Final Report
A multi-tasking environment for real-time control

Prabhas Chongstitvatana
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University

A quick summary

Objective
Create a programming system to support real-time control applications. The aim is to facilitate the writing of multi-task programs. The main use of this system is for control the experimental robots.

Motivation
The application programs for embedded control usually composed of concurrent modules with shared resources which are the common characteristics of multi-task programming systems. By providing concurrency control and protection of shared resources in a programming system, it becomes easier to write an application program. The aim of this research is to provide a programming system which is hardware independent by having an abstract layer separate an application program and the underlying control system.

Method of research
The proposed programming system consisted of a compiler which compile a source language into intermediate codes which will be executed on the target machine by an interpreter. The source language is a new programming language (named R1) that provides concurrency control and protection of shared resources including real-time facilities such as clock, time-out etc. The intermediate code is an architectural neutral byte code aims to be portable and reasonably efficient across many platforms. The interpreter is an abstract machine which execute the byte code, providing the multi-task environment for the target machine.

List of activities
1. Determine the function of the language
2. Design the concurrency control and protection of shared resources to support the language. Determine algorithms to implement those facilities.
3. Design an intermediate code as byte code.
4. Design a source language and its compiler.
5. Implement an interpreter for a microprocessor.
6. Test and debug the compiler and interpreter for the target architecture.
7. Improve the speed of byte code.
8. Implement the interpreter for another microprocessor to demonstrate the independence from specific hardwares.
9. Write reports and documents for the resulting system.

We have accomplished all the above activities.

Accomplishments
1. The language R1 is completely specified. Its features included : concurrency, protection of shared resources via semaphores, symmetric communication between processes, real-time facilities.
2. An intermediate code is fully specified. It is a byte code system which emphasizes portability.
3. A compiler that translates a source language, R1, to the intermediate code, is fully implemented for a PC system and a UNIX system. The compiler works under DOS, Windows 3.11 and Windows 95, SUN Solaris 2.3 producing an intermediate code file which can be run on any target machine.
4. An interpreter is implemented using a high level language C. It is compiled and tested on a Sun Sparc architecture to demonstrate the ability to run on a different platform (such as compile on PC run on Sun). It is fully functional, providing multi-task, interprocess communication and real-time facilities.
5. The interpreter has been ported to run on PC using interrupt timer to provide real-time facilities. This interpreter also demonstrates the portability of R1 intermediate code to run on two different platforms: Sun Solaris and DOS (under Windows 95).

6. One experimental system has been constructed to optimise intermediate code sequences for speed. The work is published in : Chongstitvatana, P., "Post processing optimization of byte-code instructions by extension of its virtual machine", Proc. of 20th Conf. of Electrical Engineering, Bangkok, Thailand, 1997.

Work since the progress report

1. Real-time facilities are completed : access to clock, delay, time-out.
2. The compiler is totally rewritten in C for portability and it is ported from PC to Sun successfully. The structure of the compiler is changed to two-pass compiler hence eliminate all the forward declaration.
3. The interpreter on Sun is ported to PC with some part rewritten for interrupt timer. This is also successful.

Content of this report

Section 1 discusses the design of a concurrent language. Section 2 describes the method for compilation. The detailed account of the compiler and the intermediate code is in section 3. Section 4 provides the discussion of the intermediate code and the supporting environment. Section 5 discusses the issue of real-time, the real-time facilities and the hard real-time scheduling problems. Section 6 gives the detail of the implementation and performance measurement. Section 7 summarises the work. Appendix A provides the grammar of the language R1 in BNF notation. Appendix B contains the semantic of the intermediate code. Appendix C is the published work.

The most uptodate source and bug fixed including the benchmark programs can be found and downloaded from the home page of this project : http://www.cp.eng.chula.ac.th/faculty/pjw/r1/R1.htm

Table of contents

1. Language

This section describes the design of a new language, R1. R1 is a simple language which provides concurrency control, protection of shared resources, interprocess communication and real-time facilities. The author assumes that the reader has some basic knowledge about programming language (such as C or Pascal). This document is not intended to be a "how to" program in R1 but to describe the main concept and some design issues.

1.1 Design goal

The aim of this language is for it to be a small, simple, and practical language for programming an embedded application. The scope of the language is not ambitious because the emphasise of this research is more towards the run-time environment aspect of the language. The design decision is based on providing the minimum necessary function for multi-tasking, protection of shared resources, interprocess communication and real-time facilities. R1 has a minimum set of features to satisfy all these requirements.

1.2 A short summary of the language

R1 is a typeless, concurrent language. It has static process (the number of process is fixed at compile time, cannot dynamically create a process). It is a single load system (no module, everything is known at compile time) and has no dynamic storage. The method of interprocess communication is symmetric synchronous message passing. The protection of shared resources (which are global data) is via semaphores (by specifying critical regions). Scheduling system is a time-slicing round-robin policy. It provides the following real-time facilities : access to clock, delay, and time-out.

Syntax

The syntax of R1 is intentionally made to be "like" C language. So that the user who is familiar with C language can read and write R1 easily. Informally, the syntax can be described as following (reserved words are in **bold face**) :

```
// comment
global  var-list ;
semaphore sem-list ;
function-name (formal-param-list) { body }
process process-name (formal-param-list) { body }
main ( ) { body }
```

`body` composed of sequence of statements :

statements	examples
`assignment`	`a = b + 1 ;`
`if-statement`	**`if`** `( expr ) stmt-true [` **`else`** `stmt-false ] ;`
`while-statement`	**`while`** `( expr ) stmt ;`
`return-statement`	**`return`** `( expr ) ;`
`function-call`	`function-name ( actual parameters ) ;`
`process-call`	`process-name ( actual parameters ) ;`

where var-list is the list of variable name separated by ",", sem-list is similar (but semaphores are special variables), formal-param-list is the usual formal parameter list of a function declaration and body is a usual sequence of statements.

A program is composed of declarations and a main. There are four types of declaration : global variable declaration, semaphore declaration, function declaration, and process declaration. There are several types of statements : assignment statement, flow-control if and while, and some additional function for concurrent processing. Operators are basic operators such as + - * / etc. including addressing operators '*' (dereference) and '&' (address). The scalar data is basically a word (which can be 16 bits or 32 bits depended on the architecture of the target hardware) with no type, so all operators can be applied indiscriminately (this is both good and bad). Only structured data is a one dimensional array of words. Global variables must be declared. Local variables are automatically declared and they are lexical-scoped. Local variables appear in the formal parameter list and can not be an array variable.

An array variable is index by 0..max-1 (with the syntax array-name[index]) when max is the size of array (declared in global declaration). The value stored in a variable can be any type : character, integer or pointer. There are two operators for accessing a variable indirectly, * (dereference) and & (address) (both has the meaning similar to the same operators in C language). These two operators are used to pass parameters "by reference" (same as using "var" in formal parameter declaration in Pascal or '*' in C). An example, a function increment which pass parameter by reference :

```
increment ( n ) {  *n = *n + 1; }
main ( ) {  a = 1;  increment(&a); }
```

A statement is "like" a statement in C language. It is based on "expression" and *must* ends with ";" except for a sequence of statements in "{…}". A statement can be null (no operations), i.e. ";" and "{}". "if" statement is similar to C and Pascal, the dangling "else" (when nested if) belongs to the last if and can be unambiguous by using "{…}". Please note that the use of ";" is like C which is different from Pascal, especially in Pascal the statement before the else must *not* ends with ";".

Pascal	`if a = b then c = 1 else c = 2;`
C	`if( a == b ) c = 1; else c = 2;`
R1	`if( a == b ) c = 1; else c = 2;`

The precedence of operators
List below from the highest (top) to lowest (bottom) :

`( )`	do an expression inside ( ) first
`- ! * &`	unary op - minus, ! logic not, * deref, & address
`* / &&`	multiply, divide, logical and
`+ - ||`	add, subtract, logical or
`< <= == != >= >`	relational operators

The type of operation is : word `X` word `->` word. Therefore the overflow and underflow can occur. All arithmetic operators treat a value as a signed integer (the divide is integer divide). Please note that the set of operator is far from complete, there are a lot of useful operators that have been left out such as : mod, bitwise logical ; and, or, not, xor, shift left-right, increment, decrement etc.

Example of a R1 program

```
//   bubble sort
global data[10] ;
swap(a,b)
{
   t = data[a]; data[a] = data[b]; data[b] = t;
}
sort()
{
   i = 10;
   while(i) {
     j = 1;
     while(j < i) {
       if ( data[j] < data[j+1]) swap(j,j+1);
       j = j+1;
     }
     i = i-1;
   }
}
main() { ... }
```

1.3 Process

A process in an independent computation which can run concurrently with other process. Process declaration is like a normal subprogram or function. Initial values can be passed as parameters at the starting time of a process. A process will end its execution by self terminating when it executes the last instruction at the end of program (this is different from the execution of a subprogram or a function which ends its execution by returning to the caller). A program that calls a process will start that process execution, that program then will continue the work without waiting.

