheckSocket(VM v)**

  Checks, parses, and processes any available messages from a virtual machine **v**.

4.2.4 Limitations

DesCaRTeS has a few limitations compared to its SR counterpart. Many of the difficulties in implementing DesCaRTeS resulted from the fact that the library did not have the benefit of gathering information at compile-time. This prevented DesCaRTeS from supporting certain features that ensure correct semantics. This section discusses the limitations of DesCaRTeS and proposes possible solutions as future work.

Due to DesCaRTeS's run-time nature, the library is unable to perform rigorous type checking of parameters to operation invocations. This is primarily why operations lack signatures as in SR. Although some type checking occurs when parameters are parsed (by checking the header of each parameter as discussed in Section 4.2.3.5), rigorous type checking of the actual parameters is not present. Such a feature would be time consuming to implement and was not the goal of this research. Signatures for operations could be implemented by adding an additional field to operations in the operation table. This field could contain a string that designates the appropriate sequence of parameters for an invocation. Upon an invocation of an operation, the string containing the parameters could be checked with this signature.

Another limitation of DesCaRTeS is the lack of support for parameter dependent such-that clauses for input statements (see Section 2.1). Unlike SR's **st** clause, this expression cannot be dependent on the values received by an invocation of the operation being serviced. This is because messages are received as strings and must be parsed before the parameters can be used. The difficulty here lies in the fact that the programmer is required to supply the necessary sequence of function calls to parse the parameters for an invocation. Since DesCaRTeS does not know what to expect for parameters, it cannot parse the parameters of an invocation. Even if DesCaRTeS had the signatures discussed above to aid in this parsing, the macros used to define input statements currently do not support this feature. The structure of these macros would need complete revision to support this feature.

Another drawback of DesCaRTeS comes from the fact that Dynamic C does not support dynamic process creation. As Figure 4.1 revealed, the programmer must declare ahead of time the number of processes expected to be created. Remote procedure calls also suffer from this limitation. In SR, a capability may refer to an operation serviced by either an input statement (or receive statement) or a remote procedure call. DesCaRTeS supports both servicing mechanisms, but restricts remote procedure calls. The inability of Dynamic C to create processes dynamically disallows an unknown number of simultaneous calls to a procedure to be made. Thus, operations serviced by procedures must not contain any explicit or implicit yields. Doing so could eventually leave the system in deadlock.

Currently, DesCaRTeS does not provide resource capabilities. As discussed in Section 4.1.2, capabilities to process operations are passed implicitly back to the creator during operation declarations. Although resource capabilities provide a nice interface for using operations of newly created resources, they were not required to achieve the goals of this research.

Due to time constraints, destruction of processes and virtual machines has not been fully implemented. **RTSDestroyVM** does close the connection with designated virtual machine. However, the virtual machine is also completely shutdown, preventing the program from creating the virtual machine again later in the program. If the machine was left running, all processes, operations, invocations, and any other component using memory besides the RTS itself would have to be destroyed. All relevant capabilities to operations that are removed would have to be invalidated. This includes those capabilities passed to other virtual machines. Since these features would require extensive time to complete and were not necessary for this research, they were not included in the current version of DesCaRTeS.

As mentioned in Section 4.2.3.5, strings used with DesCaRTeS have a limitation. Specifically, strings must not contain ')'. Such strings would cause incorrect parsing of a message and would cause a program to crash. Additional code could be introduced to remove this limitation, but such a feature was not necessary for this research and thus was saved for future work.

Chapter 5

Experimental Performance Results

This chapter presents the results of experiments conducted on the load balancing and non-load balancing models discussed in Chapter 3. Both SR and Dynamic C implementations were tested.

5.1 Experimental Setup

This section describes the general setup for the experiments including a description of the parameters that define each test. The parameters discussed provide the same functionality in both DDP and DRW for both SR and Dynamic C versions.

Processes in all experiments are executed on three different processors. One processor handles process creation, process synchronization, and load balancing. The other two processors represent regions as described in Sections 3.1 and 3.2.

Both the DDP and DRW experiments used eight parameters as defined in Table 5.1. Figure 5.1 illustrates the use of NON_CS_WORK, CS_WORK, and WORK with code fragments written in SR. The loop bounded by WORK is nested within both loops bounded by NON_CS_WORK and CS_WORK. Under PCM, an explicit context switch is placed immediately after this inner loop finishes. This context switch makes PCM more comparable to CP. Without it, PCM would simply execute all work loops until completion. CP, however, would perform implicit context switches for WORK values greater than or equal to 10000. This is the value for which the SR implementation performs implicit context switches (see Section 2.1).

Parameter	Description
N	Limit on the number of processes allowed in a region.
TEST_NUM	Number of processes created during a test. Test completes when all of these processes terminate.
ITER	Upper bound on the number of iterations a process executes during its lifetime. Actual value is randomly selected between 1 and this value.
NON_CS_WORK	Number of iterations the loop in the non-critical section of a process's code executes.
CS_WORK	Number of iterations the loop in the critical section of a process's code executes.
WORK	Upper bound of inner loop. Under CM, context switches appear after this loop.
CREATE_TIME	The upper bound on the number of seconds between process creations. Actual time is randomly selected between 1 and this value.
BALANCE_TIME	Time, in seconds, between load balancing attempts.

Table 5.1: Parameter definitions for SR experiments

Two sets of experiments using different parameters were conducted for both DDP and DRW. Table 5.2 shows the values of the parameters used for each experiment.

5.2 SR Implementations

Three Pentium class machines were used for the SR tests. Initial tests revealed that the programs running on relatively equivalent machines remained balanced during the duration of their executions. Thus, to provide an environment that required load balancing, both regions for each experiment were placed on machines with significantly different performance capabilities. Region 1 was run on a 200 MHz Pentium Pro. Region 2 was run on a 1400 MHz Pentium 4. The main resource, which included the load balancer, was run on an 866 MHz Pentium 3. These machines run Red Hat Linux 7.0 communicating on a 100 Mb ethernet network.

