DesCaRTeS: A Run-Time System with SR-like Functionality for Programming a Network of Embedded Systems*

Justin T. Maris, Matthew D. Roper, and Ronald A. Olsson
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562 U.S.A.
{maris,roper,olsson}@cs.ucdavis.edu

October 2, 2003

Notes to Publisher:

- *Please send correspondence regarding this paper to* Olsson

Abstract

This paper describes a run-time system (DesCaRTeS) that has been used for developing applications that run on a network of embedded system controllers. DesCaRTeS is written in Dynamic C and provides Dynamic C programs with easy-to-use functionality to run on such networks, send and receive messages, perform remote procedure calls, and dynamically create processes. This paper reports on some of the applications for which DesCaRTeS has been used, the experience in developing them using DesCaRTeS, and the experience gained in developing DesCaRTeS. It also presents performance comparisons between these applications written using DesCaRTeS and equivalent applications written using native Dynamic C.

Keywords: concurrent programming models, concurrent programming languages, cooperative multithreading, embedded systems, run-time systems.

*This research was supported in part by Z-World, Inc. and the University of California under the MICRO program.

1 Introduction

The cooperative multithreading execution model (CM) is a specialized model of concurrent program execution. In this model, threads execute one at a time. A thread executes until it chooses to yield the processor or to wait for an event such as a shared variable meeting a particular condition, a device completing some operation, or a timeout occurring. This model of execution is especially well-suited for writing programs for real-world programmable controllers for embedded systems [1], such as those found in irrigation control systems and railroad crossing control systems. One language for writing these controllers is Z-World's Dynamic C [2], which is used for programming Z-World's Rabbit processor. The Rabbits, compared to common PCs, are very slow and limited in memory.

The SR concurrent programming language [3] provides a variety of synchronization mechanisms: remote procedure call (RPC), asynchronous message passing, rendezvous, and dynamic process creation. It is intended for programs that run under the more general concurrent programming model (CP), in which process execution is non-deterministic and multiple processes can execute at the same time on multiple processors, e.g., on a shared-memory multiprocessor or in a network of workstations.

SR's high-level abstraction of interprocess communication motivated the design of a run-time system, similar to that of SR's, that runs on a network of Rabbits. The SR run-time system was used as a model to develop DesCaRTeS: a *D*ynamic *C* *R*un-*T*ime *S*ystem. This run-time system provides Dynamic C programs with functionality similar to that used in SR programs, namely, the abilities to run programs on a network of systems, send and receive messages, perform remote procedure calls, and dynamically create processes. Although this functionality can be achieved with native Dynamic C programs with the TCP/IP library, using DesCaRTeS can reduce the programming effort. Moreover, DesCaRTeS is a CM-style library, utilizing the benefits that accompany CM-style programming. While some CP-style programming languages (e.g., Ada and Java) provide similar functionality on embedded system controllers, this research aimed to use a CM-style programming language to provide interprocess communication between embedded systems.

To gain feedback on the design and implementation of DesCaRTeS, it has been used to develop several applications. This paper reports on two of those applications. The first application is a simple "ping pong" program. It is a micro-benchmark designed to illustrate the costs of the basic DesCaRTeS mechanisms. The second application is a solution to the classic Readers/Writers Problem. It is a macro-benchmark designed to illustrate the overhead of using DesCaRTeS within a more realistic program. We have also used DesCaRTeS in experiments with load balancing algorithms for variants of Readers/Writers and Dining Philosophers; further details appear in Reference [4].

Our work has three goals. First, it aims to provide a richer communication scheme for an embedded system. It allows programmers to perform operations such as sending and receiving messages using mechanisms at a high-level of abstraction. Second, this research provides experience using the CM-style programming language Dynamic C. This experience has provided feedback and insight into the benefits and limitations of using such a language. Although programming in a CM-style language such as Dynamic C uses simple shared variable synchronization, the responsibility of introducing context switches in correct locations requires an increased level of attention to detail. Furthermore, Dynamic C's inability to dynamically manage memory can hinder a programmer, especially when trying to provide features such as dynamic process creation. However, Dynamic C does provide the necessary tools needed to implement a library such as DesCaRTeS. Finally, our work assesses the performance overhead in using DesCaRTeS compared to native Dynamic C. Our results show that for the micro-benchmark program, DesCaRTeS programs generally require about 100–200% additional execution time, but for the macro-benchmark program, DesCaRTeS programs generally require about 0–20% additional execution time. This paper analyzes and explains these results. Some of the high overhead is due to DesCaRTeS's use of slower memory, extra copying of messages, extra layers of procedure calls in using a library like DesCaRTeS, and type checking. These factors are not present in the native Dynamic C programs. Thus, the extra layer of abstraction provided by DesCaRTeS results in degraded performance. That is analogous in some ways to other abstraction/performance tradeoffs such as seen in programs written in C (or other higher-level languages) versus those written in assembly language.

The rest of this paper is organized as follows. Section 2 presents relevant background material: overviews of the SR language, Dynamic C language, and Rabbit processor. Section 3 introduces DesCaRTeS and outlines its use and features. Section 4 describes the applications developed using DesCaRTeS and gives performance comparisons. Section 5 discusses the experience gained in developing DesCaRTeS and the applications using DesCaRTeS. Section 6 presents the conclusions. Further details and discussion appear in Reference [5].

2 Background

This section discusses relevant background material. Section 2.1 provides an overview of the SR programming language. Section 2.2 presents an overview of the Dynamic C programming language. Section 2.3 discusses the Rabbit processor.

2.1 SR Overview

The SR programming language [3] provides high-level abstractions for concurrent program execution. The key concepts are are virtual machines, resources, and operations.

A virtual machine (VM) represents an address space. VMs are used for distributing a program onto physical machines. Each VM resides on one physical machine. VMs are created dynamically.

SR provides resources as its key modular 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 or create additional resource instances.

Processes in SR communicate using operations. A process may invoke these operations, sometimes with parameters, by using a **send** or **call** statement. Invocations via **send** are asynchronous; the invoker continues immediately after the invocation is made. Invocations via **call** are synchronous; the invoker blocks until the invocation is serviced. Invocations are serviced by input statements (defined by the keyword **in**) or procedures. An input statement receives an invocation from any of the operations it specifies. The invocation selected is determined on a FCFS basis. This selection method, however, can be modified two clauses: **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 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; the waiting invoker can continue with its execution once it gets 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 primitives, which are simplified input and **send** statements, respectively, are used on semaphores.

As noted above, VMs, resources, and operations are created dynamically and hence SR programs need a way to reference instances of such. SR uses *capabilities* for this purpose. A capability acts as a pointer and can be assigned to variables and passed as parameters, thus permitting, for example, dynamic communication paths.

To illustrate many of SR's features discussed in this section, Figures 1 and 2 give SR solutions for two classic concurrency problems [3]: the Dining Philosophers problem and the

Reader/Writers problem. The 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. As a result, no two neighboring philosophers may eat at the same time. The 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 writer currently has 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 Dining Philosophers solution presented in Figure 1 (from Reference [3]) uses a centralized controller (i.e., the **server** process) that manages the distribution of forks. Similarly, the Readers/Writers solution presented in Figure 2 also uses a centralized controller (i.e., the **RW_allocator** process) that handles requests to the database. These requests are serviced with a FCFS policy, thus not starving any process. When choosing between waiting readers or writers, the controller gives preference to whichever has been waiting longest. Furthermore, readers that have arrived before any waiting writer are allowed to proceed [6].

Although SR is intended as a language in which to write CP-style programs, it can also be used to write CM-style programs. The SR implementation provides a compile-time option that inhibits implicit context switches when the generated code is executed. Also, 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.)

Figure 3 illustrates a philosopher from Dining Philosophers written in both CP-style and in CM-style. This version of the Dining Philosophers problem presents a decentralized

```
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 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: 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
   # (basically test and set
   # on shared integer array fork;
   # left and right are indices
   # of neighboring philosophers)
   do fork[left] = 0 ->
     nap(0) # i.e., yield
   od
   fork[left] := 0
   do fork[right] = 0 ->
     nap(0) # i.e., yield
   od
   fork[right] := 0
   # eat
     ...
   # release forks
   fork[left] := 1; fork[right] := 1
od
```

(b) CM-style

Figure 3: Code for a philosopher in Dining Philosophers

solution. That is, compared with the version presented in Figure 1, this version has no centralized servant that manages the forks; instead, the philosophers themselves manage the forks. In the CP code, philosophers acquire forks through semaphores. In the CM code, philosophers acquire forks using shared variables. Most philosophers grab their left forks first and their right forks second. To prevent deadlock due to a circular wait condition in either the CM or CP code, the code introduces an asymmetric philosopher (not shown). This philosopher grabs its right fork first and its left fork second.

2.2 Dynamic C Overview

Dynamic C [2] extends the C language with various features to support programming in the CM model. 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. A thread can also **abort**, which also yields the processor; execution will begin at the beginning of the **costate**'s block the next time the costatement is executed.