A process in R1 is static, i.e. it can be created and run immediately by a caller but the termination of that process cannot be perform by anyone except itself (when it executes the last instruction at the end of a program) Therefore a process can never return to its caller. There are two ways to communicate (passing some value) between processes :

1. by share variables
2. by message passing

1.4 Share variables and semaphores

A process can access to share resources. Because of the concurrency there must be some guarantee for mutual exclusion, i.e. only one process can access to a share resource at a time. The standard method is by using semaphores (Dijkstra 1968). A semaphore is a special variable, it has a different structure from a simple (scalar) variable. We can imagine the use of semaphore to guarantee mutual exclusion as if using a "lock". A semaphore is a lock. Before using a share resource, the user must "lock" that resource to prevent other process from accessing that resource. The user then "unlock" it when finish. There are two operations on a semaphore `wait(sem)` and `signal(sem)` similar to lock and unlock. The part of program that must be guaranteed of mutual exclusion is called "critical region". To use a semaphore for mutual exclusion, `wait()` and `signal()` must be issued by the same process. To facilitate writing a critical region a special syntax is provided :

```
region(sem) { stmt }
```

which the compiler will generate

```
wait(sem)  stmt  signal(sem)
```

To use a semaphore for mutual exclusion, the initial value of that semaphore must be 1 and there must not be any other operation on that semaphore. A semaphore can be access just like a variable.

Beside using semaphores for mutual exclusion, there are many other uses such as for resource allocation and process synchronization but it is becoming more complex. The details of the use of semaphore for other purposes can be found in the text book of operating system (such as Whiddett 1987, Bacon 1993).

1.5 Message passing

Semaphores are used to protect share resources and global variables are accessed in a critical region to pass values between processes. Another method to pass values between processes is by sending a message, in R1 there are

```
send(pname, message)
```

and

```
receive(pname, &message)
```

A message is a word. The send-receive is synchronous, i.e. a sender will stop and wait until the receiver receives the message before resuming its execution (vice versa, when a receiver waits for a message that has not been delivered yet it will stop and wait) . Another use of send-receive is to synchronise processes. The lack of guarded commands (Dijkstra 1975) in R1 language prohibits nondeterminacy therefore messages cannot be used for a more flexible communication pattern (such as having a priority message or creating a post office process).

1.6 Real-time facilities

There are several functions provided :

1. Access to clock : `gettime(&t)` read the system clock.
2. Delay : `delay(t)` , delays the execution of a process for t time units.
3. Time-out : use the extended semantic of wait for semaphore and message passing to include time-out (the `after` clause). The syntax are :

   ```
   wait(... ) after( t ) { body }
   send(... ) after ( t ) { body }
   receive(... ) after ( t ) { body }
   ```

   where body is the statements that will be invoked if time-out occurs.

1.7 Example : a producer-consumer problem

```
process producer( n ) {
    while( n ) {
       send ( consumer, n );
       n = n - 1;
   }
}
process consumer( n ) {
   while( n ) {
      receive( producer, &m );
      ....  //  do something with m
      n = n - 1;
   }
}
main ( ) { producer(5); consumer(5); }
```

1.8 Design justification of R1

Many design decisions are made based on the feasibility to do hard real-time scheduling. The real-time scheduling is based on the concept of Temporal Scope which will be discussed in more details in section 5. Some of the main design issues are discussed below :

1. No dynamic memory allocation and no dynamic process creation, to make it possible to calculate the execution time of each Temporal Scope.

2. Use semaphore, it provides a simple model of shared memory for programmer. It can be implemented efficiently. It is also flexible to use in other purpose such as for resource allocation (but its use will be more complex).

3. Use synchronous message-passing, it combines communication and synchronisation in a single high-level primitive. Other alternative model of message-based process synchronisation are :

a. asynchronous (no wait)
b. remote invocation

There are relationships between asynchronous, synchronous and remote invocation semantics. Two asynchronous events can constitute a synchronous relationship if an acknowledgment message is always sent (and waited for). Two synchronous communications can be used to construct a remote invocation. It could be argued that the asynchronous model gives the greatest flexibility but there are a number of drawback :

a. Potentially infinite buffers are needed to store messages that have not been read yet.
b. In asynchronous model, more communication are needed, hence programs are more complex.

Also, a synchronous model can emulate an asynchronous communication simply by using a buffer process.

4. Use direct naming, to facilitate the use of priority inheritance scheme for dynamically change the priority of the process. Priority Inheritance (Sha and others, 1987), a process's priority is not static; if a process *p* is suspended while waiting for process *q* to undertake some computation then the priority of *q* becomes equal to the priority of *p* (if it were lower to start with). However, please note that the current implementation uses the round-robin scheduling policy without priority scheme (for simplicity).

1.9 Related work

Algol-68 was the first language to introduce semaphores. Semaphore is considered to be an elegant low-level synchronisation primitive. However the use of semaphores can be error prone. The syntactic sugar `region( )` which is introduce in R1 is a design from an early concurrent system (Dijkstra, 1968). An abstract data type for semaphores can be constructed in current languages Ada and Modula-2 (Hoppe 1980). A further refinement of the semaphore concept is to encapsulate critical regions into a single module call a monitor. The design and analysis of monitor structures was undertaken by Dijkstra (1968), Brinch-Hansen and Hoare (1974). Message passing is a single high-level primitive that combines communication and synchronisation. It forms the basis of both Ada and Occam-2. Synchronous send is used in CSP (Hoare, 1978). Remote invocation is found in Ada, SR, (Andrews and Olsson, 1986), a synchronised communication is often called "rendezvous" and the remote invocation is called "extended rendezvous". Burns and Wellings (1997) describe various programming languages for real-time systems.

2. Compiler

This section describes a compiler and intermediate code and how the compile translates source language to intermediate codes.

2.1 Formal syntax

A formal syntax of R1 is in the appendix A.

```
unaryop is    { -, !, *, & }
binaryop is   { +, -, *, /, &&, ||, <, <=, ==, !=, >=, > }
```

Precedence of operators (from highest)

( )	do an expression inside ( ) first
- ! * &	unary op - minus, ! logic not, * deref, & address
* / &&	multiply, divide, logical and
+ - ||	add, subtract, logical or
< <= == != >= >	relational operators

Built-in function and special forms

[ ] $_{opt}$ = optional

```
region(sem) statement
wait(sem) [ after(t) statement ] opt
signal(sem)
send(pname,message) [ after(t) statement ] opt
receive(pname,&message) [ after(t) statement ] opt
print( actual-parameter and string constant )
gettime(&t)
delay(t)
```

2.2 Recursive descent compiler

The compiler is a two-pass recursive descent compiler. The first pass scans the source program looking for all declarations. This pass generates and intermediate file named "tmp000.bak" which is the input file for the second pass. The second pass scans all the statements and generates intermediate codes (called I-code). The whole I-code will be kept in a buffer, when the source program is translated successfully to the end, the I-code in the buffer will be post processed (for “improvement”, will be explained in the later section) before storing it to a file. For sake of efficiency the syntax is rewritten in the form of LL(1) grammar, which has a property that there is only one look ahead symbol and there is no backtrack. The compiler is written according to the flow of the grammar in LL(1). An example of LL(1) grammar for parsing an expression :

```
expr ->       term exprs
exprs ->      relop term exprs | null
term ->       fac terms
terms ->      addop fac term | null
fac ->        item facs
facs ->       mulop item facs | null
item ->       num | -num
              var | -var | * var | & var | ! var
              pname
              fname()
              ( expr ) | -( expr ) | !( expr )
var ->        variable | array[index]

where  relop is { <, <=, ==, !=, >=, > }
       addop is { +, -, || }
       mulop is { *, /, && }
```

2.3 Name of variables

A variable name in R1 language is case-sensitive (like C). A name of variable is composed of the character `'A'..'Z', 'a'..'z', '_', '0'..'9'`. It must not begin with a digit. The limit of the length of name is 30 characters (limited by this implementation). The followings are the reserved words :

```
after, delay, else, gettime, global, if, main, print, process,
receive, region, return, semaphore, send, signal, wait
```

The followings are special characters `( ) { } ; , " + - * / = & | ! < >`
The comment begins with "`//`" and will be skipped to the end of line.

2.4 I-code

The I-code has two forms

a. one opcode and no operand
b. one opcode and one or more operands.

We will use the notation `[ I-code ]` refers to a whole I-code, an opcode is one byte. `[ I-code #operand ]` is I-code with operand(s). An operand is two bytes and will be prefix with "#". Appendix A shows the table of all I-code.