Parameter	Experiment 1	Experiment 2
N	20	20
TEST_NUM	50	50
ITER	100	1000
NON_CS_WORK	100	10
CS_WORK	100	10
CREATE_TIME	10	20
BALANCE_TIME	2.5	5

Table 5.2: Parameters for SR experiments

```
# iter is a randomly generated integer
# between 1 and ITER
do iter > 0 ->
        ...
        # think
        fa i := 1 to NON_CS_WORK ->
                fa j := 1 to WORK -> af
                nap(0) # PCM version only
        af
        ...
        # grab forks
        # eat
        fa i := 1 to CS_WORK ->
                fa j := 1 to WORK -> af
                nap(0) # PCM version only
        af
        # release forks

        iter--;
od
```

(a) DDP philosopher

```
# iter is a randomly generated integer
# between 1 and ITER
do iter > 0 ->
        ...
        # local work
        fa i := 1 to NON_CS_WORK ->
                fa j := 1 to WORK -> af
                nap(0) # PCM version only
        af
        ...
        # gain access to database
        # dbwork
        fa i := 1 to CS_WORK ->
                fa j := 1 to WORK -> af
                nap(0) # PCM version only
        af
        # release access to database

        iter--;
od
```

(b) DRW worker

Figure 5.1: Code fragments in SR illustrating use of test parameters for DDP and DRW

WORK	DPLBsmall/ DPNLB (PCM)	DPLBlarge/ DPNLB (PCM)	DPLBsmall/ DPNLB (CP)	DPLBlarge/ DPNLB (CP)
100	0.7924	0.8362	0.8156	0.7584
1000	0.9822	1.0678	1.0001	0.9545
10000	0.9238	0.8934	0.8827	0.7597
100000	0.6437	0.6317	0.6451	0.6725
1000000	0.7693	0.6832	0.7466	0.6925

Table 5.3: Performance ratios for DDP using load balancing (Experiment 1)

WORK	DPLBsmall/ DPNLB (PCM)	DPLBlarge/ DPNLB (PCM)	DPLBsmall/ DPNLB (CP)	DPLBlarge/ DPNLB (CP)
100	0.8415	0.6935	0.7918	0.8716
1000	0.9624	1.0561	1.0069	0.9364
10000	1.0015	1.1291	0.9160	0.9771
100000	0.5543	0.5989	0.5425	0.5045
1000000	0.5355	0.5021	0.5731	0.5447

Table 5.4: Performance ratios for DDP using load balancing (Experiment 2)

5.2.1 Load Balancing

This section presents the results of DDP and DRW when comparing load balancing and non-load balancing models.

5.2.1.1 DDP

Figures A.1, A.2, A.3 and A.4 (in Appendix A) present data for DDP. All axes use a log scale. **DPLBsmall** and **DPLBlarge** both perform better than **DPNLB** for larger workloads (see Section 3.1.2 for definitions of models). That is, in these tests, load balancing improves performance despite its extra overhead. However, there is only a negligible difference in performance between the two load balancing models. Tables 5.3 and 5.4 show the performance ratios of load balancing models over the non-load balancing model for both experiments.

The trends illustrated by the data can be explained as follows. Let C be the average time in milliseconds between philosopher creations. Now let $T = C*N$, where N is defined as in Table 5.1. If the average lifetime of a philosopher is much less than T, then the system generally has few, if any, philosophers present at any time. In this state, illustrated by workloads of 100 and 1000, load balancing is ineffective since there is not enough workload to cause the system to become uneven. If the average lifetime is about T, then the system will have a moderate amount of philosophers present. This state occurs for workloads between 10000 and 100000. Moderate loads benefit from load balancing since a large enough workload is present to benefit from redistribution, but the number of philosophers alive is not so great that the regions become full. For average lifetimes much greater than T, the system becomes heavily impacted (ie, each region typically has N philosophers present at any given time), leaving no room for the load balancer to move philosophers across regions. In this state, shown by WORK values between 100000 and 1000000, load balancing loses its effectiveness. Gains for these values are achieved only during the beginning and ending stages of the experiment, when the number of philosophers is just starting to grow or diminish, respectively.

Successful load balancing is defined as the load balancer attempting and accomplishing a move of one process from one region to another. More specifically, the load balancer first evaluates whether or not moving a process would yield a more balanced system. Then, the load balancer checks to make sure that the destination region of a process migration has enough available space to hold another process. When both of these conditions are met, the balancer proceeds to move a process. Once started, a process migration never fails since a slot in the destination region is reserved before any other processes are created or destroyed. Load balancing failure occurs when the destination region contains the maximum number of processes (defined by **N**) or the system is already balanced. In this case, the load balancer aborts and retries again later. Tables 5.5 and 5.6 show the percentages of successful load balancing on all load balancing DDP tests. These percentages are calculated as the the number of actual process migrations over the total number of load balancer attempts and multiplied by 100. These percentages illustrate the success of load balancing under different workloads. Generally, load balancing is more successful for larger workloads. The sudden drop in these percentages for WORK values of 1000000, however, can be explained

WORK	DPLBsmall (PCM)	DPLBlarge (PCM)	DPLBsmall (CP)	DPLBlarge (CP)
100	9.33	8.64	0.95	0.35
1000	10.95	12.71	1.59	2.04
10000	24.69	36.40	23.63	38.19
100000	26.27	31.58	14.66	22.83
1000000	4.84	5.46	2.09	3.25

Table 5.5: SR DDP load balancing success percentages (Experiment 1)

WORK	DPLBsmall (PCM)	DPLBlarge (PCM)	DPLBsmall (CP)	DPLBlarge (CP)
100	14.20	21.24	18.49	17.61
1000	12.04	10.80	11.76	17.02
10000	23.18	20.18	27.80	23.96
100000	26.42	43.02	22.63	39.82
1000000	5.21	11.02	2.56	6.32

Table 5.6: SR DDP load balancing success percentages (Experiment 2)

in two ways. First, region populations are restricted to 20 philosophers. When the WORK increases to large enough values, the system becomes full. Thus, the load balancer still attempts to execute, but fails due to unavailable room to complete a move. Second, only a limited number of load balancing moves can be made during the execution. In our experiments Region 1 runs more slowly than Region 2. Thus, most philosophers are moved from Region 1 to Region 2. In the worst case (in which all philosophers are created on Region 1), the balancer would succeed at most 49 times. But since the workload is higher, the experiment will take longer to run. This causes the load balancer to execute more times. Thus, the percentage of successful load balancing will drop. Of course, a few philosophers are moved from Region 2 to Region 1. Thus, a realistic estimate for the number of load balancing successes could exceed 50.

WORK	RWLBbynum/ RWNLB (PCM)	RWLBbywork/ RWNLB (PCM)	RWLBbynum/ RWNLB (CP)	RWLBbywork/ RWNLB (CP)
100	0.8286	0.9257	0.9928	1.0551
1000	0.9090	0.8295	0.9525	0.9803
10000	1.0861	1.0698	0.9818	0.9430
100000	1.0056	0.8711	0.9703	1.0372
1000000	0.8855	0.8668	0.9601	0.8486

Table 5.7: SR performance ratios for DRW using load balancing (Experiment 1)

5.2.1.2 DRW

Figures A.5, A.6, A.7, and A.8 present data for DRW. The x-axes representing WORK use a log scale. Tables 5.7 and 5.8 show the performance ratios of load balancing models over the non-load balancing model for both experiments. **RWLBbywork** outperformed both **RWLBbynum** and **RWNLB** (see Section 3.2.2 for definitions of models). **RWLBbywork** achieved performance gains much like the load balancing models for DDP (see Section 5.2.1.1), although the percentage of improvement is significantly less. **RWLBbynum**, however, seemed to have similar results as the non-load balancing model **RWNLB**. Since readers and writers are created and distributed randomly, the processes will generally be evenly distribution. Thus, **RWLBbynum** fails to load balance most of the time. This is reflected by Tables 5.9 and 5.10, which show the percentages of successful load balancing on all load balancing DRW tests. The reason for the decline of these percentages for WORK values of 1000000 is similar to that described for DDP in Section 5.2.1.1.