Each **costate** statement may specify a **CoData** structure, which can be used to control the thread's execution. 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. Four functions can 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 of the function

```
#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 4: Dining Philosophers in Dynamic C

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 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 specify multiple **cofunc** functions. This statement yields if any of the **cofunc** functions do not complete execution as a result of explicit yields. When the thread runs again, the **cofunc** functions resumes execution of all uncompleted **cofunc** functions.

Figure 4 gives the Dynamic C equivalent of the solution to the Dining Philosophers problem given in Figure 3(b). This solution uses many of the Dynamic C features mentioned in this section.

2.3 Rabbit Processor

The specific embedded system controllers used for this research are Z World's Rabbit 2000 TCP/IP development boards. The Rabbit 2000 is an 8-bit processor with an 18 MHz clock speed. Each Rabbit provides 256K of Flash EPROM for program and data, 256K for file storage, and 128K of SRAM.

The Rabbit's memory structure has a major impact on the benchmark results in Section 4. Although the Rabbit's memory is limited in size, it is sufficient for most embedded applications. However, it may not be sufficient for applications that have larger data space requirements. DesCaRTeS itself uses various internal tables (see Section 3); applications using DesCaRTeS might also require more space. The Rabbit allows access to additional data space, which is called *xmem* (for extended memory) to distinguish it from *rootmem* (for root memory). However, access to *xmem* is indirect and is more costly. For example, consider how to set to zero the `x` field of the `S` structure pointed at by `p`. If `p` points to root memory, then the Dynamic C code is the usual

```
p->x = 0;
```

However, if `p` points to *xmem*, then the code is

```
xmem2root(s, p, sizeof(S)); // copy from xmem to rootmem
s.x = 0;
root2xmem(p, s, sizeof(S)); // copy from rootmem to xmem
```

(where `s` is an `S` structure in root memory). This copying from *xmem* to *rootmem* and back again is much more expensive than updating directly within *rootmem*. Each "copy" above requires two procedure calls to map the 16-bit logical *rootmem* pointer into a 20-bit physical address and then at least 70 assembly instructions to save registers, setup and perform the copy, and then restore the registers.

3 Dynamic C Run-time System (DesCaRTeS)

We developed a run-time system (RTS) in Dynamic C to provide many of the features of the SR language (see Section 2.1). These features include operations, capabilities, semaphores, interprocess communication, and process creation. This RTS, called DesCaRTeS, mimics the run-time system of SR [3], but with some changes due to different functionality and different target environment. DesCaRTeS defines new datatypes, constants, and functions. These various components will be seen in the discussion and examples in the rest of this section. Their specific purposes should be reasonably clear by their names. Appendix A presents the details of these DesCaRTeS library components.

3.1 DesCaRTeS Overview

DesCaRTeS provides programs written in Dynamic C with SR-like features by using data structures and algorithms that provide semantics similar to those of SR. This section provides an overview of DesCaRTeS. It also 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 interprocess 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 also uses this library to establish connections with other processors. This is similar 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.

DesCaRTeS provides a simplified form of SR-like operations through which processes communicate. 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 takes 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, being a run-time system (i.e., not a translator), has no idea of the number of parameters or their types. Thus, the correctness of building and parsing the messages that are passed to operations is the responsibility of the programmer, with help from functions in the DesCaRTeS library. (See Sections 3.1.2 and 3.2 for details.)

3.1.1 DesCaRTeS Template

Programs intending to use DesCaRTeS must follow a specific format to ensure correct functionality of the library. Figure 5 provides a basic template for writing programs using DesCaRTeS. The code uses various DesCaRTeS constants, datatypes, and functions. 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 the main program 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.

3.1.2 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 (see Section 2.1) can be passed to other processes as well.

To illustrate, consider the SR code in Figure 6. Execution begins in resource **A**. The code creates an instance of resource **B** and then invokes operation **b** within that resource instance. It then receives an invocation on operation **a**.

Figure 7 shows code written using DesCaRTeS that is functionally equivalent to the SR code in Figure 6. Process **A** declares an operation **a** and sends the capability to a newly created process **B**. 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.

The DesCaRTeS library performs run-time type checking to ensure that the parameters in a message match the type of those in the operation named in the receive. For example, the code for process **B** in Figure 7 uses `RTSParseInt` to parse an integer. The `RTSParseInt` procedure checks that the type of field in the message is actually an integer. (See Section 3.2 for details.)

```
#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
     RTSEND_PROCESS;
}
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 5: Template for using 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 # resource creation completes early due to this reply.
          # i.e., creator continues and initial process in this
          # instance continues into the loop below.
    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 6: Sample SR code with resource creation and operation invocations

```
cofunc void A(CoData * p){
    ParamBuffer params;
    ParamBuffer rtn;
    // A's operation
    OP a;
    // capability to B's operation
    CAP b;
    int i;
    RTSSTART_PROCESS(p);
    RTSOpDeclare(a, INTEGER, CALL);
    // 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){
    ParamBuffer params;
    ParamBuffer rtn;
    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
    RTSSTART_PROCESS(p);
    RTSOpDeclare(b, NULL, SENDCALL);
    // grab and parse the creation parameters
    RTSGetParams(&params);
    RTSParseCap(&params, &a);
    RTSReply(NULL);
    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 7: Dynamic C code functionally equivalent to the SR code in Figure 6

3.1.3 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 8 illustrates how a remote procedure call, with the limitations described later in Section 5.1, can be simulated. Here, a procedure is represented by a process, explicitly created by its parent process, that 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.

3.2 DesCaRTeS Memory Placement and Message Formats

The previous discussion described the general design and some of the details of DesCaRTeS. Two other aspects of DesCaRTeS can have a major impact on performance and are important in understanding the benchmark results in Section 4. They are: whether a data object is stored in *xmem* or *rootmem* and whether messages are encoded in ASCII format or binary format. DesCaRTeS has implementations for each of these possible combinations. The specific implementation of DesCaRTeS are referred to as $RTS_{f,m}$, where f indicates the message format (a for ASCII or b for binary) and m indicates the kind of memory (x for *xmem* or r for *rootmem*). For example, $RTS_{a,r}$ is the version of DesCaRTeS that uses ASCII message format and *rootmem*.

Section 2.3 discussed *xmem* and *rootmem*. As noted in Section 3.1, DesCaRTeS requires various internal data structures for representing processes and operation invocations. These

```
// this cofunction is the resource instance which creates the procedure
cofunc void parent_cofunc(CoData * p){
    ParamBuffer rtn;
    ...
    // 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){
    ParamBuffer params;
    ParamBuffer rtn;
    CAP caparray[2];
    int loopstop;
    OP proc;
    OP procdie;

    RTSSTART_PROCESS(p);
    // declare operations and send them back to the parent
    // 2 NULLs below should be replaced to reflect actual parameters to procedures
    RTSOpDeclare(proc, NULL, SENDCALL);
    RTSOpDeclare(procdie, NULL, CALL);
    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 8: Template for remote procedure calls using DesCaRTeS

data structures can be stored in either *xmem* or *rootmem*. If stored in *rootmem*, the user must specify through `#define` directives the maximum number of each kind of structure required for the program. If stored in *xmem*, DesCaRTeS provides dynamic memory allocation and deallocation (within *xmem*; see Appendix A.3) for its internal data structures. (But, see Section 5.1 for further discussion.) When a message is received by DesCaRTeS, it is placed into *rootmem*. For the *xmem* versions of DesCaRTeS, the message is copied to the invocation list for the operation, which resides in *xmem*. Later, when a user process selects the message for servicing, the message is copied back from *xmem* to *rootmem*.

The Dynamic C TCP/IP library supports messages sent between systems in ASCII format or binary format. In general, binary format is more space efficient (shorter messages) and time efficient (less and faster code to marshal and unmarshal parameters), but ASCII format is helpful for program development and debugging (e.g., ASCII messages can be observed using a sniffer on the network of Rabbits).

The messages DesCaRTeS sends encode parameter values and their types. The latter is needed so that, on receipt of a message, DesCaRTeS can check that the parameters in the message match the type of those in the operation named in the receive (as described in Section 3.1.2). In ASCII format, the integer 273, for example, appears as the string `_INT(273)`. In binary format, the above parameter is represented as a byte specifying an integer type (the "type-byte") followed by the actual binary representation of 273. The code that handles ASCII messages is considerably less efficient than the code that handles binary messages for two reasons. First, the code that handles ASCII messages uses a series of `strcmp` tests to determine the type of the message, whereas the code that handles binary messages uses a `switch` statement on the "type-byte". Second, the code that handles ASCII messages converts each integer into a string, sends the string as part of the message, and then converts the string back into an integer. The cost of this conversion, therefore, depends on the value of the integer. The code that handles binary messages just sends the integer as part of the message with no conversion on either side. Hence, the cost of sending an integer parameter depends on the value of the integer, but such cost is fixed in the binary versions.