2.5 Symbol tables

When a compiler comes across an identifier (id) or function name (fname) or process name (pname), it will store information about that name in a symbol table. Each entry in a symbol table contains the following fields :

name :	string
type :	variable, array, fname, pname
reference :	the location in the memory
argument:	parameter of this entry (such as size of array)
process_id :	for process

A symbol table is organised as a hash table. There are two symbol tables, one for the global symbols, i.e. the variables in the global declaration, semaphore declaration, function name and process name. All the local variables (in the formal parameter and in the scope of function or process declaration) are stored in a separate symbol table called local symbol table. In searching for a symbol, the local symbol table will be searched first then the global symbol table. During compilation, a name will be translated into its reference: a global symbol reference is its address in Data segment, a local symbol reference is its "ordering" number (ordered by its first appearance). This is actually the reference to its slot in the run-time stack frame.

3. How to compile

3.1 Global and semaphore declaration

The declaration of global variables and semaphores affects the allocation of Data segment, which is addressed as word. Each variable will be assigned a reference, allocation of Data segment is as following :

1. a simple variable occupies one word.
2. an array variable occupies as many words as its size.
3. a semaphore occupies two words, the first word stores its value, the second word stores its list of waiting process. (the mechanism of semaphore is described in the section "interpreter")

Its reference is :

1. a simple variable reference is its location in Data segment
2. an array reference is the location of the first element.
3. a semaphore reference is the location of its first word.

3.2 Function and process declaration

The declaration of function and process affects the allocation of Code segment, which is addressed as byte. The reference of function or process is its location of the first byte code.

Formal parameters will be stored and counted. When compiling the body, the local variables will be stored and counted. After a complete compilation of a function or process, the local symbol table will be cleared. I-code are as follows :

```
[Func, #nformal, #nlocal ]
[Proc, #pid, #nformal, #nlocal ]
```

where #nformal is the number of formal parameters, #nlocal is the number of local parameters. (Note that nparam + nlocal = total number of local variables of that function of process). The #pid (process id) of a process is a unique number and is assigned according to these rules :

pid	comment
0	special value, don't use
1	the main( ) process
2..n	other processes

at the end a function declaration is ended with I-code "Ret0", and a process declaration is ended with I-code "Stop".

The main program `main()` is compiled similar to a function but don't need to enter the symbol table (as no one can refer to the main process).

3.3 Body of function and process

The body composed of sequence of statements. A part of a statement is an expression. The compilation of an expression will be discussed first. In parsing an expression, the precedence of operators are satisfied by the grammar. The operators are transformed from prefix and infix to postfix. The following discussion is applied for the case of right hand side expression, or getting the value of an expression.

Variables

A variable that appears in an expression is translated into its value (called Rvalue), in this context it is often appears at the right hand side of an assignment statement. The I-code is "Rvalue" followed by the reference to that variable :

```
[ Rvalue  #reference ]
```

If the variable is global, its I-code is "Rvalg" and its reference is its location in Data segment. If the variable is local, its I-code is "Rval" and its reference is its "ordering" number. The reference is found in the symbol table's entry associated with the name of that variable.

Array variables

For the right hand side expression, an array variable is translated into its reference (called Lvalueg , as it is global) followed by the I-code for the expression of its index and "Index" and "Fetch" :

```
[ Lvalueg #reference ]  expr for index [ Index ]  [ Fetch ]
```

Numbers
A number is translated into the I-code "Literal" followed by its value :
```
[ Literal  #value ]
```

Operators
An operator is translated into one byte I-code :
```
[ Op ]
```
Operators are : add, sub, multiply, divide, LT, LE, EQ, NE, GE, GT, not, and, or.

Dereference operator
To dereference a variable (to get its value indirectly) after translated that variable, another I-code "Fetch" will be added :
for a simple variable
```
[ Rvalue #reference ] [Fetch]
```
for an array variable
```
[ Lvalueg #reference ] expr for index [Index] [Fetch] [Fetch]
```

Address operator
To get address of a variable, instead of "Rvalue", the "Lvalue" is used. For an array variable, there is no "Fetch".
for a simple variable
```
[ Lvalue #reference ]
```
for an array variable
```
[ Lvalueg #reference ] expr for index [ Index ]
```

Statements

Assignment statements
For an assignment of the form :
```
id = expression
```
the left hand side is to get the address, the I-code is "Lvalue" or "Lvalueg", followed by the I-code for the right hand side expression, followed by I-code "Set".
```
[ Lvalue #reference ]  expr [ Set ]
```
for
```
*id = expression
```
the left hand side is "Rvalue" or "Rvalueg" (for getting the "value" of that variable as an address)
```
[ Rvalue #reference ] expr [ Set ]
```

Control flow statements
The flow of control is translated into "Jump" (to a location in Code segment). Sometime the destination is not yet known because it may refer forward of the current location. The location that a forward reference appears will be remembered and after the destination of the jump is known, the reference will be updated.
```
if expr stmt =>                 expr [ Jz #1 ] stmt <1>
if expr stmt else stmt =>       expr [ Jz #1 ] stmt [ Jmp #2 ] <1> stmt <2>
while expr stmt =>              <1> expr [ Jz  #2 ] stmt [ Jmp #1 ] <2>
return expr =>                  expr [ Ret ]
```
For "return" there are two types : "Ret0" for no return value, "Ret1" for returning a value. Also the "Ret0" will be automatically append at the end of the body of a function.

Function and process call
```
name( actual parameter ) =>     expr of actual parameters [Call #reference]
```
The actual parameters are parsed as expressions, followed by "Call".

Special forms
The special forms are the built-in functions : send, receive, wait, signal, print, delay, gettime.

Send and receive
For safety purpose, the process name is restricted to be a constant. The process name cannot be passed in a variable. This is to avoid the error in handling the process id. In parsing the actual parameters, the first parameter is the process id. The name of the process has the form "pname" which will be translated into its pid which will become a literal :

```
[ Literal  #pid ]  ... [ Send ]
```

Wait and signal

The only parameter is a semaphore which is translated into the reference to that semaphore :

```
[ Wait #reference ]
```

Print

The actual parameter is a variable length list. Only two types of value allows in the list :

1. a value of an expression :

```
expr [ Print ]
```

2. a string constant, because there is no string data type in R1 language the string constant is decomposed into a sequence of printing a character :

```
[ Literal #character ] [ Printch ]
```

a string constant can embed a newline character by using "\n" (similar to C embed the newline character in the format string of printf( )). An example, "AB\n" is translated into :

```
[Lit #'A'] [Printch] [Lit #'B'] [Printch] [Lit #nl] [Printch]
```

3.4 Post-processing of I-code

It is necessary to keep a buffer of the whole I-code while compiling a program. This is because the forwarding reference may required that the location as far back as the beginning of the Code segment is updated. Having a buffer of the whole I-code is useful when some improvement of the I-code is performed. We will discuss a few techniques.

Renaming local variables

The references of local variable are "ordering" from 1..N (where N is the total number of local variable in a scope) starting with the first formal parameter as no. 1. Because of the policy of the interpreter, it is easier (and faster for the interpreter) to have the first parameter numbering as no. N. (this particular interpreter uses an overlap stack frame to avoid copying the actual parameters to its stack, the detail can be found in the document "interpreter"). The renaming operation will rename the reference of the local variables from 1..N to N..1.

Peep-hole optimisation

There are many other possible post processing of I-code to improve either code density or the speed of execution of I-code. For example, a peep hole optimization can be performed (to improve the speed of execution). Some code sequence can be made shorter :

```
if ( X == 0 ) ... =>          X  [Literal #0]  [EQ]  [Jz #.. ]
                 to           X  [ Jnz #.. ]
```

The code density can be improved if some I-code is designated special small literal (such as 0,1,2, -1). The small literal is used quite often in a program.

```
[ Literal #0 ]  (3 bytes )   =>   [ Lit0 ]  ( 1 byte )
```

or changing the address of a jump instruction to be relative and if the offset is small enough to fit in one byte, assigned a special I-code for it :

```
[ Jmp #ads ] (3 bytes )      =>   [ Sjmp offset ]  (2 bytes )
```

Extending the virtual machine

By adding instructions to the virtual machine, some sequences of byte-code can be represented by a shorter code which can be executed faster. This is a bottom up approach which improves the speed of execution without recompiling the source program. The frequency of use of some sequences of byte-code (grouping as basic blocks) is analysed and special byte-codes are designed to substitute these sequences. See table 1 and 2 below (full details can be founded in the appendix) :

Table 1.1 The most frequently used sequences

byte-code sequence	correspond to
lval a, rval a, lit 1, plus, set.	a = a + 1;
lval b, lvalg c, ... , index, ...	b = c[..] ...
lvalg c, ... , index, ...	c[..] = ...
lval a, lit 0, set	a = 0;
lvalg c, ..., index, lit 0, set	c[..] = 0;
lval a, rval a, exp, plus, set	a = a + exp;
lvalg c,..,index,lvalg c,..,index, fetch,..,plus set	c[n] = c[n] + ...
rval a, rval b, EQ, Jz	if (a == b)
rval a, lit 0, EQ, Jz	if (a == 0)
lvalg c, ..., index, rval b, LE, Jz	if (c[..] <= b)
rval a, rval b, LT, Jz	while (a < b)

Table 1.2 The extended byte-code

extended byte-code	for the sequence
inc v (dec v)	lval v, rval v, lit 1, plus, set.
addset a	lval a, rval a, exp, plus, set.
set-var a	lval a, ... set.
set-0 a	lval a, lit 0, set
EQjz a b $1	rval a, rval b, EQ, jz $1
Jnz a $1	rval a, lit 0, EQ, jz $1
LEjz a b $1	rval a, rval b, LE, jz $1
LTjz a b $1	rval a, rval b, LT, jz $1

An experimental system has been constructed using this optimisation technique. Running Stanford integer benchmark suit (Hennessy and Nye) this technique yields 25% - 120% speedup (means it is faster as much as 2.2 times as before the optimisation) with 10% -30% code size reduction (Chongstitvatana, 1997), see Appendix C.