5.2.2 PCM versus CP

Tables 5.11, 5.12, 5.13, and 5.14 illustrate the performance ratios of PCM versus CP. For smaller WORK values, CP and PCM have similar performance. For larger WORK values, PCM outperformed CP. More specifically, this gain in performance begins at some WORK value greater than 10000. This can be explained by the frequency of implicit context switches in SR, which occur every 10000 iterations of a loop (see Section 2.1). Thus, programs written in PCM will make fewer context

WORK	RWLBbynum/ RWNLB (PCM)	RWLBbywork/ RWNLB (PCM)	RWLBbynum/ RWNLB (CP)	RWLBbywork/ RWNLB (CP)
100	0.8803	0.9356	0.8387	1.0921
1000	0.9743	0.9321	0.9497	0.9330
10000	1.1468	0.9762	0.9784	1.0180
100000	1.2222	0.9541	0.9814	0.9072
1000000	0.8590	0.9098	0.8991	0.8832

Table 5.8: SR performance ratios for DRW using load balancing (Experiment 2)

WORK	RWLBbynum (PCM)	RWLBbywork (PCM)	RWLBbynum (CP)	RWLBbywork (CP)
100	1.02	1.82	0.34	0.63
1000	0.34	2.58	0.34	0.00
10000	5.26	8.83	8.44	9.80
100000	3.79	12.03	3.11	5.22
1000000	0.45	1.00	0.21	0.48

Table 5.9: SR DRW load balancing success percentages (Experiment 1)

WORK	RWLBbynum (PCM)	RWLBbywork (PCM)	RWLBbynum (CP)	RWLBbywork (CP)
100	4.63	6.29	3.86	6.94
1000	2.92	10.17	3.25	9.18
10000	5.62	8.59	5.32	12.74
100000	7.38	22.83	5.93	13.23
1000000	0.68	1.17	0.24	0.46

Table 5.10: SR DRW load balancing success percentages (Experiment 2)

WORK	DPNLB	DPLBsmall	DPLBlarge
100	0.8830	0.8578	0.9735
1000	0.9282	0.9115	1.0384
10000	0.8609	0.9010	1.0124
100000	0.6169	0.6155	0.5795
1000000	0.5988	0.6170	0.5908

Table 5.11: PCM vs. CP ratios for DDP (Experiment 1)

WORK	DPNLB	DPLBsmall	DPLBlarge
100	1.0203	1.0843	0.8117
1000	1.0212	0.9761	1.1518
10000	0.9338	1.0209	1.0791
100000	0.6306	0.6443	0.7485
1000000	0.6604	0.6170	0.6088

Table 5.12: PCM vs. CP ratios for DDP (Experiment 2)

switches than their CP counterpart for WORK values exceeding 10000. That results in less total CPU time used to perform context switches and more CPU time used to perform real work.

5.3 Dynamic C Implementations

Three Rabbit 2000 TCP/IP Development Boards were used for the experiment. Each processor has an 18 MHz clock speed and communicates on a 10 Mb ethernet network. Two sets of experiments using different parameters were conducted. Ta-

WORK	RWNLB	RWLBbynum	RWLBbywork
100	1.0724	0.8951	0.9409
1000	1.0457	0.9978	0.8848
10000	0.9439	1.0441	1.0708
100000	0.6389	0.6622	0.5366
1000000	0.6133	0.5656	0.6265

Table 5.13: PCM vs. CP ratios for DRW (Experiment 1)

WORK	RWNLB	RWLBbynum	RWLBbywork
100	1.0046	1.0544	0.8606
1000	0.9707	0.9959	0.9698
10000	0.9059	1.0619	0.8687
100000	0.5440	0.6775	0.5721
1000000	0.5201	0.4969	0.5358

Table 5.14: PCM vs. CP ratios for DRW (Experiment 2)

Parameter	Experiment 1	Experiment 2
N	10	10
TEST_NUM	30	30
ITER	10	100
NON_CS_WORK	100	10
CS_WORK	100	10
CREATE_TIME	10	20
BALANCE_TIME	2.5	5

Table 5.15: Parameters for Dynamic C DDP experiments

ble 5.15 shows the parameter values used for each DDP experiment. Table 5.16 shows the parameter values used for each DRW experiment. Section 5.1 gives a description of each of these parameters. The values used for these experiments are smaller than those for SR (see Section 5.2) because of the differences in processor speeds and the limited amounts of memory. Both sets of experiments are run stand-alone.

5.3.1 Load Balancing

5.3.1.1 DDP

Figures A.9 and A.10 present data for DDP. The x-axes representing WORK uses a log scale. Tables 5.19 and 5.20 show the performance ratios of load balancing versus non-load balancing models. In some cases, load balancing has only a small amount of benefit for smaller amounts of work. But as workload increases and the system begins to reach maximum capacity, load balancing gradually becomes less effective and begins to hinder performance due to overhead. In other cases, as in Experiment

Parameter	Experiment 1	Experiment 2
N	5	5
TEST_NUM	20	20
ITER	10	100
NON_CS_WORK	100	10
CS_WORK	100	10
CREATE_TIME	10	20
BALANCE_TIME	2.5	5

Table 5.16: Parameters for Dynamic C DRW experiments

WORK	$DPLBsmall$	$DPLBlarge$
1	12.86	0.00
10	16.67	3.24
50	10.16	7.56
100	8.10	1.51

Table 5.17: Dynamic C DDP load balancing success percentages (Experiment 1)

1, increased work per iteration causes synchronization between philosophers and the load balancer to take longer amounts of time. This is due to the longer periods of time between possible times of synchronization, which occur only once per iteration. Thus, load balancing is never effective, and has a larger negative impact on performance for larger WORK values.

Tables 5.17 and 5.18 show the percentages of successful load balancing on all load balancing DDP tests. These percentages peak somewhere between WORK values of 10 and 50, and then begin to drop off. The reason for this drop off is explained in Section 5.2.1.1.