4 Performance Comparisons

We developed several applications that used DesCaRTeS and compared them with native Dynamic C programs.[1] These applications can be viewed as micro-benchmarks and macro-benchmarks. The micro-benchmark is intended to measure the basic overhead in using DesCaRTeS compared to native Dynamic C. The macro-benchmark is designed to measure the overhead of using DesCaRTeS within a more realistic program. We have also used DesCaRTeS in experiments with load balancing algorithms for variants of Readers/Writers and Dining Philosophers [4].

We ran our performance experiments on a network of Rabbits (see Section 2.3). The applications used two or three Rabbits on a 10 Mb ethernet network. The network uses a hub (not a switch), so collisions can occur on applications that use more than two Rabbits.

4.1 Micro-benchmarks

The first application (micro-benchmark) is a "ping pong" program, which runs on two machines. A process on one machine sends a message to the second machine and waits for a response; a process on the second machine waits for a message from the first machine and then sends a response. The essence of the program, in SR-like pseudo-code, is given in Figure 9. In this program, `f` and `g` are operations that take parameters. The actual parameter lists are represented by `P` and `R`; these parameters are received into variable lists represented by `Q` and `S`. Appendix B gives the actual code for the ping pong program written in Dynamic C and in DesCaRTeS.

Table 1 shows the execution time per pair of invocations for the ping pong programs,

[1]For brevity, we refer to the former as "DesCaRTeS programs" and the latter as "Dynamic C" programs. These terms are less precise (but we hope are not confusing) because DesCaRTeS is really a run-time system for use with Dynamic C programs, not a language, and that both kinds of programs are Dynamic C programs.

```
# process A (on machine 1)      # process B (on machine 2)
fa k := 1 to N ->               fa k := 1 to N ->
  send f(P)                       receive f(Q)
  receive g(S)                    send g(R)
af                              af
```

Figure 9: SR-like ping pong program

Parameters	DyC_a	DyC_b	$RTS_{a,x}$	$RTS_{a,r}$	$RTS_{b,x}$	$RTS_{b,r}$
0	9.38	6.67	28.68	17.91	22.54	12.11
1	11.28	6.79	32.82	22.03	23.09	12.66
2	12.85	6.93	36.81	25.88	23.46	13.02
5	17.97	7.35	48.47	37.78	24.61	14.34
10	26.86	8.06	68.65	58.03	26.71	16.33
15	36.01	8.76	89.55	78.53	28.79	18.35
20	45.45	9.47	110.70	100.00	30.71	20.39
25	55.45	10.17	133.33	121.95	32.72	22.34
30	65.50	10.87	156.25	144.23	34.80	24.33

Table 1: Round-trip invocation times (msecs) for the ping-pong programs

i.e., the time it takes for one iteration of the loop in Figure 9. Figure 10 show these data graphically. Data are presented for the four DesCaRTeS implementations described in Section 3.2 and two versions of the Dynamic C program, one using ASCII messages and the other using binary messages, respectively denoted DyC_a and DyC_b (both of which used only *rootmem*). The data do not include the program start-up costs. The data represent averages of multiple executions. The variances were insignificant. The table shows data for different numbers of integer parameters in the operation, i.e., that get sent in messages. Each execution used randomly selected values for the parameter values in the message. (The parameter value affects message length for the ASCII versions as described in Section 3.2.)

As can be seen from the data in Table 1 and the graphs in Figure 10 (and as one might expect), DyC_b performs the best and $RTS_{a,x}$ performs the worst. The major factors contributing to the differences in performance are:

- ASCII versus binary message format (see Section 3.2).
- message copying. As described in Section 3.2 the *xmem* versions of DesCaRTeS copy

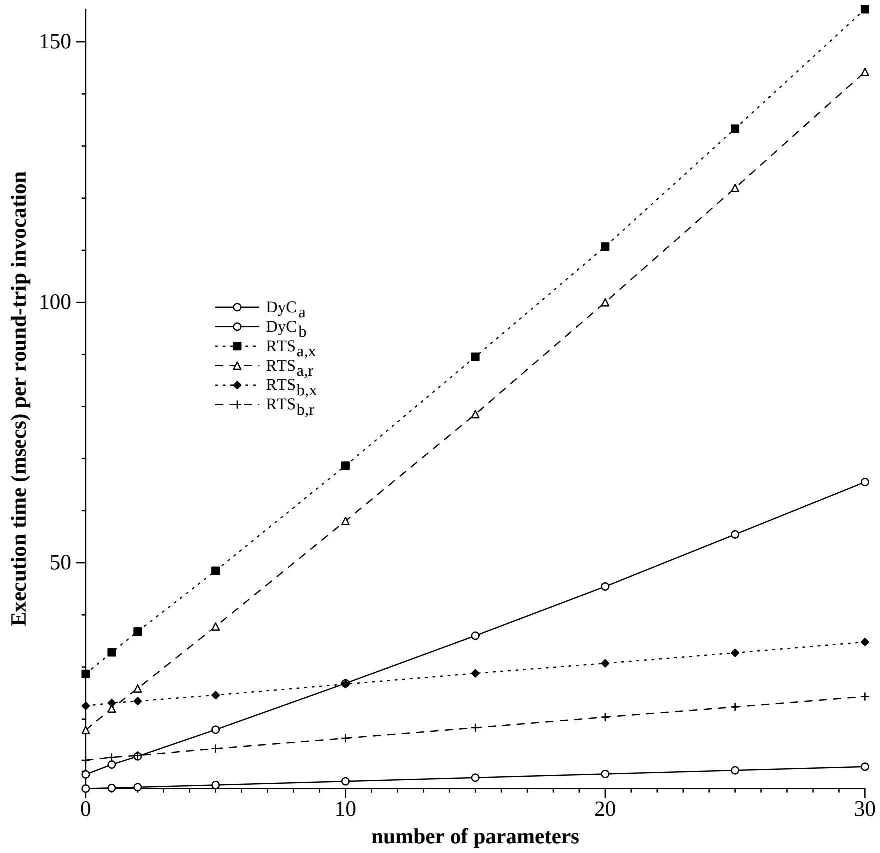

Figure 10: Round-trip invocation times (msecs) for the ping-pong programs

messages from *rootmem* to *xmem* and back again.

- extra layers of procedure calls due to the structure of the DesCaRTeS library. Each call to a DesCaRTeS library procedure may involve several other procedure calls. For example, a message receive in the DesCaRTeS program takes six procedure calls (plus additional procedure calls for allocating and deallocating memory in the *xmem* versions of DesCaRTeS), whereas it takes just one in the Dynamic C program.

- operation lookup in DesCaRTeS internal tables. Each send or receive of a message requires DesCaRTeS to lookup the specified operation in its operation table to locate the machine on which the operation is located. The Dynamic C program does no such lookup; it just sends the message directly on the socket.

Parameters	$RTS_{a,x}/DyC_a$	$RTS_{a,r}/DyC_a$	$RTS_{b,x}/DyC_b$	$RTS_{b,r}/DyC_b$
0	3.06	1.91	3.38	1.82
1	2.91	1.95	3.40	1.86
2	2.86	2.01	3.39	1.88
5	2.70	2.10	3.35	1.95
10	2.56	2.16	3.31	2.03
15	2.49	2.18	3.29	2.09
20	2.44	2.20	3.24	2.15
25	2.40	2.20	3.22	2.20
30	2.39	2.20	3.20	2.24

Table 2: Ratios of execution times for the ping-pong programs (from Table 1)

- run-time type checking. As noted in Sections 3.1.2 and 3.2, DesCaRTeS programs check that the types of parameters in a message match the types of those in the operation named in the receive. The Dynamic C program does no such type checking.

To make clearer the relative performance of the programs, Table 2 shows the ratios of execution times, specifically,

$Time(\text{DesCaRTeS})/Time(\text{Dynamic C})$

The ratios shown compare DesCaRTeS and Dynamic C programs that use the same message format (ASCII or binary). As shown by the ratios in Table 2, DesCaRTeS programs incur a significant overhead compared to their Dynamic C counterparts, for the reasons enumerated above. Notice that in Table 2 as the number of parameters increases the ratios in the third and fifth columns increase but the ratios in the second and fourth columns decrease. As seen in Figure 10, the overall execution times are roughly linear in the number of parameters, N.[2] In the limit as N increases, each ratio approaches the ratio of the slopes of the lines. As N increases, whether a specific ratio increases, decreases, or is constant is determined by the specific values of the lines' slopes and y-intercepts. Intuitively, these differing trends

[2]They are not exactly linear due to factors such as memory allocation and splitting up messages into multiple TCP packets. Also, when a message contains any parameters, some parameter handling setup code is executed, but it is skipped over when a message has no parameters.

in the ratios reflect where the programs spend most of their execution time. The overall execution time can be viewed as the sum of the times for message copying and for handling the parameters (i.e., marshaling, unmarshaling, and type checking). For example, when N is small, the DesCaRTeS programs that use *xmem* spend more time relatively in copying than in handling (and they are being compared with the *DyC* programs, which use *rootmem*). But, as N increases, the time they spend in handling increases, so the ratios in the second and fourth columns decrease.

