Language and Operating System Features for Real-time Programming

Marc D. Donner and David H. Jameson
IBM Thomas J. Watson Research Center

ABSTRACT: The adjective *real-time* when applied to a computer system means that the system in question was designed to perform a task that, by its very nature, has firm deadlines that *must* be met for the system to be considered to be performing the task. Real-time does not mean simply *fast* or even *very fast*.

The design features that make a programming language or operating system efficient and effective for information processing are often exactly the features that make it useless for real-time system programming. This paper discusses several of these design mismatches and offers a language and operating system designed specifically for building real-time systems. This language and operating system, named ORE, is being designed and constructed for the immediate purpose of programming our high-performance juggling robot [Donner86]; however it is intended to be useful for a broader range of problems.

Design Mismatches

When an engineer or programmer designs a system, he has one or more applications in mind for it as he refines the design from original rough outline to final complete implementation. At every step of the process there are decisions that must be made: selections of algorithms, choices of techniques and data structures, and definition of features. Good performance for the task at hand is usually the criterion by which candidates are compared.

Most existing programming languages and operating systems are designed with *information processing* tasks in mind. These are tasks in which there is a flow of information from some *input* through some processing out to some *output*. It is rare in this context to see active feedback, that is to say the output of one computation directly affects the input to the same computation at some later time. The primary applications of such information processing systems are economic in nature: for example, banking and other business applications where there is a strong emphasis on guaranteeing the correctness of every calculation so that no economic resource is wasted. The same force also drives economically the designers of these systems to try to make the most efficient use of the computing resource. In recent years, as the cost of computer time has fallen and as the cost of labor has risen, the efficient utilization of human resources, both programmer and user, has shifted the emphasis to modern programming techniques and user interface design. The next four sections will describe four design criteria that drive computational and real-time systems in opposite directions: throughput, privacy, abstraction, and verification. This paper will seek to show that the conventional notions of all four of these ideas are inadequate tools for the guidance of real-time system design and implementation. It will also exhibit the features and some of the implementation details of the ORE language, designed explicitly to address some of the demands of real-time system building.

Throughput Versus Guaranteed Response Time

The goal of efficient usage of the computing resource drives the designers of languages and operating systems to try to maximize throughput. This usually means trying to minimize the number of cycles spent on putatively unnecessary resource management operations. In a timesharing operating system, where the processor resource is multiplexed among some number of competing users, the goal of maximizing throughput results in an effort to make the time slices awarded to individual processes as long as possible, that is do as little resource management as possible per cycle delivered to work on the problem.

In addition to the influence of throughput maximization on operating system design decisions, there is a similar influence on programming language design and implementation. A good example is the compiler optimization known as loop unrolling, in which the compiler reduces the loop closure overhead by concatenating several copies of the loop body and looping around that rather than around a single instance.

The loop unrolling example highlights the design mismatch quite well. One of the most common loops in a real-time system waits for a condition to become true. The loop body, as a consequence, does little more than compute its termination condition. For good response time this needs to be done frequently but more importantly it needs to be done regularly. A thousand tests performed in 100 milliseconds, followed by 900 milliseconds with no tests averages to one test per millisecond, but the worst case response time is 900 milliseconds. Loop unrolling might squeeze 1100 tests into that 100 milliseconds, but it does nothing to ameliorate the 900 millisecond dead time while the test is not being executed at all.

Many architectural features introduced to computers in recent years are designed to capitalize on the statistical properties of programs to improve average throughput. Examples include caching and virtual memory, both of which take advantage of locality of reference and of branching observed in most programs. These techniques make a large slow memory perform *on average* like a much faster memory. The important point, as far as

real-time systems are concerned, is that in the absence of guarantees that a specific piece of code will be in the cache when needed, the only thing that can be *guaranteed* is the performance of the larger, slower memory. In general, optimizations designed to capitalize on the statistics of resource usage are useless in real-time system building. In the worst case they are harmful.

In contrast to maximizing throughput, the goal of the language and operating system for implementing a real-time system is guaranteeing response time. Response time is the time interval between when an event becomes observable by the controlling computer and when the software running actually notices it and acts. In a time-sliced operating system, the response time can be as long as the product of the number of active processes and the maximum timeslice length. This is an unacceptably long time because of the way systems are typically tuned.

Process Privacy Versus Fast Communication

In the earliest days of computing, computers were private machines. You signed up for time, came to the computer room, cleared the machine's memory, and loaded in your program. While you were there you had the entire machine to yourself. This made debugging relatively straightforward, since you never had to worry about any code or data but your own. The designers of operating systems quickly discovered that it would be a good idea to expend effort to provide each user with the illusion that he still had the entire computer to himself. This resulted in the evolution of a variety of schemes to make each user's world private and inviolable. In early machines there were a collection of separately addressable regions and one was assigned to each user. When switching from one user to another, the hardware was told which regions to make addressable and which to make invisible. Swapping systems were built on top of that, allowing the number of addressable regions to be increased beyond the physical memory limits of the machine. Virtual memory and demand paging made it possible to free the users from the addressing limits of earlier systems. Through it all the basic notion has been that each user has a private address space all to

himself. As in the single user/single computer case, the private address space made debugging more tractable, since there was no one else on whom to blame errors.

Recently people have discovered that complex interactive systems are often better expressed as a collection of communicating concurrent processes. The problem is that communication is a violation of the address space privacy notion. After all, what is a message? It is a piece of data that one process wants to send from its address space to another's. The result of policing address space privacy is that communication between address spaces is expensive, typically requiring one or more system calls by each party to the communication for each packet of data transmitted. The consequence is a very large latency for communication between processes in a traditional timesharing operating system. When the communicators are humans, the delay is unnoticeable, but when the communicators are control programs trying to deliver millisecond response times the cost is unacceptable.