WORK	$DPLBsmall$	$DPLBlarge$
1	13.67	1.14
10	28.86	11.22
50	9.94	13.14
100	7.60	6.15

Table 5.18: Dynamic C DDP load balancing success percentages (Experiment 2)

WORK	DPLBsmall/ DPNLB	DPLBlarge/ DPNLB
1	1.0324	1.0405
10	1.4411	1.1408
50	1.4293	1.0829
100	1.3729	1.2999

Table 5.19: Dynamic C performance ratios for DDP using load balancing (Experiment 1)

WORK	DPLBsmall/ DPNLB	DPLBlarge/ DPNLB
1	0.8808	0.9852
10	0.9840	0.8578
50	1.2073	0.9625
100	1.2000	1.2902

Table 5.20: Dynamic C performance ratios for DDP using load balancing (Experiment 2)

5.3.1.2 DRW

Figures A.11 and A.12 present data for DRW. The axes representing WORK use a log scale. Tables 5.23 and 5.24 show the performance ratios of load balancing versus non-load balancing models. Both load balancing models increase performance as workload increases. **RWLBbynum**, however, performs slightly better than **RWLBbywork**. Performance gains are achieved for smaller values of WORK when processes spend more time in CS_WORK and NON_CS_WORK loops and perform fewer iterations. This is because processes that perform fewer iterations access the database less often, causing less interprocess communication and synchronization to occur. The reduction of these execution characteristics allow processes to work more quickly, and thus respond to load balancing synchronization messages more rapidly.

Tables 5.21 and 5.22 show the percentages of successful load balancing on all load balancing DRW tests. These percentages seem to peak somewhere between WORK values of 10 and 50. The drop off that follows can be explained as in Section 5.2.1.1.

WORK	RWLBbynum	RWLBbywork
1	0.00	0.00
10	5.88	14.14
50	15.62	26.79
100	11.70	14.86

Table 5.21: Dynamic C DRW load balancing success percentages (Experiment 1)

WORK	RWLBbynum	RWLBbywork
1	2.75	7.27
10	7.65	18.28
50	2.12	5.30
100	0.71	2.78

Table 5.22: Dynamic C DRW load balancing success percentages (Experiment 2)

WORK	RWLBbynum/ RWNLB	RWLBbywork/ RWNLB
1	1.0188	1.0188
10	1.0141	1.0141
50	0.7839	0.9464
100	0.6860	0.9606

Table 5.23: Dynamic C performance ratios for DRW using load balancing (Experiment 1)

WORK	RWLBbynum/ RWNLB	RWLBbywork/ RWNLB
1	0.9966	1.0069
10	0.9060	0.9382
50	1.0364	1.1268
100	0.8857	0.8884

Table 5.24: Dynamic C performance ratios for DRW using load balancing (Experiment 2)

Chapter 6

Discussion

This chapter discusses the results presented in Chapter 5. It also comments on differences between CM and CP style programming and differences between SR and Dynamic C and their respective run-time systems.

6.1 Load Balancing

Load balancing is beneficial on moderately loaded systems where the workload per processor becomes uneven. Load balancing has negligible effectiveness on systems with light workloads, since little if any balancing can be done. On heavily loaded systems with limits on regional load carrying capacities, load balancing is ineffective and can even slow the system down. In such a case, no room exists to move processes from one region to another, but additional CPU time, messages, and network traffic are introduced by the load balancing algorithm.

The frequency of load balancing attempts should also be considered when developing a load balancing model. Too many attempts will incur excessive overhead with minimal gains resulting from any additional process migrations. Too few attempts will not be effective since the system will spend more time in unbalanced states. Choosing an optimal time interval to run load balancing is system dependent and requires extensive study of the workload status of each processor during program execution.

For load balancing to be effective in the applications examined, processes must perform CPU intensive tasks. Early tests revealed that when processes spend a majority of their lives idle, load balancing had negligible effects. When processes are told to sleep for a given amount of time, that time passes at the same rate on all processors. Thus, processes that spend the majority of their lives idle have execution times that are dominated by the time spent sleeping. This causes the processors, regardless of speed, to appear to execute processes at the same rate. Processes need also to be present in the system for long enough so that load balancing can redistribute the workload. If processes live short lives with high CPU intensive work, and the system on average has a large number of these processes present at any given time, then the load balancer cannot gather accurate information about the state of the system. The load balancer will try and move processes that have terminated (or are about to). This will significantly reduce the performance of the system since being moved prolongs the lives of processes.

Load balancing seems to be equally effective under both CM and CP. That is, as seen in Section 5.2.2, speed-up percentages due to load balancing for both experiments are nearly equivalent. Load balancing does not perform any work that would take advantage of CM style programming. Load balancing code executes until completion unless it is waiting to synchronize with another process. Both styles of programming execute this code in the same manner, thus producing similar results.

Load balancing effectiveness over different applications, however, does vary. In SR experiments, DDP benefitted more from load balancing than DRW did due to the differences between the two classic problems. In DDP, no one philosopher can stall the system every time it eats*. DRW, however, is different. One writer, regardless of which region it lies on, can prevent all processes in the system from entering the critical section. While this is happening, all processes could potentially sit idle while waiting for their turns. This bottleneck occurs every time a writer enters the critical section and can cause a faster processor to execute at the same rate as the slower processor. A similar comparison in Dynamic C, however, cannot be made due to the differences in the setup of the experiments. DDP experiments allowed more processes

*The only philosopher that can stall the entire system is the asymmetric philosopher mentioned in Section 3.1.1. This state is reached only when the asymmetric philosopher is eating and all other philosophers are involved in a near circular wait condition that is broken when the asymmetric philosopher puts down its forks. This state, however, is rarely reached.

per region and tested for a larger number of total processes created than DRW. This would make load balancing performance on DDP and DRW incomparable.

Load balancing performance can also vary between different implementations. As discussed in Section 6.3, differences in Dynamic C, SR, and their respective run-time systems can cause algorithms to perform differently.

6.2 CM vs. CP

This section compares CM to CP. It examines the general performance of CM compared to CP and discusses the advantages and disadvantages of program development using CM as an alternative to CP.

6.2.1 Performance

In general, CM outperforms CP when programs contain loops that execute for long periods of time. Under CM, longer loops may have fewer context switches (if any) than under CP. This allows processes to perform more work before giving up the processor. The end result yields fewer context switches during the execution of a program.

Fairness, however, must be considered when comparing CM to CP. Although CM performs the same amount of work as CP in less time, individual processes under CM spend more time waiting for the CPU than in CP. Of course, allowing processes to execute to completion or point of synchronization would yield the best results. But having processes spend long periods of time idle is undesirable if not unacceptable in some scenarios. Multitasking systems that service multiple users simultaneously, for example, must divide up the CPU time appropriately among the various users.