4. Interpreter

This section describes an interpreter for the byte code, I-code, of a concurrent language R1. The I-code is the output of compiling the source language R1. I-code can be used on both 16 bit and 32 bit system. The I-code is a low level instruction set of a virtual machine (VM) which is based on stack architecture.

4.1 Why I-code ?

By defining an intermediate code as an executable specification, the output from a compiler becomes portable. I-code is architectural neutral, i.e. it runs on many different machine architectures. The intermediate code can be executed both on 16-bit and 32-bit systems* where the interpreter is available.

(* The source program must be written with the portability in mind, for example, if an integer is used to hold a value larger than +32767, that program cannot be ported to a 16 bit system, or if a data is allocated larger than 64K, it will not work with a machine that has 16 bit address.)

Porting the interpreter to a different platform is not difficult, as the interpreter is written in a high level language with the goal of minimising the part that is machine dependent or that must be written in an assembly language. Some characteristic of machine unavoidably has implication for the speed of execution of the interpreter, for example, the architecture should be able to access the individual byte, to fetch the byte code from the code segment of the interpreter efficiently. The portability and architectural neutral is the important goal of the I-code. It is also the main goal of the current popular language "Java" (Gosling and McGilton 1996, Tribble 1996) and this goal can be traced back through the use of byte code in the implementation of Smalltalk (Goldberg and Robson, 1989). The use of virtual machine instruction set as a target for a compiler also has a long history, a popular and well known system in the past is P-system which Wirth used it to distribute the Pascal system (P4 compiler).

4.2 Stack machine

The virtual machine defined for executing the I-code is a stack machine. The operators act on the operands on the stack. The stack is as a working area and also to pass parameters to a subprogram. The memory model of this VM consisted of :

1. A code segment, which stored the instructions (the I-code).
2. A data segment, which stored global variables.
3. A stack segment, which is used for computation and stored the thread of execution.

The data segment and stack segment resided in the same address space. This is because the global and local variables share the same address space. Please note that there is no dynamic memory (often called "heap").

There are three special registers of the VM : an instruction pointer (Ip), points to the current instruction in the code segment, a frame pointer (Fp), points to the current stack frame (more explanation later) and a stack pointer (Sp), points to the top element on the stack. The state of computation consisted of these three registers (Ip, Fp, Sp).

The stack architecture was very popular and can be dated back quite far, from the Burrough machine with a version of an early multi-tasking operating system. With the current VLSI technology, the register-based architecture dominates the computer design. For a more current discussion about modern stack architecture, the readers are invited to consult Chapman's book. (Chapman, 1988). Presently, one of a commercial CPU that is being designed especially for byte-code interpreting is based on stack architecture (Picojava 1996). Picojava is a special CPU which executes Java byte-code, aims for a low power, embedded application market, such as Network Computers and hand-held devices.

4.3 Stack frame

The state of the current stack is called "stack frame" (or activation record). Beside the working area in a stack frame, there is an area for local variables and an area to store the state of computation (of the previous stack frame). A stack frame is shown in the figure below (fig. 4.1) :

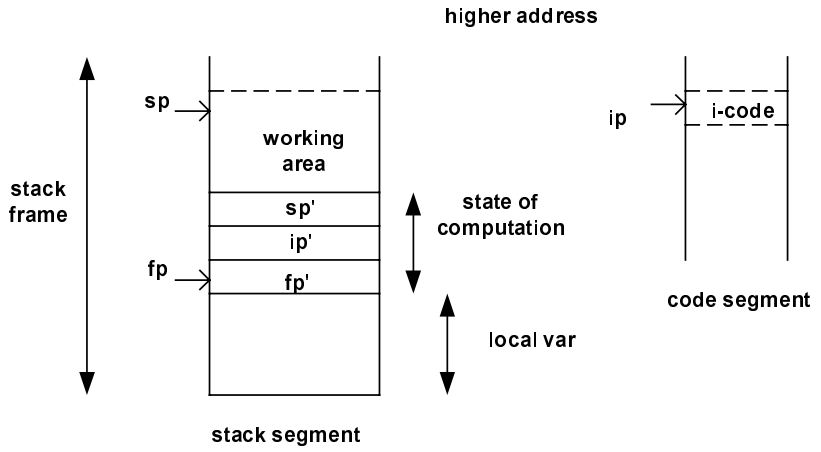

Figure 4.1 Stack frame

The local variables are accessed by referring to their locations in relative to Fp, for example, the location of the first local variable is the location at Fp - 1.

4.4 Parameters passing

A stack is used to pass and return parameters between the caller and the callee. When a call of subprogram (called "function" in R1 language) occurs, a new stack frame will be created to start a new computation of the callee, and the passing parameters are transferred to it (usually by copying the passing parameters from the caller's stack to the callee's local variables). The passing parameters of the callee are declared as its formal parameter. For example, when call a function with 3 parameters, the state of both stack frames are shown as follows :

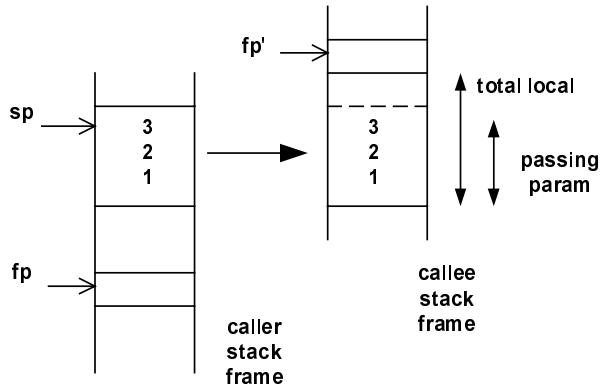

Figure 4.2 Caller's stack and callee's stack. Show how parameters are passed.

If the stack frame of callee is overlapped with the stack frame of the caller, there will be no need to copy the passing parameters. In order to use overlapping stack frame, the ordering of the local variables must be shuffled, i.e. the first parameter must be numbered with N instead of 1 (where N is the total number of local variables). Therefore instead of numbering the local variables from 1..N according to the order in which the variables appear, the numbering will become N..1. The renaming step is the part of the post processing of I-code and can done by the compiler. Please note that the numbering needs not to be this way if a different method to pass parameters is employed.

4.5 I-code format

There are 4 formats : zero operand, one and two and three operands (all operands are 16 bits).

`[ Ic ]`	Arithmetic op, Logical op, etc.
`[ Ic #ref ]`	Literal, Lval, Rval, Jump, Wait, etc.
`[ Ic #nparam #nlocal ]`	function call
`[ Ic #pid #nparam #nlocal ]`	process call

4.6 Operational semantic of I-code

Notation for describing the operational semantic of I-code,

CS[i] code segment at the address i.
DS[i] data segment at the address i.
SS[i] stack segment at the address i.
Both data segment and stack segment resided in the same address space and can be denoted by M[i] memory contained DS and SS, at the address i.

Some operations take operand(s) from stack and leave a result on the stack. The state of stack can be described by a notation that indicates the values in the stack (before an operation – after an operation) when the top of stack is the left most item, the "..." denotes the items that are of no interested to us.

Push(x) is defined as
Sp = Sp + 1, SS[Sp] = x
x = Pop is defined as
x = SS[Sp], Sp = Sp - 1

Literal

push the constant into the stack

`[Literal #n ]`	push(n)	(... – n)

Variables access

Lvalue, push the address of that variable. If the variable is global its reference is its address. If the variable is global, its address is the address of that variable in the current frame. Rvalue, push the value of that variable.

`[Lvalueg #ref ]`	push(ref)	(... – ref)
`[Lvalue #i ]`	push(Fp-i)	(... – ads)
`[Rvalueg #ref ]`	push(DS[ref])	(... – value)
`[Rvalue #i ]`	push(SS[Fp-i])	(... – value)

Fetch, get the value of the variable which its address is on the stack. Set, store a value to an address, both value and address are on the stack. Index, which is used to access an array variable, calculates the address of that element of the array and leaves the result on the stack.

`[Fetch]`	push(M[pop])	(ads – value)
`[Set]`	M[pop1] = pop2	(ads, value – ...)
`[Index]`	push(base_ads + index)	(base, index – ads)

Transfer of control

Jmp, Jz, jump to a location, without and with condition (Jz is jump if the top of stack is zero).