4.2 Macro-benchmarks

We used DesCaRTeS to implement a version of the Readers/Writers problem. This version extends the classic problem, presented in Section 2, to allow readers and writers to periodically enter and leave the system. A new process enters the system as a reader or a writer and performs a certain number of iterations of local work and accessing the database. Once the process has completed all of its iterations, it then terminates. We refer to this version of Readers/Writers as Dynamic Readers/Writers (DRW), in the spirit of the Dynamic Dining Philosophers [7]. The DRW application splits its readers and writers into two regions, one on each Rabbit. It uses a centralized database manager controller process (i.e., the **RW_allocator** process from Figure 2), located on a third Rabbit. Figure 11 illustrates DRW for two regions.

We ran two sets of experiments with the DRW programs. The experiments used the parameters defined in Table 3. Figure 12 illustrates the use of LOCAL_WORK, DB_WORK, and WORK in SR-like pseudo-code. Table 4 shows the specific parameter values used for each DRW experiment. As noted in Table 3, some parameter values are chosen randomly. Also, whether a given worker is a reader or a writer is determined randomly. This randomness can lead to varying patterns of interaction among the reader processes and the writer processes. In the experiments, it led to high variances for specific tests and is a key factor in the overall results, as noted below. The data presented for the experiments represent

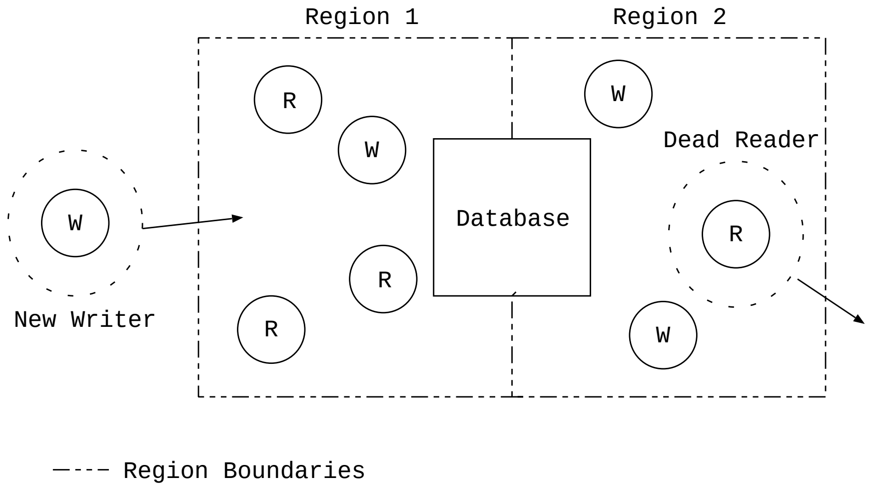

Figure 11: The Dynamic Readers/Writers Model

averages of ten executions of each specific test.

Tables 5 and 6 show the execution times of the programs for the two experiments.[3] Figures 13 and 14 show these data graphically. Tables 7 and 8 show the ratios of the executions times for the two experiments.

The performances across the various DesCaRTeS versions are close. The messages between processes in these programs contain no parameters, except in those messages used to start up workers (which amount to a relatively small portion of the total messages). Hence, the performance differences between the ASCII and binary versions are small here, unlike the performance differences seen for the ping pong programs (Section 4.1). The performance of the DesCaRTeS programs that use *xmem* and those that use *rootmem* are generally quite close, but there are a few notable differences. At one extreme (Experiment 2, $Work = 50$, $RTS_{a,r}$ *vs.* $RTS_{a,x}$), the *rootmem* program executes about 19% faster than the *xmem* pro-

[3] Unlike for the ping pong benchmark in Section 4.1, where we presented data for ASCII and binary versions of Dynamic C, in this section, we present data for only a binary version of Dynamic C. It uses a single byte in its messages to indicate message type, e.g., start reading, end reading, etc. An ASCII version would also use just a single byte, so there would be no real difference in these implementations. (However, their performances could differ slightly as the ASCII and binary versions of DesCaRTeS do, as explained later in this section.)

Parameter	Description
N	Limit on the number of workers (threads) allowed in a region at any point of time.
TEST_NUM	Number of threads created during a test. Test completes when all of these threads terminate.
ITER	Upper bound on the number of iterations a thread executes during its lifetime. Actual value is randomly selected between 1 and this value.
LOCAL_WORK	Number of iterations for the loop before accessing the DB.
DB_WORK	Number of iterations for the loop while accessing the DB. in the critical section of a thread's code executes.
WORK	Number of iterations of inner loop. This parameter is varied within each experiment — see data tables.
CREATE_TIME	The upper bound on the number of milliseconds between thread creations. Actual time is randomly selected between 1 and this value.

Table 3: Parameter definitions for DRW experiments

Parameter	Experiment 1	Experiment 2
N	5	5
TEST_NUM	20	20
ITER	10	100
LOCAL_WORK	100	10
DB_WORK	100	10
CREATE_TIME	10000	20000

Table 4: Parameters for DRW experiments

```
# iter is a randomly generated integer between 1 and ITER
do iter > 0 ->
    # before accessing DB
    fa i := 1 to LOCAL_WORK ->
        fa j := 1 to WORK ->
            nap(0) # i.e., yield
        af
    af
    # gain access to DB
    start_read()
    # simulate work during access to DB
    fa i := 1 to DB_WORK ->
        fa j := 1 to WORK ->
            nap(0) # i.e., yield
        af
    af
    # release access to DB
    end_read()
    iter--;
od
```

Figure 12: Code fragment in SR illustrating test parameters for a reader process in DRW

Work	DyC	$RTS_{a,x}$	$RTS_{a,r}$	$RTS_{b,x}$	$RTS_{b,r}$
1	98.90	102.10	106.60	108.80	98.00
10	107.30	110.90	114.90	106.90	114.70
50	205.50	252.20	264.80	261.20	246.40
100	481.40	491.10	473.60	497.70	501.90

Table 5: Execution times (secs) for the DRW programs (Experiment 1)

Work	DyC	$RTS_{a,x}$	$RTS_{a,r}$	$RTS_{b,x}$	$RTS_{b,r}$
1	191.00	194.10	186.40	195.70	180.20
10	193.40	197.80	196.50	203.20	211.70
50	257.80	339.10	285.40	313.00	310.00
100	448.60	572.30	537.10	523.80	553.50

Table 6: Execution times (secs) for the DRW programs (Experiment 2)

Work	$RTS_{a,x}/DyC$	$RTS_{a,r}/DyC$	$RTS_{b,x}/DyC$	$RTS_{b,r}/DyC$
1	1.03	1.08	1.10	0.99
10	1.03	1.07	1.00	1.07
50	1.23	1.29	1.27	1.20
100	1.02	0.98	1.03	1.04

Table 7: Ratios of execution times for the DRW programs (Experiment 1) (from Table 5)

Work	$RTS_{a,x}/DyC$	$RTS_{a,r}/DyC$	$RTS_{b,x}/DyC$	$RTS_{b,r}/DyC$
1	1.02	0.98	1.02	0.94
10	1.02	1.02	1.05	1.09
50	1.32	1.11	1.21	1.20
100	1.28	1.20	1.17	1.23

Table 8: Ratios of execution times for the DRW programs (Experiment 2) (from Table 6)

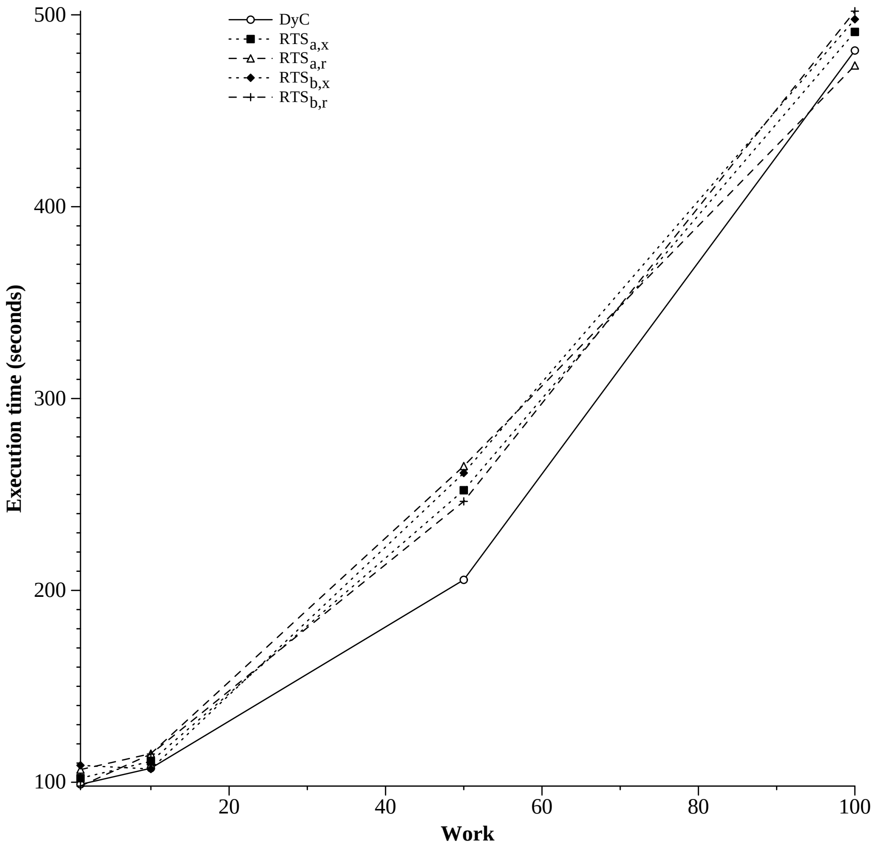

Figure 13: Execution times (secs) for the DRW programs (Experiment 1)

gram. At the other extreme (Experiment 2, $Work = 100$, $RTS_{b,x}$ *vs.* $RTS_{b,r}$), the *xmem* program executes about 5% faster than the *rootmem* program. These few differences are attributed to the randomness mentioned above.