6.2.2 Program Development

Developing programs using CM has some advantages and some disadvantages when compared to CP. Although CM generally needs less overhead than CP to ensure properties such as mutual exclusion, a programmer must be careful to guarantee correct program execution under CM.

When developing large programs or libraries with multiple threads under CP, programmers often must use some method to guarantee exclusive access to variables stored in shared memory, e.g., semaphores, locks, or monitors. Programs written in CM, however, do not need such protection. Code segments that require exclusive access simply execute, without context switches, until they leave a critical section. As a result, libraries like DesCaRTeS (see Chapter 4) do not require additional code to provide exclusive access to shared memory. For example, the invocation queues, blocking queues, operation table, and global variables within DesCaRTeS could all be accessed by multiple threads simultaneously. All operations on these resources, however, do not yield until after completion of critical sections and are essentially atomic in nature. This was an advantage during the development of the library, making the algorithms much simpler to implement.

Programming under CM, however, does have its disadvantages. It may be difficult, for example, to know exactly when and where to include a yield. Failure to place these yields in certain areas may result in starvation. More specifically, a process may enter a particular state and loop indefinitely while waiting for a certain condition, set by another thread, to become true. If a yield is not present inside the loop, then no other process will execute and the program will fail to make progress. Yields may also be poorly placed, causing the program to enter a faulty state. Introducing a yield inside a critical section that relies on CM for mutual exclusion instead of condition testing would eliminate any guarantee of mutual exclusion. Thus, a programmer using CM has the responsibility of placing yields in appropriate locations that prevent starvation or deadlock, but guarantee mutual exclusion for critical sections.

6.3 SR vs. Dynamic C

As the results revealed, data between SR implementations and Dynamic C implementations was not always consistent. This can be explained through the differences of the two languages and the implementation of SR's run-time system and Dynamic C's DesCaRTeS. In addition, the algorithms for DDP and DRW on SR and Dynamic C had some small, and possibly significant, differences. Moreover, the hardware used for SR and Dynamic C are considerably different. This section discusses these differences and how they might affect the results.

Surprisingly, SR is more dynamic than Dynamic C. Specifically, SR provides dynamic creation of operations, resources, remote procedure calls, and processes. SR has this capability since it is able to dynamically allocate memory. Dynamic C, on the other hand, does not provide such features. The number of processes, variables, and other memory dependent components of a program must be known at compile-time. This is primarily due to Dynamic C's inability to manage memory. Although Dynamic C provides a memory allocation function, **xalloc**, no memory deallocation function exists. Memory management was not provided for Dynamic C since memory on the Rabbit development boards is limited to only 256K for program code and data. Although DesCaRTeS provides dynamic memory allocation for use by the run-time system, dynamic process creation and other dynamic memory allocation features are still not supported by the compiler.

Another difference between SR and Dynamic C is the implementations of their run-time systems. DesCaRTeS uses many data structures and algorithms similar to those in SR's run-time system. But due to the compile-time nature of SR's run-time system and run-time nature of DesCaRTeS, some differences exist (see Section 4.2.4). In addition, access to memory allocated dynamically takes longer for DesCaRTeS: all allocated memory must be read or written using explicit function calls. This can occur frequently, since all entries into invocation lists, blocking lists, and operation tables are created with dynamic memory allocation. DesCaRTeS also uses strings as its messages and parameter lists to invocations. The operations required to build and parse these string take time, and variables used to store the results use already limited memory. The DesCaRTeS implementation could also be optimized by cleaning up code and improving algorithms to save memory and speed up performance.

Current implementations of DDP and DRW could be optimized to improve performance. Resource capabilities, as in SR, are simulated in Dynamic C by storing all operations declared in a particular resource in a structure. To be consistent, all capabilities are sent back to the creator of a resource or invoker of an operation that returns a resource capability. Most of the time, only a few of the capabilities returned are actually used. The unused capabilities use CPU time when they are added to the parameter list for the reply and parsed by the invoker. This scenario occurs in both

DDP and DRW load balancing implementations where extra capabilities are passed but not used during the periodic load information gathering. Thus, removing these extra capabilities may favor load balancing.

The SR implementations and Dynamic C implementations also used different hardware. Dynamic C's Rabbit processors (see Section 5.3) are much slower than the processors used by SR (see Section 5.2). Furthermore, SR implementations communicated on an ethernet that transmitted messages at a rate ten times faster than the ethernet used for Dynamic C implementations. Thus, Dynamic C tests were able to perform a far less amount of work than the SR tests were.

Chapter 7

Conclusion

This chapter presents the conclusions of this research. It also discusses ideas for future work.

7.1 Summary

This research presented a comparison between various load balancing and non-load balancing applications. These applications were implemented in both CM and CP styles. Furthermore, two different programming languages were used to implement the CM versions. Tests conducted on these implementations allowed us to compare load balancing performance using different programming styles, languages, and platforms.

The applications developed involved variations of two classic problems: Dining Philosophers and Readers/Writers. To make Dining Philosophers more interesting, philosophers were allowed to enter and leave the system. Likewise, readers and writers were allowed to enter and leave in the Readers/Writers problem. With these features, two load balancing algorithms were applied to each classic problem.

We implemented these algorithms in two programming languages: SR and Dynamic C. SR was used to develop the initial algorithm. Thus, the SR implementations were straightforward. To develop Dynamic C versions, a library called DesCaRTeS was designed and implemented to provide an SR-like run-time system and provide some features similar to those in the SR programming language.

Performance tests were conducted to examine the benefits and/or drawbacks of the load balancing algorithms used. In cases of very high or very low workloads, load balancing slightly hindered performance. In cases of moderate workload, both SR and Dynamic C implementations of load balancing generally performed well. Dining Philosophers implementations benefitted more from load balancing than Readers/Writers implementations. The exception to these conclusions is when processes have fewer opportunities to synchronize with the load balancer due to increased amounts of work per iteration. Such situations cause load balancing to yield unfavorable results.

For larger workloads, CM outperformed CP. For small to moderate workloads, however, performance comparisons favored CP. This research also revealed that developing programs under CM can have both advantages and disadvantages. Programming under CM requires less code to provide properties such as mutual exclusion, but a higher level of responsibility is placed on the programmer to ensure that these properties are not removed by misplaced yields.

7.2 Future Work

This research only examined a few load balancing applications. These applications could be further studied by varying a number of parameters. Different numbers of processes, values for workloads, and numbers of regions could all produce interesting results. Other classic problems, such as Producer/Consumer and Jacobi Iteration, could also be studied to reveal possible benefits and drawbacks of load balancing or CM style programming.