`[Jmp #ads ]`	Ip = ads	
`[Jz #ads ]`	if pop = 0 then Ip = ads	(bool – ...)

Call, push the return value and jump to that address to continue with the execution of a subprogram (a function or a process). For a function, a new stack frame is created, the parameters are passed (by overlap stack frame) and the current state of computation (of the caller) is saved, the execution is continue with the code of that function. For a process, a new process descriptor is created, its state of computation is initialised and the process is awaked. Stop, terminate a process by removing its process descriptor from the ready list. (discussion about process is in the next section). Return has two varieties : Ret0, and Ret1. Both instructions remove the current stack frame, restore the previous state of computation (which is stored in the current frame). For Ret1, a value will be returned to the caller's stack.

`[Call #ads ]`	push(Ip), Ip = ads	(... – return_ads)
`[Func #nparam #nlocal ]`	save state, new stack frame, pass parameters	
`[Proc #pid #npara #nlocal]`	new process descriptor, initialise state, awake	
`[Ret0]`	remove stack frame, restore state	
`[Ret1]`	remove stack frame, restore state, return a value	
`[Stop]`	terminate the process	

(Note that nparam + nlocal = total number of local variables of that function or process)

Arithmetic and Logic operators

Binary arithmetic operators (Aop) are Add, Sub, Mul, Div. They will take 2 operands from the stack and return the result. These are arithmetic on integer. Unary operators (Uop) are Minus and Not, will take one operand

from the stack and return the result. Binary logic operators (Lop) are LT, LE, EQ, NE, GE, GT, And, Or. They are similar to arithmetic operators except the result is boolean.

`[Aop]`	push (pop1 Aop pop2)	(a, b – result as integer)
`[Lop]`	push (pop1 Lop pop2)	(a, b – result as boolean)
`[Uop]`	push (Uop pop)	(a – result)

Output

Print, take an operand from stack and print it out as an integer. Printch is similar and print as a character.

`[Print ]`	print an integer	(a – ...)
`[Printch ]`	print a character	(c – ...)

4.7 Process

A process is a program in execution. In R1 system, several processes can be active at the same time. The implementation of concurrency is multi-threading (which is also called "light weight process"). A heavy weight process is a process with a separate address space and need virtual address mapping to physical address, may need a MMU. A light weight process has single address space. A thread is a trace of execution, a single thread process is resulted from co-operative process. A multi-thread process has several traces at the same time, can be accomplished by pre-emptive scheduler with time-slicing.

4.8 Run-time environment

Each process will have its own stack segment. In this implementation, there is single address space, the stack segment of all processes are in the same address space. The advantage is that there is no translation between virtual address and physical address therefore it is fast and simple. The disadvantage is there is no protection between processes. Each process has its process descriptor (PD) to store the necessary information, that is :

Pid	is the process id. It is a unique number to identify each process.
Link Previous, Next	use to link the list of processes.
Status	indicates the status of the process { ready, running, wait, ... }
Ip,Fp, Sp	save the state of computation.
In -, Await-Mail box	keep the list of mail that is sent to this process or that is awaiting a mail from this process.

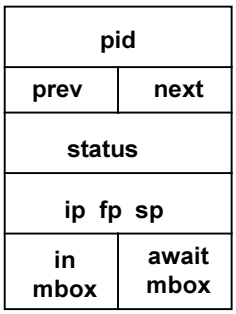

Figure 4.3 Process descriptor

When a process is created and is ready to start the execution, its PD will be linked to the "ready list" which is a doubly linked circular list used by the scheduler. A scheduler has the duty of selecting a process to run from the ready list. The scheduling policy is a Round-Robin policy with an equal time-slice for every process. A scheduler will enable a process in the ready list to run until its time-slice is over and then switches to the next process in the list. If a process enters a "wait" status (it is said to be blocked, usually because it performs some operation that requires waiting for another process, such as waiting for the receiver to receive a mail), its PD will be removed from the ready list. The process in "wait" status can be awaken to enter "ready" status by inserting its PD into the ready list (usually at the end of the list). To perform the switch from one process to another (called context switch), the current state of computation (Ip, Fp, Sp, of the active process) is saved in its PD and the state of computation of the process to be run is restored. The first process to be active is the process to run the main program.

A doubly linked circular list is an appropriate structure for the PD list. The scheduler uses the Round-Robin policy therefore needs a circular list. The use of doubly linked is necessary because a PD can be inserted and removed at the middle of the list.

4.9 Message Passing

Two processes can communicate by sending and receiving mails. A mail is a data structure with three fields : Id, stored the pid of the owner of the mail, Message, stored the message, and Next link stored the link to the list of mails (fig. 4.4). The list of mails is a singly linked list. Sending and receiving messages is a one to one communication and the pid of the sender and the receiver must be specified. The communication is synchronous, that means the sender will wait (goes into the "wait" status) for the receiver until the mail has been received, vice versa, if a receiver executes a receive instruction it will wait until the mail arrived. To avoid "busy wait", the process that is in "wait" status will be removed from the ready list. It will be awaken by the partner whom it communicated with. For example, the sender sent a mail and waited, that mail was linked to in-mailbox of the receiver immediately. When the receiver wanted to receive a mail it would check its own in-mailbox for the mail from that particular process (by looking at the Id of the mail). Once the receiver retrieved the mail it would awaken the owner of the mail.

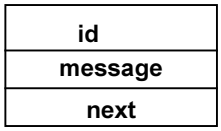

Figure 4.4 A data structure of a mail.

Similarly, if a receiver wanted to receive a mail and there was no mail in its in-mailbox, it would sent a mail with an empty message to the await-mailbox of the process it wanted to communicate with to notify that process that there was a process waiting for its mail. The receiver then would be waiting. When that process sent its mail, it would checked in its await-mailbox and found the awaiting notification from the receiver process. It would put the message directly to the receiver process's stack and would awaken the receiver. Send and receive is described below (where p is a process id) :

```
send( p, message)  is
        search its own await-mailbox for a notify mail from p
        if found (p is waiting), retrieve the mail, put the message to p's stack and awake p.
        if not found, link mail to the in-mailbox of p and self enters "wait".
receive( p, message )  is
        search its own in-mailbox for a mail from p.
        if found ( p is waiting ), retrieve the mail, get the message and awake p.
        if not found, link an empty mail to the await-mailbox of p, self enters "wait".
```

It is interesting to note that the send and receive have a symmetry of operation. The list of mail is implemented by a singly linked list. It is an appropriate structure because the operation "insert" for the list can be done at the head of the list. This is possible because the order of mail in the list is not important as there can be only one mail from one process (because the communication is synchronous when the sender sends one mail it will have to wait until the receiver retrieves it). The "retrieve" operation is done while doing "search" therefore a back-link is not required (as search operation will already traverse the list therefore it has to keep two pointers, one points to the previous node).

`[Send ]`	do send operation, pid and message are on stack	(pid, mess – ...)
`[Receive ]`	do receive operation, pid on stack, store message to mess	(pid, mess – ...)

4.10 Semaphores

A semaphore is a mechanism to protect share resources. It was invented by Dijkstra (1968) to solve many problems in the early days of the use of computer to do many things at once. A semaphore is a special variable. By using "wait" and "signal" with a semaphore many useful synchronisation of processes can be achieved. "Wait" and "signal" has the following semantic :

```
wait ( sem )
        if sem <= 0 wait
        else sem = sem - 1
signal ( sem )
        sem = sem + 1
```

The “wait” and “signal” must be atomic operations, i.e. its operation will run to completion without any interrupt from other process. This is to guarantee mutual exclusion of the semaphore variable. To avoid “busy wait” the process that enters “wait” status will be removed from the ready list and will be awaken by the corresponding “signal”. A semaphore variable has 2 fields : value and waiting list. The “value” field holds the value of a semaphore. The “waiting list” field holds the list of the processes that wait on that semaphore. The implementation of “wait” and “signal” without “busy wait” is as follows :

```
wait ( sem )
        if sem <= 0 self enters “wait” status, insert its PD to the waiting list of sem
        else sem = sem - 1
signal ( sem )
        check the waiting list if there are any waiting process
        if found remove the first process in the list and awake it.
        else sem = sem + 1
```

The waiting list is a First In First Out queue therefore the selection of a process to be awaken (in case that there are many processes in the list) will be fair to all processes. The list must be inserted at the end and removed at the head, which requires to have a pointer to the end of list to avoid traversing the list. Because the waiting list is a list of PD and PD already has two links therefore FIFO list can be implemented.