The Dynamic C programs generally perform better than their DesCaRTeS counterparts, although the performance results are mixed. The ratios indicate a range of 94% to 132%. The DesCaRTeS programs run slightly faster in a few cases, about the same or slightly worse in most cases, and notably worse in several cases. These differences are again attributed to the randomness mentioned above. However, one factor contributing to the general closeness of results is that the particular choice of parameters (e.g., CREATE_TIME and ITER) was observed to lead to some periods of idleness, i.e., when no workers were alive. Thus, what would be an otherwise longer running program has a chance to "catch up" unlike in the

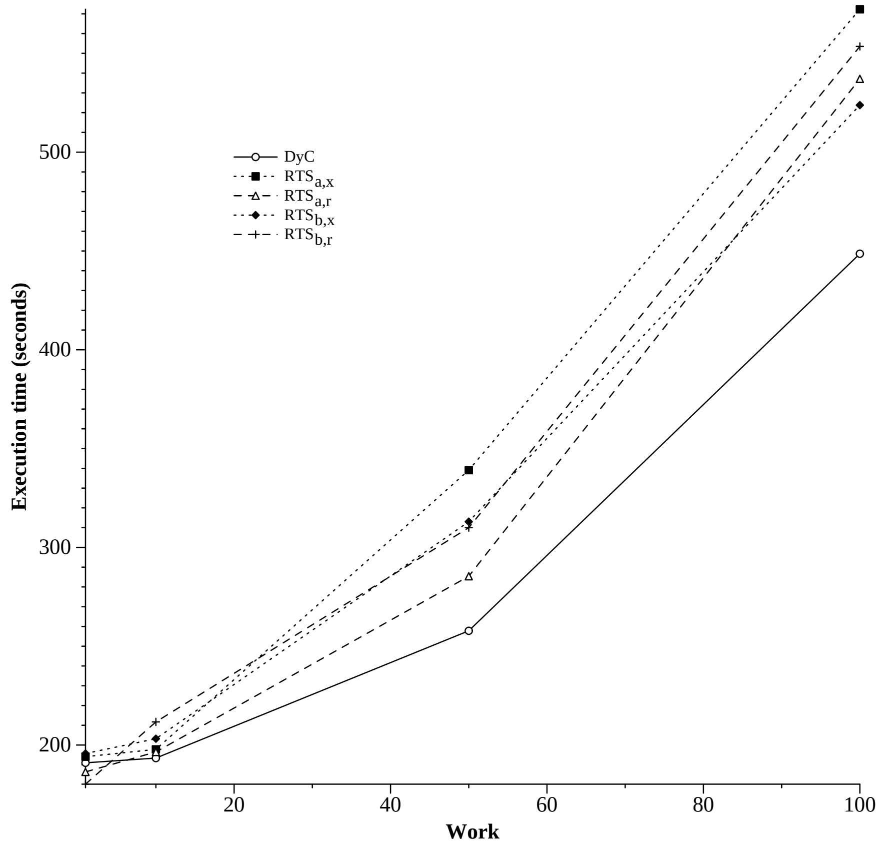

Figure 14: Execution times (secs) for the DRW programs (Experiment 2)

ping pong programs (Section 4.1).

5 Discussion

This section first discusses limitations of DesCaRTeS. It then presents general experience and insight gained in developing CM-style programs and in using Dynamic C to implement DesCaRTeS, and briefly discusses related work.

5.1 DesCaRTeS Limitations

DesCaRTeS has some limitations compared to its SR counterpart. Many of the difficulties in implementing DesCaRTeS arose because the library does not have the benefit of gathering information at compile-time. This prevented DesCaRTeS from supporting certain features.

Some of the limitations are the result of our limited resources for developing the implementation and our focusing on a prototype with which we could develop the applications in Section 4. The key limitations are:

- DesCaRTeS does not support parameter dependent such-that clauses for input statements (see Section 2.1, especially the example in Figure 1). Unlike SR's **st** clause, this expression cannot be dependent on the values received by an invocation of the operation being serviced (which is very useful).

- DesCaRTeS does not support truly dynamic process creation. As seen in Figure 5, each process is represented by a separate **costate** statement in Dynamic C. Because these statements are specified statically, the number of processes is bounded. (This bound should match the number of internal DesCaRTeS process data structures in the *rootmem* version mentioned in Section 3.2.) 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.

- DesCaRTeS allows only one virtual machine per physical machine. As in SR, a virtual machine in DesCaRTeS can be destroyed. After the VM on a Rabbit is destroyed, a new VM can be created on that Rabbit.

5.2 General Observations

Developing programs using Dynamic C, a CM-style language, has some advantages and some disadvantages when compared to CP-style languages. Although CM sometimes needs less overhead than CP to ensure mutual exclusion [8], 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 under 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 (Section 3) 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 can all be accessed without protection. All operations on these resources, however, do not yield until after completion of critical sections and are essentially atomic in nature. This advantage helped to simplify the development of the library.

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 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. Developing libraries such as DesCaRTeS requires a programmer to exercise this responsibility.

The distinction on the Rabbit processor between whether a data object is stored in *xmem* or *rootmem* raises performance and ease-of-use issues. Programs that use *xmem* incur considerable overhead to their counterparts that use only *rootmem*, as illustrated by the performances of the DesCaRTeS implementations in Section 4. Programs that are able to fit all their data objects into *rootmem* are desirable. However, if such a program's data space requirements grow to exceed *rootmem*, then the program needs substantial rewriting to take advantage of *xmem*, e.g., to access structures indirectly as in the example in Section 2.3.

Other CP-style languages have been developed and used for embedded systems. For example, Ada has been used for programming applications such as data communication switches [9]. The MaRTE OS is a real-time kernel for embedded systems with interfaces for Ada [10]. As another example, Java has been used for such purposes too [11]. However, it is difficult to compare our results with those for other systems or languages because of the significant differences in the hardware and software platforms used. For example, the Rabbit is an 8-bit processor with its particular memory organization and DesCaRTeS was built on top of a CM-style programming language, Dynamic C.

6 Conclusion

This work successfully implemented a run-time system, DesCaRTeS, for use with distributed programs on an embedded system. This run-time system was also created using a CM-style design. While this style of programming introduces some complexity in program design, DesCaRTeS demonstrates that CM-style programs can be successfully developed while maintaining the desired functionality. Moreover, the development of DesCaRTeS for use in an embedded system reflects that a high-level communication scheme can be implemented in a system with limited resources.

Section 4 described some applications developed using DesCaRTeS and gave performance comparisons between the DesCaRTeS applications and their native Dynamic C counterparts. As seen there, although the overhead for the micro-benchmark program is quite high (generally 100–200% additional execution time), the overhead for the macro-benchmark program is more reasonable (generally 0–20% additional execution time). Also, programs run using

DesCaRTeS provide type checking, while those run with native Dynamic C do not. Overall, these results for the macro-benchmarks are encouraging.

DesCaRTeS was designed to simulate some of the features of the SR language. As discussed in Section 5.1, this library has some limitations and could be improved. Further development to DesCaRTeS could include implementing more of the SR language mechanisms, such as parameter dependent such-that clauses. In addition to improving DesCaRTeS, it would be interesting and useful to develop an SR compiler that would generate Dynamic C code (whose form is outlined by the templates in Figures 5 and 7) that uses DesCaRTeS and runs on the network of Rabbit processors.

Acknowledgments

Gene Fodor, Takashi Ishihara, and Aaron Keen provided helpful comments on the design and implementation of DesCaRTeS. Matt Farrens provided helpful comments on drafts of the thesis [5] on which this paper is partially based. The anonymous reviewers gave constructive comments that led to improvements in this paper.

A DesCaRTeS Library Details

This appendix discusses DesCaRTeS in detail. Appendix A.1 discusses its new datatypes and their uses. Appendix A.2 lists relevant constants that should be used in a program using DesCaRTeS. Appendix A.3 describes both external and internal functions of DesCaRTeS.

A.1 Datatypes

DesCaRTeS offers five new datatypes that provide functionality similar to their SR equivalents [3].

- **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 four 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 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. The fourth field records the operation signature, represented by a string. It is used to ensure the correct building and parsing of parameter lists.

- **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.