DesCaRTeS was designed to simulate some of the features of the SR language. As discussed in Section 4.2.4, this library has many limitations and could be improved. Further development could include implementing operation signatures, resource capabilities, and cleaner destruction abilities. In addition to improving DesCaRTeS, it would be interesting to port the entire SR language to run on the Rabbit processors.

Appendix A

Graphs

This appendix contains the graphs referenced in Chapter 5.

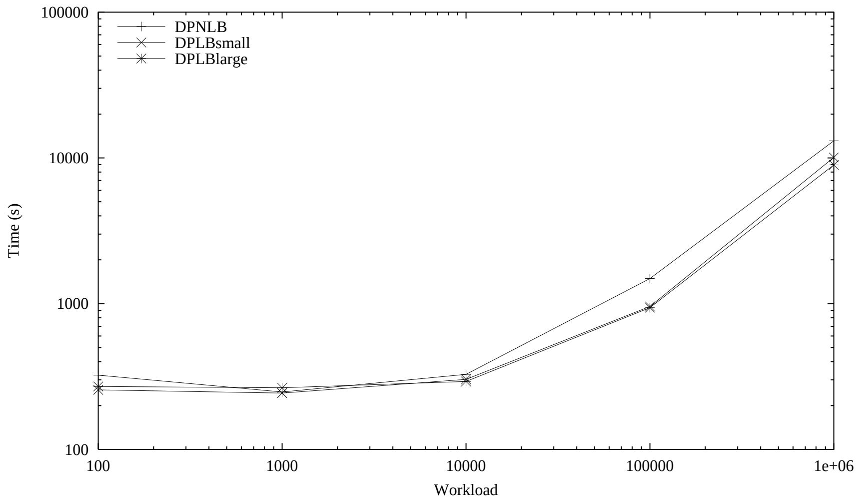

Figure A.1: SR DDP results under PCM (Experiment 1, axes are in log scale)

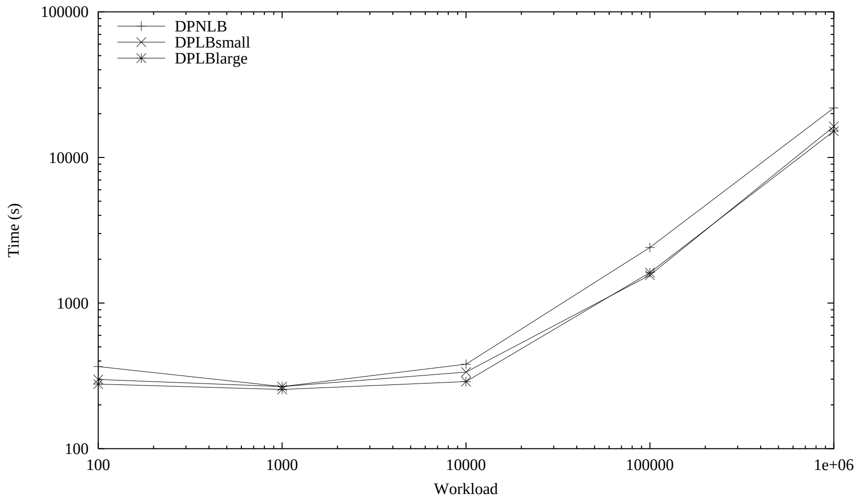

Figure A.2: SR DDP results under CP (Experiment 1, axes are in log scale)

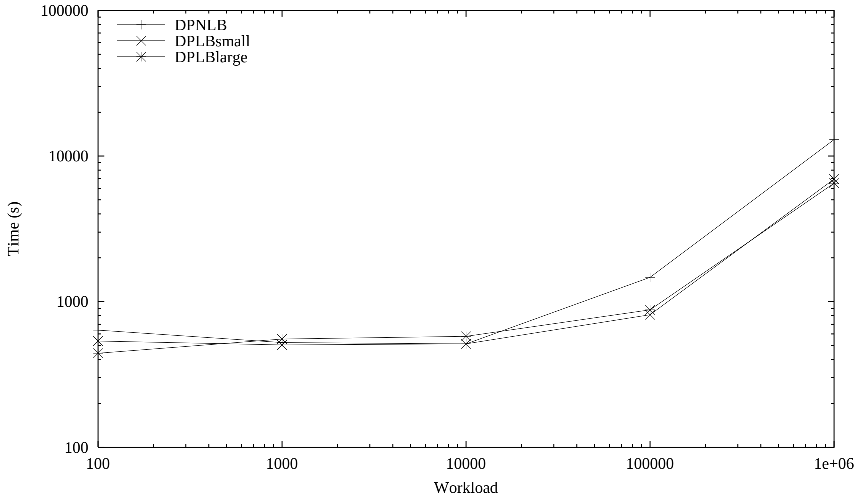

Figure A.3: SR DDP results under PCM (Experiment 2, axes are in log scale)

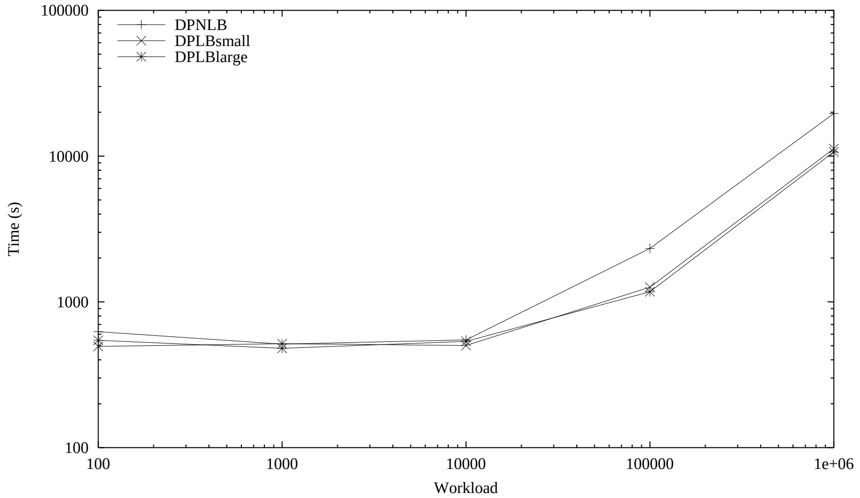

Figure A.4: SR DDP results under CP (Experiment 2, axes are in log scale)