`[Wait #sem]`	do wait operation on sem
`[Signal #sem ]`	do signal operation on sem

5. Real-time facilities

For real-time applications some operations concerning time are necessary. The system time is implemented using a 32-bit counter. This is incremented by the real-time clock of the underlying system. For the PC with DOS, we have implemented the system time using the DOS timer interrupt which has the frequency 18.2 Hz, i.e. the clock will be increment 18.2 times per second or a task-switch occurs every 54.5 ms (on UNIX system, because it is already multi-task, its timer can be used directly to set to any desired interval). The main loop of the interpreter is as follow :

```
while( activep != nil ) {
    execprim(cs[ip]);
    if( flagtimeout ) {
        flagtimeout = 0;
        switchp();
    }
}
```

The `flagtimeout` is set by the underlying real-time clock. The interpreter loop executes one byte-code and checks if the timer interrupt occurs. If it is so the interpreter will switch to the next task. As the interrupt will be checked only at the completion of a byte-code execution, the byte-code is considered "atomic". There are several time functions provided : `Gettime, Delay` and `After`.

5.1 Access clock

The system clock has clock as a variable which its value is updated every "tick" (the internal time unit). The tick unit has a strong relationship with the frequency of interrupt timer of the underlying system. As the system time is always increasing and all the timers are relative to the system time, the system time is not allowed to change or the inconsistency will occur. Therefore the `gettime` function is provided but no function to set time.

`[gettime]`	store system time to ads	(ads –)

The system time is 32-bit, on a 16-bit system the value will be stored as two consecutive 16-bit words in big-endian format (most significant word first). For a 16-bit system the variable to store time must be declared as an array of 2 words.

5.2 Timer list

The timer list is a list of "timer" which is created by processes to keep their time (for time-out operations and delay operations). Each timer consisted of an owner (pid of a process) and the time count. The timer list is ordered by the amount of the time count of each timer in ascending order. Therefore only the first timer needs to be checked at the regular interval against the system time. Once the system time is greater than the timer the timer is said to be "time-out" and is removed from the timer list.

5.3 Delay

A delayed process sent a timer to the timer list and waited for the timer to time-out and awoke itself.

`[delay ]`	pop a value from stack and send a timer	(t – ...)

5.4 Time-out

There are 3 functions that can caused a process to enter "wait" status : send, receive, wait. A time-out can be programmed for those functions using the `after` clause. The semantics of "wait" for semaphore and message passing is extended to include time-out. The syntax are :

```
wait(... ) after( t ) { body }
send(... ) after ( t ) { body }
receive(... ) after ( t ) { body }
```

The byte-codes for time-out are the cousins of the normal byte-codes : `tmwait, tmsend, tmreceive.`

`[tmwait]`	wait with time-out	(sem, t – ...)
`[tmsend]`	send with time-out	(pid, mess, t – ...)
`[tmreceive]`	receive with time-out	(pid, &mess, t – ...)

For example, a send with time-out will generate I-code :

```
pid mess t [tmsend] [jump #1] time-out body <1>
```

When a process enters "wait" it will send a timer to the timer list. When a process is normally awaken (not because of time-out) its timer will be removed from the list. That process will continue to execute from `jump` instruction which will jump around the time-out code. If a process is awaken because of time-out its entry in the waiting list must be appropriately removed. There are 3 cases :

1. the wait list in Mailbox In-mail,
2. the wait list in Await-mail and
3. the waiting list of a semaphore.

In cases of Mailbox, the message of that process must be appropriately removed from its partner's mailbox. In case of waiting on a semaphore, the entry of that process in the waiting list of that semaphore must be removed. The time-out code will be executed.

5.5 The use of time-out

The time-out can be used to implement non-determinism. Usually, this can be achieved using "selective waiting" which relax the restriction of strict synchronous message passing. Guarded command (Dijkstra, 1975) when used to guard a message operator, is implemented "selective waiting". This was first introduced in CSP (Hoare, 1987). A guarded command is executed only when its guard evaluate to TRUE, for example

```
x < y -> m := x
```

if `x` is less than `y` then assign the value of `x` to `m`. To implement the alternative choice :

```
if x <= y -> m := x
# x >= y -> m := y
fi
```

the `#` denotes the choice. If both choices are possible then an arbitrary choice is made. This construct is non-deterministic. Using time-out we can implement non-determinism. For example, a mailbox process connects to multiple senders, the receiver loop uses time-out to poll the input channels. Assume message is not 0.

```
process sender1() { send(mailbox, m1); }
process sender2() { send(mailbox, m2); }

process mailbox() {
   while( n == 0 ) {
      i = random(2);
      if( i == 0 )       { receive(sender1,&n) after(1); }
      else if ( i == 1 ) { receive(sender2,&n) after(1); }
    }
   ...
}
```

Here the arbitrary choice is made by using a random number 0..1.

5.6 Real-time scheduling

The scheduling algorithm is based on Rate Monotonic theory (Liu and Layland, 1973; Zhao and others, 1985; Lehoczky and others, 1989; Ramamritham, 1990) with a good summary in (Sha, 1988; Sha and Goodenough, 1990). R1 language is designed to be suitable for this method. Some important concepts are introduced here.

Temporal Scopes identify the collection of statements with an associated timing constraint. The possible attributes of a Temporal Scope (TS) are :

1. Deadline
2. Minimum delay
3. Maximum delay
4. Maximum execution time
5. Maximum elapse time

A system is said to be "hard" real-time if it has deadlines that cannot be missed for if they are, the system fails. A system is "soft" if the application tolerant of missed deadlines. A system is "interactive" if it does not have specified deadlines but strives for "adequate response times".

Two types of process are present in the real-time domain : periodic and aperiodic. Periodic processes sample data or execute a control loop and have explicit deadlines that must be met. Aperiodic processes (or sporadic) arise from external asynchronous events. These processes have specified response time associated with them. The process must be analysed to give its worst-case execution time, also may obtained average execution time.

Schedulability
Given a collection of processes and all associated deadlines, determine if this set of processes is schedulable. This means that it is possible for all deadlines to be met indefinitely into the future. In general, necessary and sufficient conditions for schedulability is not known. However, there are many different algorithms presented in the literature which test for schedulability under certain preconditions and restrictions (Leinbaugh, 1980; Sha, 1988).

Method
For each TS, estimate the time that it will take to execute (worst case analysis). Applying the following restrictions :
1. Not allow dynamic memory allocation and dynamic process creation.
2. Place upper bounds on all loops (maximum number of iterations).
3. Giving time-out values for all interprocess communications and external interactions.

Scheduling algorithms
1. Preemptive preference algorithm
2. Scheduling periodic processes , rate monotonic (Liu and Layland, 1973)
3. Aperiodic process (Lehoczky et al, 1987)
4. Earliest dead line
5. Least slack time

A scheduling scheme defines an algorithm for resource sharing and predicts the worst-case behaviour of a program. Most periodic real-time systems are implemented using a cyclic executive which is restrictive. A priority-based scheme gives a more flexible model however interprocess synchronisation can give rise to priority inversion. Some form of priority inheritance (Rajkumar, 1993) must be used. There are several scheduling algorithms suitable for priority-based scheme such as : rate monotonic, deadline monotonic and arbitrary. The R1 scheduler uses a simple round-robin scheme, it has in effect a cyclic executive and is restrictive but very predictable.

6. Implementation

6.1 Compiler

The compiler for R1 language has been constructed and ported to two machines, PC and Sun Sparc. The compiler is written in C for portability. The previous version of this compiler was written in Pascal on a PC and was completely rewritten in C for porting to Unix machines. The size of source program of the compiler is approximately 1800 lines.

It is a two-pass compiler. The first pass processes all global declarations and produced an intermediate file "tmp000.bak". The intermediate file which is used by the second pass, contains only the body of code. All declarations and comments are stripped out. All global names are known by the end of the first pass. The second pass reads the intermediate file, processes the body of code and generates i-code in the memory. At the end of successful compilation, the i-code is post-processed to improve its speed of execution. The buffer is written out with a header containing information necessary for an interpreter to check the validity of the executable code. The header is 16 bytes long and has the following format :

```
putInt(MAGIC);
putInt(VER);
putInt(csize);
putInt(dsize);
putInt(psize);
putInt(0);
putInt(0);
putInt(0);
```

where `putInt` writes 2 bytes binary, hi byte first, `MAGIC = 0x1abc`, `VER = 1` for this release, `csize`, `dsize` and `psize` are the size of code, data and the number of process consecutively.

The source code of compiler contains the following parts :

lexical analyser	300 lines
parser	850 lines
code generator	150 lines
symbol table	250 lines
others (.h etc.)	200 lines
total approx.	**1800 lines**

The compiler on PC is compiled by Turbo C v 2.0 and it is tested to work properly under DOS, MS Windows 3.11, MS Windows 95. The version on Sun Solaris is compiled by GNU C compiler and is tested to work under Solaris 2.3. As this is a simple recursive descent compiler, the compilation speed is reasonably fast. The source code of compiler are arranged as following :

```
compile.c, dcl.c, expr.c, head.c, icode.c, lex.c, stmt.c, symtab.c
```

The compiler accepts input source file (in R1) and produces an I-code file (also produces a readable version of I-code). The command line

```
c> compile test.txt
```

produces `test.out` (I-code file) and `test.lst` (readable I-code).