- **RCAP**

  Variables of type **RCAP** are structures that contain three fields. The first field is a pointer to a process's CoData structure. The second field is the virtual machine index into the virtual machine table. This value indicates where the associated process was created. The third field records the physical address of the process's entry into the process table.

In addition, DesCaRTeS offers two new datatypes that provide abstractions of other SR entities.

- **PROCESS**

  Provides an abstraction of a process. It is a structure that contains various information regarding a process, e.g., that it is blocked waiting for an invocation.

- **ParamBuffer**

  Provides an abstraction of an operation's message. It is a structure that contains a character buffer to store a message and some integer fields that are used internally by DesCaRTeS to speed up message construction and parsing.

A.2 Constants

Type Constants DesCaRTeS provides the following constants that are used to build up signatures of operations, i.e., to specify parameter types in messages. Below lists the constants for the ASCII and binary versions of DesCaRTeS. In the ASCII version, these constants are strings (e.g., `"_INT"` and `"_CHAR"`) and signatures are a concatenation of such strings (e.g., `"_INT_INT"`) In the binary version, these constants are bytes and signatures are a list of such bytes.

- **INTEGER** (ASCII) or **TYPE_INT** (binary)

  Used to define an integer as part of the signature of an operation when declared using **RTSOpDeclare**.

- **REAL** (ASCII) or **TYPE_REAL** (binary)

  Used to define a real as part of the signature of an operation when declared using **RTSOpDeclare**.

- **CHARACTER** (ASCII) or **TYPE_CHAR** (binary)

  Used to define a character as part of the signature of an operation when declared using **RTSOpDeclare**.

- **STRING** (ASCII) or **TYPE_STRING** (binary)

  Used to define a string as part of the signature of an operation when declared using **RTSOpDeclare**.

- **LONG_INTEGER** (ASCII) or **TYPE_LONG** (binary)

  Used to define a long integer as part of the signature of an operation when declared using **RTSOpDeclare**.

- **CAPABILITY** (ASCII) or **TYPE_CAP** (binary)

  Used to define a capability as part of the signature of an operation when declared using **RTSOpDeclare**.

- **RESOURCE_CAPABILITY** (ASCII) or **TYPE_RCAP** (binary)

  Used to define a resource capability as part of the signature of an operation when declared using **RTSOpDeclare**.

Other 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 5).

- **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.

A.3 Functions

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

General DesCaRTeS itself relies on the continuous execution of a cofunction that handles communication with other machines. As illustrated in Figure 5, 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.

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.

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, ParamBuffer * params, ParamBuffer * rtn, VM v)**

  Creates a process registered as **name** with the parameters **params** on virtual machine **v**. Return parameters are copied to **rtn**. The first parameter in **rtn** is always the resource capability for the process created.

- **void RTSDestroy(RCAP rcap)**

  Destroys a process using resource capability **rcap**.

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

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

- **void RTSEND_PROCESS()**

  Terminates the current process. Calling this function allows the process to be created again later.

Operations and Process Communication The functions used to implement this communication are described below.

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

  Declares the operation designated by **o** by storing it in an operation table maintained by DesCaRTeS. **sig** is a string constructed using a sequence of constants described in section A.2. 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, ParamBuffer * params)**

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

- **void RTSCall(CAP c, ParamBuffer * params, ParamBuffer * 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, ParamBuffer * 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, ParamBuffer * 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, ParamBuffer * 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(ParamBuffer * 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.

Message Processing Setup

- **void RTSStartParams(ParamBuffer params)**

  Initializes the structure designated by **params**. It sets the string field by setting it to the empty string. (Actually, this is a macro.)

- **void RTSGetParams(ParamBuffer * params)**

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

Message Processing Marshaling and Unmarshaling The "add" functions provide for parameter marshaling and the "parse" functions provide for parameter unmarshaling.

In the ASCII version of DesCaRTeS, the "add" function append the type of the parameter as a string and a string representing the parameter's value, e.g., RTSAddInt will append `"_INT(43)"` for the integer 43. The "parse" functions ensure that the front of the begins with the appropriate string (e.g., `"_INT("`) and then parses up to the first ')'. The value between the parentheses is converted and stored in the second parameter. (To keep the initial implementation of the ASCII version simple, a string value is not allowed to contain ')'.)

In the binary version of DesCaRTeS, the "add" and "parse" functions behave similarly, but they use a single type-byte (Section A.2) and store values in binary.

- **void RTSAddInt(ParamBuffer * params, int i)**
- **void RTSAddReal(ParamBuffer * params, float f)**
- **void RTSAddChar(ParamBuffer * params, char c)**
- **void RTSAddString(ParamBuffer * params, char * s)**
- **void RTSAddLongInt(ParamBuffer * params, long int l)**
- **void RTSAddCap(ParamBuffer * params, CAP c)**
- **void RTSAddRcap(ParamBuffer * params, RCAP r)**
- **void RTSParseInt(ParamBuffer * params, int * i)**
- **void RTSParseReal(ParamBuffer * params, float * f)**
- **void RTSParseChar(ParamBuffer * params, char * c)**
- **void RTSParseString(ParamBuffer * params, char * s)**
- **void RTSParseLongInt(ParamBuffer * params, long int * l)**
- **void RTSParseCap(ParamBuffer * params, CAP * c)**
- **void RTSParseRcap(ParamBuffer * params, RCAP * r)**

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**.

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(PROC_ADDR paddr)**

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

- **PROC_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, PROC_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, ParamBuffer * rtn, PROC_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, PROC_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, PROC_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, PROC_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**.

Memory and Memory Management (Internal Functions) To easily support both *xmem* and *rootmem* versions, DesCaRTeS internally defines the macro PROC_ADDR. In the *xmem* version, it expands to XMEM_ADDR, which in turn expands to **unsigned long**. In the *rootmem* version, it expands to **PROCESS ***.

Dynamic C does not provide a method to dynamically allocate and deallocate memory. Thus, we implemented versions of malloc and free for use by 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. Since this function is used only internally by DesCaRTeS, there are only four possible values of **nbytes**: chunks for process descriptors, invocations (i.e., messages), wait list nodes for processes blocked on an operation, and replies. This routine first checks one of four memory chunk stacks to see if there are any previously allocated free chunks of memory of the appropriate size. If not, a new chunk of memory is allocated using the DynamicC memory allocator **xalloc** (**xalloc** is not used directly since it behaves more like the UNIX system call **sbreak** and thus leaves no method for deallocating memory). This routine runs in $O(1)$, even when it invokes **xalloc** (since that too is $O(1)$).

- **void RTSfree(XMEM_ADDR ptr)**

  Deallocates the memory pointed to by **ptr**. As noted under **RTSmalloc**, there are only four memory chunk sizes used by DesCaRTeS and free chunks of each size are stored on a separate stack. This function determines the amount of memory pointed to by **ptr** and adds the chunk to the appropriate stack. This routine runs in $O(1)$.

B Ping Pong Programs

This appendix presents the code for the ping pong programs discussed in Section 4.1. Each experiment (DesCaRTeS and Dynamic C) consists of two programs: ping on one machine and pong on the second machine. The DesCaRTeS programs use conditional compilation (macros) to select the version (ASCII or binary). The Dynamic C programs use separate programs for each version. Each program also uses the code given in Section B.4.

B.1 Dynamic C Ping Pong Program, ASCII Version

Dynamic C Ping Program, ASCII Version

```
/* asciiping.c
 *
 * A "ping" program for rabbits using regular DynamicC network calls and
 * dumping ints to the socket as ASCII string.  Sends a ping and waits for a
 * pong repeatedly until 30 seconds have elapsed.
 */

#define MY_IP_ADDRESS   "192.168.1.10"
#define PEER_IP_ADDRESS "192.168.1.11"
#define MY_NETMASK      "255.255.255.0"
#define PORT            22703

/* How many (randomly generated) parameters per op? */
#define NUM_PARAMS      30

#memmap xmem
#use "dcrtcp.lib"

main() {
    long endtime;               /* Program end time */
    char inmsg[600];            /* Incoming message buffer */
    char outmsg[600];           /* Outgoing message buffer */
    int msgs;                   /* Round-trip message count */
    tcp_Socket sock;            /* Connection socket */
    char param[20];             /* Parameter buffer */
    int pval;                   /* Incoming parameter value */
    int i;

    msgs = 0;

    ranSeed = (int)SEC_TIMER;

    /* Initialize the network system */
    sock_init();

    /* Open the network connection */
    if (!tcp_open(&sock, 0, resolve(PEER_IP_ADDRESS), PORT, NULL)) {
        printf("Unable to make network connection.\n");
        exit(1);
    }

    /* Wait for the connection to actually be established */
    while (!sock_established(&sock) && sock_bytesready(&sock) == -1)
        tcp_tick(NULL);
    sock_mode(&sock, TCP_MODE_ASCII);

    /* Loop for 30 seconds */
    endtime = MS_TIMER + 30000;
    printf("Starting...\n");
    while (MS_TIMER < endtime) {
        strcpy(outmsg, "msg");
#if NUM_PARAMS > 0
        /* Send parameters, one per line */
```

```
        for (i = 0; i < NUM_PARAMS; ++i) {
            sprintf(param, ",%d", irand());
            strcat(outmsg, param);
        }
#endif
        sock_puts(&sock, outmsg);
        sock_flush(&sock);

        /* Receive ack */
        do tcp_tick(&sock); while (sock_bytesready(&sock) <= 0);
        sock_gets(&sock, inmsg, sizeof(inmsg));

#if NUM_PARAMS > 0
        /* Read parameters from incoming message */
        strtok(inmsg, ",");
        for (i = 0; i < NUM_PARAMS; ++i)
            pval = atoi(strtok(NULL, ","));
#endif
        ++msgs;
    }

    /* Print out the number of round trip messages. */
    printf("Total messages: %d\n", msgs);

    /* Cleanup */
    sock_close(&sock);
    exit(0);
}
```