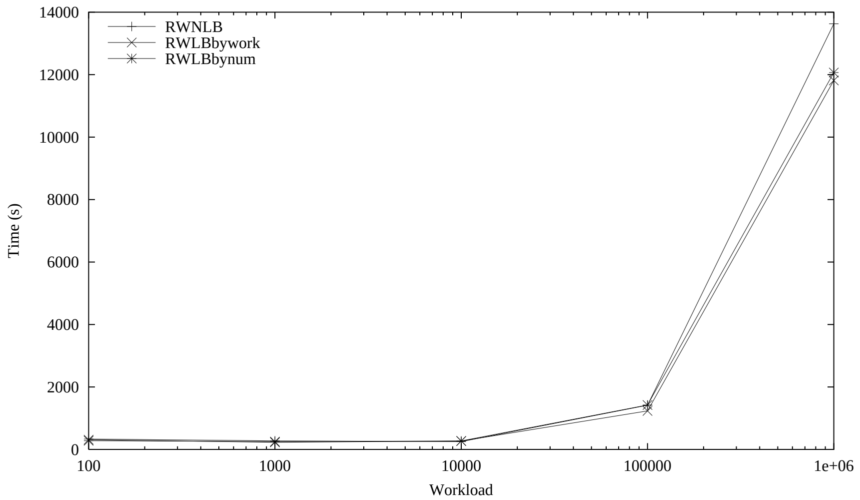

Figure A.5: SR DRW results under PCM (Experiment 1, x-axis is in log scale)

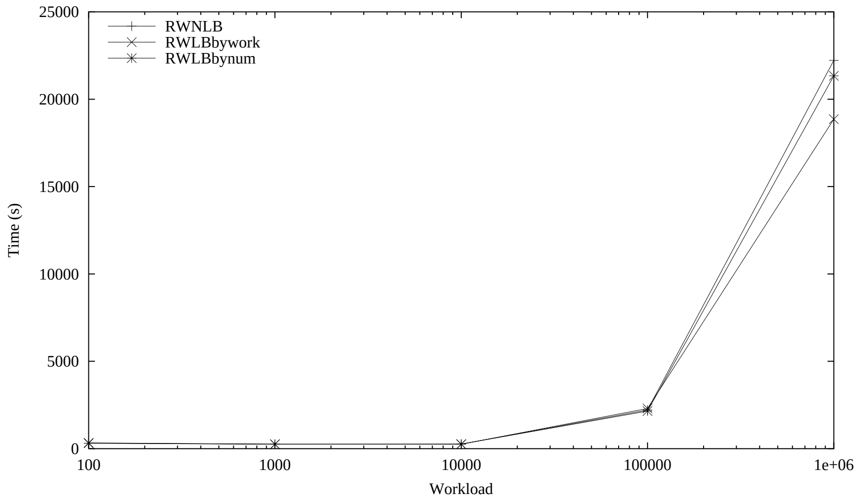

Figure A.6: SR DRW results under CP (Experiment 1, x-axis is in log scale)

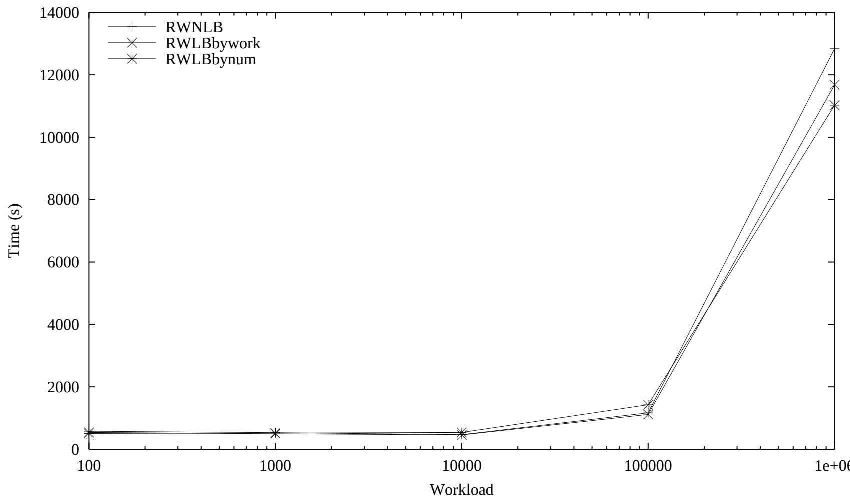

Figure A.7: SR DRW results under PCM (Experiment 2, x-axis is in log scale)

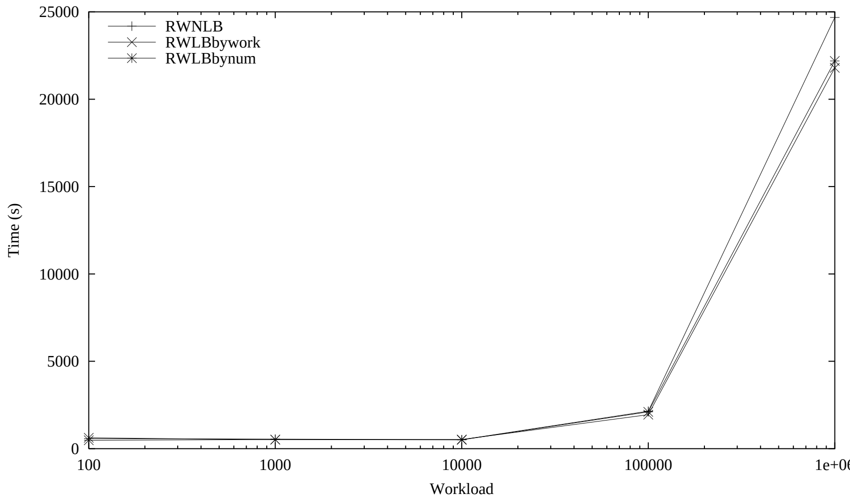

Figure A.8: SR DRW results under CP (Experiment 2, x-axis is in log scale)

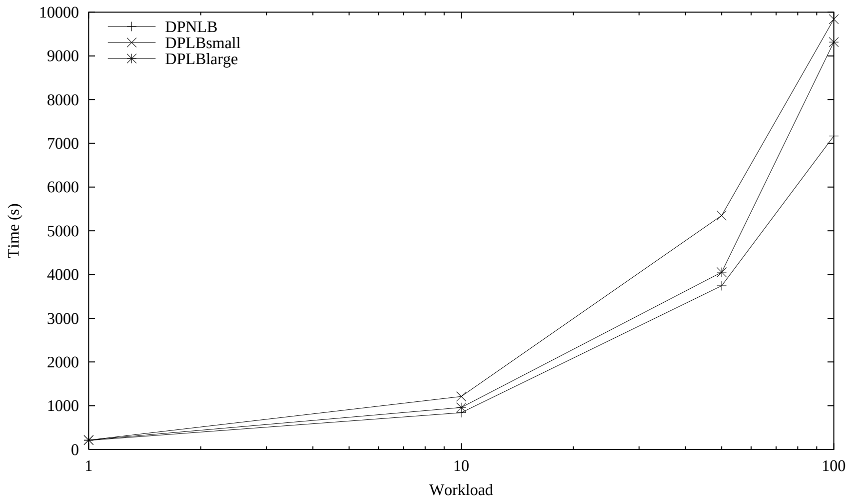

Figure A.9: Dynamic C DDP results (Experiment 1, x-axis is in log scale)