6.2 Interpreter

The interpreter is written in C for portability (using old Kernigan and Ritchie's style C). It is compiled and tested under Solaris 2.3 using GNU C compiler and on PC using Turbo C v2.0 under DOS and Windows. The source files are arrange as follows:

interpreter	330 lines
process	300 lines
message	230 lines
timer	150 lines
others	80 lines
total approx.	**1000 lines**

and consisted of the following files :

`interp.c, message.c, process.c, timer.c`

The interpreter is implemented entirely in high level language eventhough the real-time facilities required an access to low-level functions. The system time is a 32-bit counter. It is incremented through the interrupt service routine of the underlying operating system. On PC, we use the interrupt vector 0x1c which has the frequency 18.2 Hz. On Unix, we use `signal()` via `setitimer()`. The frequency of interruption can be adjusted in microsecond unit. However, the actual granuality of the resolution of alarm time on Unix is platform-dependent. If we assume the timer ticks at 20 Hz., the 32-bit system time in the interpreter will last approximately 60,000 hours or about 7 years. This should be adequate for most applications.

6.3 Performance of the interpreter

The performance of the interpreter is measured in two categories :

1. How fast it executes a single task process. This measures its execution speed of basic i-code.
2. How fast it executes multi-task process with interprocess communication. This measures the message passing performance.

Benchmark programs

The single task performance is measured using Stanford integer benchmark suite (Hennessy and Nye) which composed of seven small programs : hanoi, permutation, quicksort, bubble sort, sieve, matrix multiplication and 8-queen. The dynamic instruction count are collected and tabulate in Table 6.1. Figure 6.1 shows the frequency of use of each instruction, `Rval` dominated all other instructions, follows by `Lvalg`, `Index` and `Add` .

bubble sort	sort 100 numbers
hanoi	move 5 disks
matmul	multiply matrix 10x10
perm	permute 4 digits of 0,1,2,3,4
qsort	quick sort 100 numbers
queen	find all solution of 8-queen problem
sieve	find all prime number < 1000

The message passing performance is measured with a simple producer-consumer style program as benchmark. The interpreters on PC is compiled with speed optimisation turned on. The PC has the following specification : 486DX2 66 MHz with 24 Mbytes RAM.

Table 6.1 The number of I-code instruction executed

program	**no of inst.**
bubble	110,611
hanoi	1,300
matmul	41,099
perm	6,901
qsort	88,002
queen	752,804
sieve	57,788

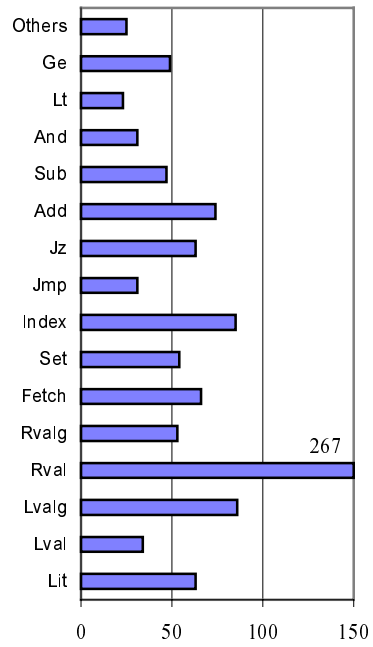

Figure 6.1 the number of each I-code (x1000) executed for Stanford integer benchmark

Table 6.2 Performance of the interpreter on PC

Benchmark	no. of inst.	inst./sec.
Stanford	1,058,520	578,426
Multi-task	2,100,550	38,190

For single task, running Stanford integer benchmark suite, the interpreter achieves 578,426 instruction per second (Table 6.2). For multi-task, the benchmark performs a tight loop of producer-consumer two processes task. There are 100,000 messages sent and received between two processes. This benchmark executes 2,100,550 instructions, has 100,041 task switches, and measured 38,190 instruction per second. The performance of this interpreter compared favourably with an implementation by Ruammahasap (1996) which reported the performance of the earliest prototype of this interpreter as 26,747 instruction per seconds on PC with 486- 33MHz (when adjusted the speed to 66 MHz the number should be x 2 = 53,494 inst./sec.).

6.4 On-line resources

The current implementation is version 1.0. The most uptodate source and bug fixed including the benchmark programs can be found and downloaded from the home page of this project :

http://www.cp.eng.chula.ac.th/faculty/pjw/r1/R1.htm

Acknowledgement

The prototype of this compiler is written by Somsak Ruammahasap as a part of his MSc thesis, Department of computer engineering, Chulalongkorn university, 1996. It has been heavily modified and in many parts rewritten into the current form. Ajarn Chatchawan Wongsiriprasert, Department of Computer Engineering, Chulalongkorn University, has helped implementing the interrupt timer on both PC and Unix system.

7. Conclusion

Our objective to create a programming system that support real-time control application is met with the design and implementation of R1 system. It provides a programming system which is hardware independent by having an abstract layer separate an application program and the underlying control system. R1 language facilitates the writing of multi-task programs. The language provides concurrency control and protection of shared resources including real-time facilities such as clock and time-out. The programming system consisted of a compiler which compile a source language into intermediate codes, called I-code which will be executed on the target machine by an interpreter. The I-code is an architectural neutral byte code aims to be portable and reasonably efficient across many platforms. The interpreter is an abstract machine which execute the byte code, providing the multi-task environment for the target machine.

7.1 Accomplishments

We have accomplished the following tasks :

1. The language R1 is completely specified. Its features included : concurrency, protection of shared resources via semaphores, symmetric communication between processes, real-time facilities.
2. An intermediate code is fully specified. It is a byte code system which emphasizes portability.
3. A compiler that translates a source language, R1, to the intermediate code, is fully implemented for a PC system and a UNIX system. The compiler works under DOS, Windows 3.11 and Windows 95, SUN Solaris 2.3 producing an intermediate code file which can be run on any target machine.
4. An interpreter is implemented using a high level language C. It is compiled and tested on a Sun Sparc architecture to demonstrate the ability to run on a different platform (such as compile on PC run on Sun). It is fully functional, providing multi-task, interprocess communication and real-time facilities.
5. The interpreter has been ported to run on PC using interrupt timer to provide real-time facilities. This interpreter also demonstrates the portability of R1 intermediate code to run on two different platforms: Sun Solaris and DOS (under Windows 95).
6. One experimental system has been constructed to optimise intermediate code sequences for speed. The work is published in : Chongstitvatana, P., “Post processing optimization of byte-code instructions by extension of its virtual machine”, Proc. of 20th Conf. of Electrical Engineering, Bangkok, Thailand, 1997.

The performance of this implementation is measured using benchmark programs. It is found to be acceptable. The portability issue is demonstrated by running the same I-code on two platforms : PC and Unix. We opt not to implement an interpreter for a small embedded controller such as a single board of 8051 or similar because we don't have adequate equipment to develop such a system in short time. The implementation on PC and Unix are also facilitate the measurement and debugging of our system.

7.2 Suggestion for further work

The R1 language is very rudimentary. It has enough features to support our claim and enables us to write example programs but nothing more. To make it really usable the language needed to be expanded. The language should be extended along four categories :

1. Basic functions should be extended to include more operators such as shift, modulo etc.
2. Control flow, at least should have for-loop and switch-case.
3. Structure of data is required similar to `struct` in C (or record in Pascal). This can be done without type.
4. To deal with 3. It is possible to package data type as `Class` therefore R1 made into Object-oriented.

Other obvious further work is to implement an interpreter for a small embedded system. This is the sort of system that benefit greatly from this work because most applications will be real-time control. The speed of execution will be of paramount importance because the resource and the computation power is still rather limited for such systems. The effort should be directed to improve the speed of execution of I-code. Our preliminary work (Chongstitvatana, 1997) show that there are still many possibilities to pursue this line of research.