Dynamic C Pong Program, ASCII Version

```
/*
 * asciipong.c
 *
 * A "pong" program for rabbits using regular DynamicC network calls.  Receives
 * a 'ping' message and sends an acknowledgement.
 */
#define MY_IP_ADDRESS   "192.168.1.11"
#define MY_NETMASK      "255.255.255.0"
#define PORT            22703

#define NUM_PARAMS      30

#memmap xmem
#use "dcrtcp.lib"

main() {
    char outmsg[600];           /* Outgoing message buffer */
    char inmsg[600];            /* Incoming message buffer */
    tcp_Socket sock;            /* Connection socket */
    char param[20];             /* Outgoing parameter buffer */
    int pval;                   /* Incoming param val */
    int i;                      /* Loop var */

    /* Seed random number generator with the clock */
    ranSeed = (int)SEC_TIMER;

    /* Initialize the network system */
    sock_init();

    /* Open the network connection */
    tcp_listen(&sock, PORT, 0, 0, NULL, 0);

    /* Wait for the connection to actually be established */
    while (!sock_established(&sock))
        tcp_tick(NULL);
    sock_mode(&sock, TCP_MODE_ASCII);

    /* Loop until connection closes */
    do {
        if (sock_bytesready(&sock) >= 0) {
            /* Receive a message */
            sock_gets(&sock, inmsg, sizeof(inmsg));
#if NUM_PARAMS > 0
            /*
             * Read parameters from incoming message (we don't actually
             * ever use them though.
```

```
             */
            strtok(inmsg, ",");
            for (i = 0; i < NUM_PARAMS; ++i)
                pval = atoi(strtok(NULL, ","));
#endif

            /* Send the acknowledgement */
            strcpy(outmsg, "ack");
#if NUM_PARAMS > 0
            /* Send response parameters, one per line */
            for (i = 0; i < NUM_PARAMS; ++i) {
                sprintf(param, ",%d", irand());
                strcat(outmsg, param);
            }
#endif
            sock_puts(&sock, outmsg);
            sock_flush(&sock);
        }
    } while (tcp_tick(&sock));

    /* Cleanup */
    sock_close(&sock);
    forceSoftReset();
}
```

B.2 Dynamic C Ping Pong Program, Binary Version

Dynamic C Ping Program, Binary Version

```
/*
 * normalping.c
 *
 * A "ping" program for rabbits using regular DynamicC network calls.
 * Sends a ping and waits for a pong repeatedly until 30 seconds have
 * elapsed.
 */

#define MY_IP_ADDRESS   "192.168.1.10"
#define PEER_IP_ADDRESS "192.168.1.11"
#define MY_NETMASK      "255.255.255.0"
#define PORT            22703

/* How many (randomly generated) parameters per op? */
#define NUM_PARAMS      30

#memmap xmem
#use "dcrtcp.lib"

main() {
    long endtime;               /* Program end time */
    char inmsg[3 + NUM_PARAMS*sizeof(int) + 1];  /* Incoming message buffer */
    char outmsg[3 + NUM_PARAMS*sizeof(int) + 1]; /* Outgoing message buffer */
    int msgs;                   /* Round-trip message count */
    tcp_Socket sock;            /* Connection socket */
    char param[20];             /* Parameter buffer */
#if NUM_PARAMS > 0
    int pval[NUM_PARAMS];       /* Incoming parameter value */
#endif
    int i;

    msgs = 0;

    ranSeed = (int)SEC_TIMER;

    /* Initialize the network system */
    sock_init();

    /* Open the network connection */
    if (!tcp_open(&sock, 0, resolve(PEER_IP_ADDRESS), PORT, NULL)) {
        printf("Unable to make network connection.\n");
        exit(1);
    }

    /* Wait for the connection to actually be established */
    while (!sock_established(&sock) && sock_bytesready(&sock) == -1)
```

```
        tcp_tick(NULL);
    sock_mode(&sock, TCP_MODE_ASCII);

    /* Loop for 30 seconds */
    endtime = MS_TIMER + 30000;
    printf("Starting...\n");
    while (MS_TIMER < endtime) {
        strcpy(outmsg, "msg");

        /* Add parameters to message */
        for (i = 0; i < NUM_PARAMS; ++i)
            ((int*)(outmsg+3))[i] = irand();

        sock_write(&sock, outmsg, 3 + NUM_PARAMS*sizeof(int));
        sock_flush(&sock);

        /* Receive ack */
        if (sock_read(&sock, inmsg, 3 + NUM_PARAMS*sizeof(int)) >= 0)
            /*
             * Read parameters from incoming messages (even though we don't
             * actually use them in this simple program.
             */
#if NUM_PARAMS > 0
            for (i = 0; i < NUM_PARAMS; ++i)
                pval[i] = ((int*)(inmsg+3))[i];
#endif

        ++msgs;
    }

    /* Print out the number of round trip messages. */
    printf("Total messages: %d\n", msgs);

    /* Cleanup */
    sock_close(&sock);
    exit(0);
}
```

Dynamic C Pong Program, Binary Version

```
/*
 * normalpong.c
 *
 * A "pong" program for rabbits using regular DynamicC network calls.  Receives
 * a 'ping' message and sends an acknowledgement.
 */

#define MY_IP_ADDRESS   "192.168.1.11"
#define MY_NETMASK      "255.255.255.0"
#define PORT            22703

#define NUM_PARAMS      30

#memmap xmem
#use "dcrtcp.lib"

main() {
    /* Message format: "MSG"/"ACK" + raw integer values + '\0' */
    char outmsg[3 + NUM_PARAMS*sizeof(int) + 1]; /* Outgoing message buffer */
    char inmsg[3 + NUM_PARAMS*sizeof(int) + 1];  /* Incoming message buffer */
    tcp_Socket sock;            /* Connection socket */
#if NUM_PARAMS > 0
    int pval[NUM_PARAMS];       /* Incoming param val */
#endif
    int i;                      /* Loop var */

    /* Seed random number generator with the clock */
    ranSeed = (int)SEC_TIMER;

    /* Initialize the network system */
    sock_init();

    /* Open the network connection */
    tcp_listen(&sock, PORT, 0, 0, NULL, 0);

    /* Wait for the connection to actually be established */
    while (!sock_established(&sock))
```

```
        tcp_tick(NULL);
    sock_mode(&sock, TCP_MODE_ASCII);

    /* Loop until connection closes */
    do {
        if (sock_read(&sock, inmsg, 3 + NUM_PARAMS*sizeof(int)) >= 0) {
            /*
             * Read parameters from incoming messages (even though we don't
             * actually use them in this simple program.
             */
#if NUM_PARAMS > 0
            for (i = 0; i < NUM_PARAMS; ++i)
                pval[i] = ((int*)(inmsg+3))[i];
#endif

            /* Send the acknowledgement */
            strcpy(outmsg, "ack");

            /* Add parameters to message */
            for (i = 0; i < NUM_PARAMS; ++i)
                ((int*)(outmsg+3))[i] = irand();

            sock_write(&sock, outmsg, 3 + NUM_PARAMS*sizeof(int));
            sock_flush(&sock);
        }
    } while (tcp_tick(&sock));

    /* Cleanup */
    sock_close(&sock);
    forceSoftReset();
}
```