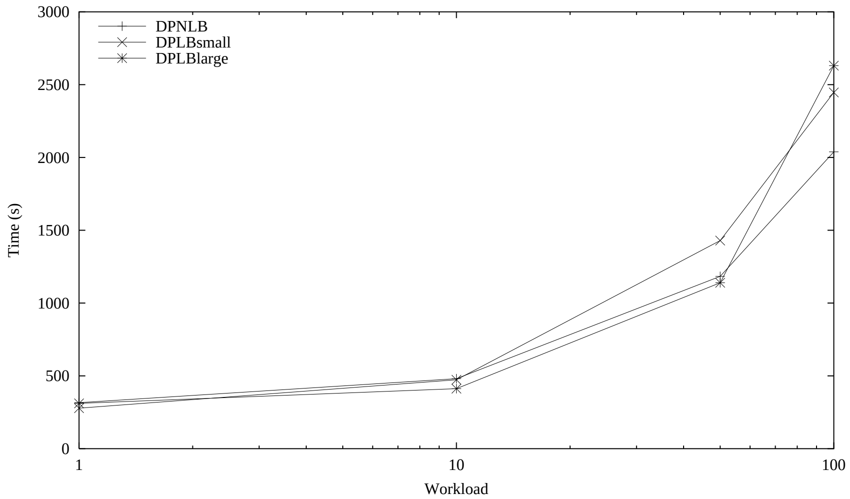

Figure A.10: Dynamic C DDP results (Experiment2, x-axis is in log scale)

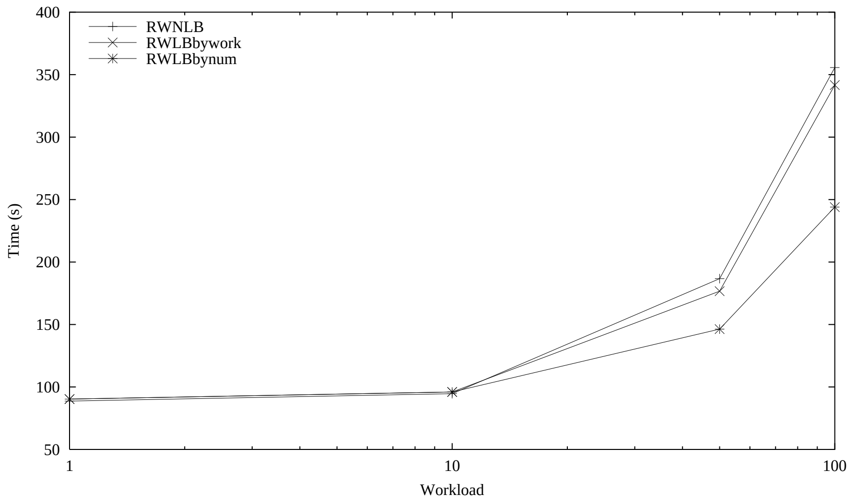

Figure A.11: Dynamic C DRW results (Experiment 1, x-axis is in log scale)

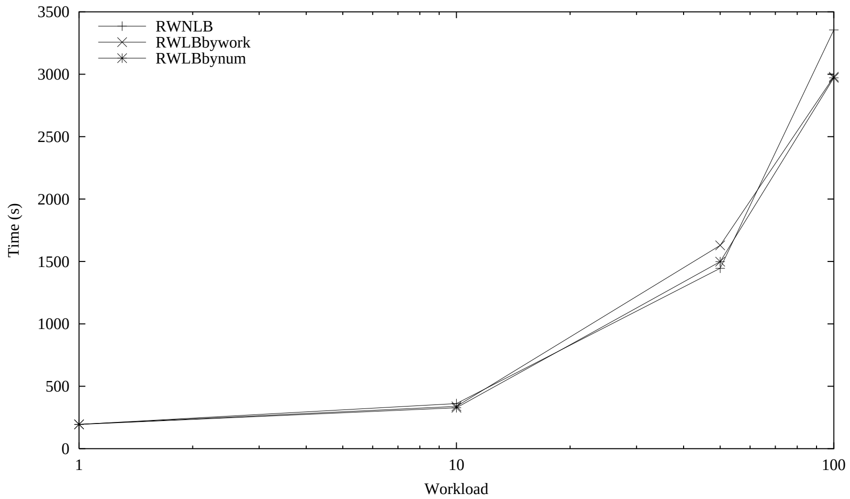

Figure A.12: Dynamic C DRW results (Experiment 2, x-axis is in log scale)

Bibliography

[1] Z World: Product Documentation Dynamic C for the Rabbit 2000, 2002. http://www.zworld.com/products/dc/docs_dcpremier.html.

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

[3] 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.

[4] G. Cornell and C. Horstmann. *Core Java.* Sun Microsystems, Inc., Mountain View, CA, 1996.

[5] G. Findlow and J. Billington. High-level nets for dynamic dining philosophers systems. In *Semantics for Concurrency. Proceedings of the International BCS-FACS Workshop*, pages 185–203, 1990.

[6] M. Hamdi and C. Lee. Dynamic load balancing of data parallel applications on a distributed network. In *9th International Conference on Supercomputing*, pages 170–179, July 1995.

[7] D. Hanson, K. Li, and J. P. Singh. Dynamic memory allocation, 1995. http://www.cs.princeton.edu/courses/archive/fall98/cs217/handouts/25alloc.pdf.

[8] C. A. R. Hoare. Communicating Sequential Processes. *Communications ACM*, 21(8):666–677, August 1978.

[9] Intermetrics, Inc., 733 Concord Ave, Cambridge, Massachusetts 02138. *The Ada 95 Annotated Reference Manual (v6.0)*, January 1995.

[10] S. Kang and H. Lee. Analysis and solution of non-preemptive policies for scheduling readers and writers. *Operating Systems Review*, 32(3):30–50, July 1998.

[11] Aaron W. Keen, Takashi Ishihara, Justin T. Maris, Tiejun Li, Eugene F. Fodor, and Ronald A. Olsson. A comparison of concurrent programming and cooperative multithreading. *Concurrency and Computation: Practice and Experience*. to appear.

[12] K. Li. Deterministic and randomized algorithms for distributed on-line task assignment and load balancing without load status information. In *ACM symposium on Applied Computing*, pages 613–622, February 1998.

[13] C. Sibertin-Blanc. Cooperative objects: Principles, use and implementation. *Concurrent OOP and PN, LNCS 2001*, pages 216–246, 2001.

[14] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, Inc., Upper Saddle River, New Jersey, 1999.