References

1. Andrews, G.R. and Olsson, R.A. (1986). The evolution of the SR language. Distributed computing, 1(3), 133-49.
2. Bacon, J. (1993). Concurrent systems : an integrated approach to operating systems, database, and distributed systems, Addison-Wesley.
3. Burns, A. and Wellings, A. (1997). Real-time systems and their programming languages (2nd ed.). Addison-Wesley.
4. Chapman (1989). Stack architecture, Ellis Horwood.
5. Chongstitvatana, P. (1997). Post-processing optimization of byte-code instructions by extension of its virtual machine. Proc. of 20th Conf. of Electrical Engineering, Thailand.
6. Dijkstra, E.W. (1968). Cooperating sequential processes. In Programming languages Genuys, F. (ed.) London, Academic press.
7. Dijkstra, E.W. (1975). Guarded commands, nondeterminacy and formal derivation of programs, CACM 18 (8), 453-57.
8. Goldberg, A. and Robson, D. (1989). Smalltalk-80 the language, Addison-Wesley.
9. Gosling, J. and McGilton, H. (1996). Java language Environment Whitepaper, http://java.sun.com/doc/language-environment/.
10. Hoare, C.A.R. (1974). Monitors : an operating system structuring concept, Comm. ACM 17 , 10: 549-557 (October).
11. Hoare, C.A.R. (1978). Communicating sequential processes. CACM 21(8), 666-7.
12. Hoppe, J. (1980). A simple nucleus written in Modula-2 : a case study. Software practice and experience, 10 (9), 697-706.
13. Lampson, B.W. and Redell, D.D. (1980). Experience with processes and monitors in Mesa. Comm. of ACM 23(2), 105-117, Feb.
14. Lehoczky, J.P., Sha, L., Strosnider, J. (1987). Enhancing aperiodic responsiveness in a hard real-time environment. IEEE real-time system symp., Dec.
15. Lehoczky, J.P., Sha, L. and Ding, Y. (1989). The rate monotonic scheduling algorithm – Exact characterization and average case behavior. Proc. IEEE Real-time Systems Symp., pp. 166-171.
16. Leinbaugh, D.W. (1980). Guaranteed response time in a hard real-time environment. IEEE trans. on software engineering, Jan.
17. Liu, C.L. and Layland, J.W. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. Jour. of ACM 20(1), 46-61.
18. Picojava (1996) http://java.sun.com.
19. Rajkumar, R. (1993). Synchronisation in real-time systems : a priority inheritance approach. Kluwer academic press.
20. Ramamritham, K. (1990). Allocation and scheduling of complex periodic tasks. IEEE CH2878-7/90/0000/0108, pp.108-115.
21. Ruammahasap, S. (1997). Development of a concurrent processing language translator, MSc thesis, Department of Computer Engineering, Chulalongkorn University. (in Thai)
22. Sha, L., Rajkumar, R. and Lehoczky, J.P. (1987). Priority inheritance protocols : an approach to real-time synchronization, Tech. Report, Dept. of CS, Carnegie Mellon Univ. (CMU-CS-87-181).
23. Sha, L. (1988). An overview of real-time scheduling algorithms, Software Engineering Institute, Carnegie Mellon Univ.
24. Sha, L. and Goodenough, J.B. (1990). Real-time scheduling theory and Ada. Computer magazine, IEEE, pp.53-62.
25. Tribble, G. (1996). Java computing whitepapers, http://www.sun.com/javacomputing/whpaper/.
26. Whiddett, D. (1987). Concurrent programming for software engineers. Ellis Horwood.
27. Wirth, N. (1977). Modula : a language for modular programming. Software practice and experience, 7(1), 3-84.
28. Zhao, W., Ramamritham, K. and Stankovic, J.A. (1985). Scheduling tasks with resource requirements in hard real-time systems. IEEE trans. on Software Engineering, April.

Appendix A

Formal syntax of R1

`courier` = terminal symbol
item $_{opt}$ = optional

Expression

expression :
  primary
  * expression
  & expression
  - expression
  ! expression
  expression binop expression

primary :
  identifier
  constant
  string
  (expression)
  primary (expression-list $_{opt}$)
  primary [expression]

expression-list :
  expression
  expression , expression-list

lvalue :
  identifier
  primary [expression]
  * expression

binop : one of
  * / + - && || < <= == != >= >

string :
" character-sequence "

Declaration

declaration-list :
  declarator
  declarator , declaration-list

declarator :
  identifier
  declarator [constant]

Statement

compound-statement :
  { statement-list $_{opt}$ }

statement-list :
 statement
 statement statement-list

statement :
 compound-statement
 lvalue = expression ;
 if (expression) statement
 if (expression) statement else statement
 while (expression) statement
 return ;
 return expression ;
 ;

Program

program :
 definition
 definition program

definition :
 data-definition
 function-definition
 process-definition

data-definition :
 global declaration-list ;
 semaphore declaration-list ;

function-definition :
 identifier (parameter-list $_{opt}$) { statement-list }

process-definition :
 process identifier (parameter-list $_{opt}$) { statement-list }

parameter-list :
 identifier
 identifier , parameter-list

Precedence of operators (from highest)

( )	do an expression inside ( ) first
- ! * &	unary op - minus, ! logic not, * deref, & address
* / &&	multiply, divide, logical and
+ - ||	add, subtract, logical or
< <= == != >= >	relational operators

Built-in function and special forms
[ ] $_{opt}$ = optional

```
region(sem) statement
wait(sem) [ after(t) statement ]opt
signal(sem)
send(pname,message) [ after(t) statement ]opt
receive(pname,&message) [ after(t) statement ]opt
print( actualparameter and string constant )
gettime(&t)
delay(t)
```

Appendix B

The I-code

```
[ Literal #ref ]                Lit
[ Lvalue #ref ]                 Lval, Lvalg, Rval, Rvalg
[ Index ]                       Index
[ Fetch ]                       Fetch, Set
[ Jmp #ref ]                    Jmp, Jz, Call
[ Ret0 ]                        Ret0, Ret1, Stop
[ Add ]                         Add,Sub,Mul,Div,LT,LE,EQ,NE,GE,GT,And,Or,Not
[ Send ]                        Send, Receive, tmSend, tmReceive
[ Wait #ref ]                   Wait, Signal, tmWait
[ Print ]                       Print, Printch
[ Delay ]                       Delay, Gettime
[ Func #nformal #nlocal ]       Func
[ Proc #pid #nformal #nlocal ]  Proc
```

Table of I-code encoding

	0	1	2	3	4	5	6	7
0	Lit	Lval	Lvalg	Rval	Rvalg	Fetch	Set	Index
8	Jmp	Jz	Call	Func	Proc	Ret0	Ret1	Stop
16	Add	Sub	Mul	Div	Minus	Not	And	Or
24	Lt	Le	Eq	Ne	Ge	Gt	Print	Printch
32	Send	Receive	Signal	Nop	TmSend	TmRec	TmWait	Delay
40	Gettime							

Operational semantic of I-code

Notation for describing the operational semantic of I-code,

CS[i] code segment at the address i.
DS[i] data segment at the address i.
SS[i] stack segment at the address i.
Both data segment and stack segment resided in the same address space and can be denoted by
M[i] memory contained DS and SS, at the address i.

Some operations take operand(s) from stack and leave a result on the stack. The state of stack can be described by a notation that indicates the values in the stack (before an operation – after an operation) when the top of stack is the left most item, the "..." denotes the items that are of no interested to us.

Push(x) is defined as
Sp = Sp + 1, SS[Sp] = x
x = Pop is defined as
x = SS[Sp], Sp = Sp - 1

Aop (arithmetic op) are Plus, Minus, Mul, Div
Uop (unary op) are Minus and Not
Lop (logic op) are LT, LE, EQ, NE, GE, GT, And, Or

I-code	Operational meaning	Parameters on stack
[Literal #n]	push(n)	(... – n)
[Lvalueg #ref]	push(ref)	(... – ref)
[Lvalue #i]	push(Fp-i)	(... – ads)
[Rvalueg #ref]	push(DS[ref])	(... – value)
[Rvalue #i]	push(SS[Fp-i])	(... – value)

[Fetch]	push(M[pop])	(ads – value)
[Set]	M[pop1] = pop2	(ads, value – ...)
[Index]	push(baseads+ index)	(base, index – ads)
[Jmp #ads]	Ip = ads	
[Jz #ads]	if pop = 0 then Ip = ads	(bool – ...)
[Call #ads]	push(Ip), Ip = ads	(... – return_ads)
[Func #nparam #nlocal]	save state, new stack frame, pass parameters	
[Proc #pid #nparam #nlocal]	new process descriptor, initialise state, awake	
[Ret0]	remove stack frame, restore state	
[Ret1]	remove stack frame, restore state, return a value	(... – result)
[Stop]	terminate the process	
[Aop]	push (pop1 Aop pop2)	(a, b – result as int)
[Lop]	push (pop1 Lop pop2)	(a, b – result as bool)
[Uop]	push (Uop pop)	(a – result)
[Print]	print an integer	(a – ...)
[Printch]	print a character	(c – ...)
[Send]	do send, pid and message are on stack	(pid, mess – ...)
[Receive]	do receive, store a message to mess	(pid, &mess – ...)
[Wait #sem]	do wait operation on sem	
[Signal #sem]	do signal operation on sem	
[tmwait]	do wait with time-out	(sem, t – ...)
[tmsend]	do send with time-out	(pid, mess, t – ...)
[tmreceive]	do receive with time-out	(pid, &mess, t – ...)
[gettime]	store system time to t	(&t –)
[delay]	pop a value from stack and send a timer	(t – ...)

Appendix C

The published work : Chongstitvatana, P., “Post processing optimization of byte-code instructions by extension of its virtual machine”, Proc. of 20th Conf. of Electrical Engineering, Bangkok, Thailand, 1997.

Final Report

A Multi-tasking Environment for Real-time Control

Faculty of Engineering
Chulalongkorn University

Prabhas Chongstitvatana
Department of Computer Engineering

16 November 1998