B.3 DesCaRTeS Ping Pong Program

DesCaRTeS Ping Program

```
/*
 * ping.c
 *
 * A "ping" program for rabbits using the DesCaRTeS runtime system.
 * Sends a ping and waits for a pong repeatedly until 30 seconds have
 * elapsed.
 */

#memmap xmem

#define MY_IP_ADDRESS   "192.168.1.10"
#define PEER_IP_ADDRESS "192.168.1.11"
#define MY_NETMASK      "255.255.255.0"

#define MAX_SOCKETS     4

/* How many (randomly generated) parameters per op? */
#define NUM_PARAMS      25

#define ROOTPROCS
#define RTS_MAX_PROCS  1
#define ROOTBLOCKS
#define RTS_MAX_BLOCKS 1
#define ROOTINV
#define RTS_MAX_INV    1

#define RTS_BINARY

#define RTS_MAX_OPS 20
#ifdef RTS_BINARY
#define RTS_MAX_SIG_LENGTH (NUM_PARAMS+1)
    #if (300 > (NUM_PARAMS*3 + 20))
    #define RTS_MAX_STRING_LENGTH 300
    #else
    #define RTS_MAX_STRING_LENGTH (NUM_PARAMS*3+40)
    #endif
#else
#define RTS_MAX_SIG_LENGTH (4*NUM_PARAMS+1)
    #if (300 > (NUM_PARAMS*12 + 20))
```

```
    #define RTS_MAX_STRING_LENGTH 300
    #else
    #define RTS_MAX_STRING_LENGTH (NUM_PARAMS*12+40)
    #endif
#endif

#use "rts.lib"

/* The op's signature.  There should be NUM_PARAMS INTEGERs
 * listed here or NULL if NUM_PARAMS is 0.
 */
#ifdef RTS_BINARY
//#define SIG NULL
const char SIG[NUM_PARAMS + 1] = {
    TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT,
    TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT,
    TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT,
    TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT,
    TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT, 0
};
#else
//#define SIG NULL
#define SIG INTEGER INTEGER INTEGER INTEGER INTEGER \
            INTEGER INTEGER INTEGER INTEGER INTEGER \
            INTEGER INTEGER INTEGER INTEGER INTEGER \
            INTEGER INTEGER INTEGER INTEGER INTEGER \
            INTEGER INTEGER INTEGER INTEGER INTEGER
#endif

/* Process declarations */
CoData main_proc;

VM peervm;                          /* VM on other rabbit */


cofunc void maincofunc(CoData* p) {
    /* Needed by RTS */
    ParamBuffer params, rtn;

    /* Local operation */
    OP pongop;

    /* Remote capability */
    CAP pingop;

    /* Normal vars */
    int i;
    int inval;
    long endtime;
    long count;

    RTSSTART_PROCESS(p);

    /* Declare ops */
    RTSOpDeclare(pongop, SIG, SEND);

    /* Create the ping handler process on the remote machine */
    peervm = RTSCreateVM(PEER_IP_ADDRESS);
    RTSStartParams(params);
    RTSAddCap(params, pongop);
    RTSCreate("ping_handler", &params, &rtn, peervm);
    RTSParseRcap(&rtn, NULL);
    RTSParseCap(&rtn, &pingop);

    RTSStartParams(params);
    endtime = MS_TIMER + 30000;
    count = 0;
    printf("Starting...\n");
    while (MS_TIMER < endtime) {
#if NUM_PARAMS > 0
        RTSStartParams(params);
        for (i = 0; i < NUM_PARAMS; ++i)
            RTSAddInt(params, irand());
#endif
        RTSSend(pingop, &params);
        RTSReceive(pongop, &params);
#if NUM_PARAMS > 0
        for (i = 0; i < NUM_PARAMS; ++i)
```

```
            RTSParseInt(&params, &inval);
#endif
        count++;
    }

    printf("Total messages: %d\n", count);

    RTSEND_PROCESS();
    RTSDestroyVM(peervm);

    exit(0);
}

main() {
    /* Seed random number generator */
    ranSeed = (int)SEC_TIMER;

    /* Initialize the RTS */
    RTSInit();

    /* Register processes */
    RTSRegCoData(&main_proc, "mainproc");

    /* Main loop */
    for (;;) {
        costate RTSprocess init_on {
            wfd RTS();
        }

        costate main_proc init_on {
            wfd maincofunc(&main_proc);
        }
    }
}
```

DesCaRTeS Pong Program

```
/*
 * pong.c
 *
 * A "pong" program for rabbits using the DesCaRTeS runtime system.
 * Receives ping messages (op invocations) and sends pong responses.
 * Used with ping.c to compare the performance of DesCaRTeS with
 * normal DynamicC code.
 */

#memmap xmem

#define MY_IP_ADDRESS "192.168.1.11"
#define MY_NETMASK    "255.255.255.0"

#define MAX_SOCKETS     4

/* How many (randomly generated) parameters per op? */
#define NUM_PARAMS      25

#define ROOTPROCS
#define RTS_MAX_PROCS  1
#define ROOTBLOCKS
#define RTS_MAX_BLOCKS 1
#define ROOTINV
#define RTS_MAX_INV    1

#define RTS_BINARY

#define RTS_MAX_OPS 20
#ifdef RTS_BINARY
#define RTS_MAX_SIG_LENGTH (NUM_PARAMS+1)
    #if (300 > (NUM_PARAMS*3 + 40))
    #define RTS_MAX_STRING_LENGTH 300
    #else
    #define RTS_MAX_STRING_LENGTH (NUM_PARAMS*3+40)
    #endif
#else
#define RTS_MAX_SIG_LENGTH (4*NUM_PARAMS+1)
    #if (300 > (NUM_PARAMS*12 + 40))
    #define RTS_MAX_STRING_LENGTH 300
```

```
    #else
    #define RTS_MAX_STRING_LENGTH (NUM_PARAMS*12+40)
    #endif
#endif
#use "rts.lib"

/* The op signatures.  There should be NUM_PARAMS INTEGERs
 * listed here or NULL if NUM_PARAMS is 0.
 */
#ifdef RTS_BINARY
const char SIG[NUM_PARAMS + 1] = {
    TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT,
    TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT,
    TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT,
    TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT,
    TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT, TYPE_INT, 0
};
#else
#define SIG INTEGER INTEGER INTEGER INTEGER INTEGER \
            INTEGER INTEGER INTEGER INTEGER INTEGER \
            INTEGER INTEGER INTEGER INTEGER INTEGER \
            INTEGER INTEGER INTEGER INTEGER INTEGER \
            INTEGER INTEGER INTEGER INTEGER INTEGER
#endif

/* Process declarations */
CoData ping_handler;

int i;
int inval;

cofunc void ping_handler_cf(CoData* p) {
    /* Needed by RTS */
    ParamBuffer params, rtn;

    /* Local operation */
    OP pingop;

    /* Remote capability */
    CAP pongop;

    RTSSTART_PROCESS(p);

    /* Get process creation parameters */
    RTSGetParams(params);
    RTSParseCap(&params, &pongop);

    /* Declare ops */
    RTSOpDeclare(pingop, SIG, SEND);

    /* Send the ping handler op back to the main process */
    RTSReply(NULL);

    while (1) {
        /* Receive a ping and send a pong */
        RTSReceive(pingop, &params);
#if NUM_PARAMS > 0
        /* Parse out incoming parameters */
        for (i = 0; i < NUM_PARAMS; ++i)
            RTSParseInt(&params, &inval);

        /* Generate outgoing parameters */
        RTSStartParams(params);
        for (i = 0; i < NUM_PARAMS; ++i)
            RTSAddInt(params, irand());
#endif
        RTSSend(pongop, &params);
    }

    RTSEND_PROCESS();
}

main() {
    /* Seed random number generator */
    ranSeed = (int)SEC_TIMER;

    /* Initialize the RTS */
    RTSInit();
```

```
    /* Register processes */
    RTSRegCoData(&ping_handler, "ping_handler");

    /* Main loop */
    for (;;) {
        costate RTSprocess init_on {
            wfd RTS();
        }

        costate ping_handler {
            wfd ping_handler_cf(&ping_handler);
        }
    }
}
```

B.4 Common Code

Each of the above programs use the following code to generate random numbers used as parameters in messages (operations).

```
/*************** From Z-World's rabbit/samples/random.c *********************/
unsigned int ranSeed; // Current Random Seed

unsigned int irand () {
    if (ranSeed == 0x5555) ranSeed--;
    ranSeed = (ranSeed << 1) + (((ranSeed>>15)^(ranSeed>>1)^ranSeed^1) & 1);
    return ranSeed;
}

unsigned int random (unsigned int range, unsigned int minimum ) {
    return irand() % range + minimum;
}
```

References

[1] Tak Auyeung. Cooperative multithreading. *Embedded Systems Programming*, pages 72–77, December 1995.

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

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

[4] Justin T. Maris, Aaron W. Keen, Takashi Ishihara, and Ronald A. Olsson. A Comparison of Concurrent Programming and Cooperative Multithreading under Load Balancing Applications. *Concurrency and Computation: Practice and Experience.* to appear.

[5] Justin T. Maris. A Comparison of Concurrent Programming and Cooperative Multithreading under Load Balancing Applications. Master's thesis, University of California, Davis, Department of Computer Science, June 2002.

[6] 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.

[7] 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.

[8] 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*, 15(1):27–53, January 2003.

[9] J. Michael Kamrad II. Ada experience report for BlazeNet, Inc. In *Proceedings of the ACM SIGAda Annual International Conference on Ada Technology*, pages 215–216, Washington, DC, November 1998.

[10] Mario Aldea Rivas and Michael González Harbour. *MaRTE OS: An Ada Kernel for Real-Time Embedded Applications*, volume 2043 of *Lecture Notes in Computer Science.* Springer-Verlag, 2001.

[11] Gregory Bollella (Editor). *The Real-Time Specification for Java.* Addison-Wesley Publishing Company, Inc., Reading, MA, 2000.