Of course there are things that can be done to speed up interprocess communication, such as using the virtual memory system to remap a page of data from one address space to another, hence eliminating at least one copy operation. Nonetheless, there is a clear contradiction between process privacy and communication speed. A designer can always improve one at the cost of the other.

Abstraction Versus Predictable Performance

The notion of hardware architecture, of providing a written specification of what a computer will do and promising to satisfy that specification, has probably done more to make computers an important technology in the modern economy than anything else. By separating the instruction set from the hardware details, manufacturers made it feasible to invest major resources in software development without the fear that the entire effort would be rendered worthless when a new and more powerful computer had to be purchased. Of course, in the modern world it is high-level programming languages that have become the

interchangeable interface, but that doesn't really change the issues. The problem for real-time programming that architecture introduced was abstraction. When trying to estimate the time required to execute a specific piece of code, it helps to have a cleanly defined and simple model of the underlying computational engine. When the instruction set is implemented by microprogramming of some lower level machine, it quickly becomes difficult to model the execution cost of the code. The worst case is illustrated by several recent mainframes in which there is so much microcode that some of it is paged.

Another place where abstraction tools conflict with real-time system design goals is in the provision of constrained types in modern programming languages. Being able to say:

var *foo* : 2..17;

is certainly very elegant and often quite interesting both pedagogically and theoretically, but such constrained types rarely can be checked at compile time. As a consequence, checks must be inserted into the executable code to verify the satisfaction of the constraints at run time. This can be quite expensive. In addition, all that is gained by discovering that a constrained type has gone out of bounds at run time is an opportunity to crash the code.

Ill-chosen abstraction is particularly evident in the design of the Ada runtime system. The interface to the Ada runtime system is so opaque that it is impossible to model or predict its performance, making it effectively useless for real-time systems. Most of the existing implementations are extremely slow and offer no facilities for examining, analyzing, or modifying the performance [Baker87]. It is not the case that abstraction is bad *per se*, but rather that it fails to penalize a host of bad practices that can make the job of the real-time system builder much more difficult. The key point is that abstraction is, as with most other good things, best taken in moderation.

Correctness Versus Expressiveness

The author's experience with programming complex physical systems for real-time performance has shown that decomposing the task into many simple asynchronous threads of control is an extremely effective way to design software for such systems. One bar to the design of languages expressive for this kind of programming is the lack of tools for reasoning about either the correctness or the temporal performance of programs written in such languages. The problem as far as correctness is concerned is that there is nothing that a concurrent language can compute that cannot be computed with a sequential language. If correct computation of some result were the only point of a programming language, then there would be no need of concurrency features. The value of concurrent expression of a program is in the leverage it gives the programmer, in the simplification and the abstraction power.

Without adding any computational power, however, concurrent languages add substantially to the difficulty of reasoning about the correctness of systems. Adding real-time constraints to the problem takes the problem from merely intractable to downright ludicrous. That the problem of reasoning about the temporal performance of programs is intractable can be shown easily by the following argument. If you could analyze a program and predict its temporal performance, let's say its response time for worst case input, you could use the technique to answer the question of whether or not a given program would halt. Since the halting problem has been proven to be undecidable, it is by contradiction impossible to provide a general tool for temporal reasoning about real-time programs.

Modern language design concerns itself with, among other things, the ease of writing verification axioms and proof rules. The theoretical underpinnings for concurrent programming are not yet well developed, but that should not prevent the design, construction, and use of concurrent languages and systems now. Recently there has been a growth in interest in languages expressive for concurrent programming [Stroustrup84 Donner83

BrinchHansen75 Jones86 Swinehart86], so the prospect for the development of tools is good.

Other Traditional Operating System Features

Most of the facilities traditionally provided by a timesharing operating system have to be redesigned for real-time applications. The considerations for these systems including file systems, networking, and a variety of specialized I/O are beyond the scope of this paper. Suffice it to say that for hard real-time problems it is not enough for something to be fast; it must provide guaranteed performance.

Language Features

So far we have argued that design criteria for real-time system programming are such that many of the decisions manifested in existing programming languages and operating environments require reexamination when designing and building tools for constructing real-time systems. In the next section we will examine the design and implementation features of a programming language and runtime system named ORE. ORE is explicitly designed with real-time system programming in mind. It has many features in common with traditional languages, including a type system, tools for defining procedures and functions, assignment of expressions and so on. We shall discuss here the ways in which it differs from traditional languages and motivate each language feature by arguing its need for expressing real-time programs or its superior implementability to comparable facilities in other languages. ORE is derived in many of its details from the OWL language [Donner83 Donner84] that was designed for programming a walking robot.

A sequence in ORE is marked as a list of statements enclosed in angle brackets:

$< stmt_1, stmt_2, \ldots ; stmt_n; >$

Semantically every sequence is an indefinite loop, with termination expressed by explicit execution of the *break* statement:

< ... ; if *B* then break; ... >

The traditional notion of a sequence, which executes once straight through, is expressed with an unconditional break at the end of the sequence:

< ... ; break; >

ORE also has a syntax to identify sequences of code that should be considered indivisible and unpreemptable. This is done by enclosing the statements in braces:

{$stmt_1$; $stmt_2$; $stmt_3$;};

This marking enables the programmer to specify critical sections and be assured that the runtime system will respect that designation.

Concurrency

Several projects have demonstrated that concurrency is useful in the design and implementation of complex software systems. Most of these examples have involved heavy-weight concurrency, systems for which a process switch is a relatively expensive operation. This has resulted in designs with a relatively small quantity of concurrency, with the number of concurrent threads being at most in the tens. Several recent systems have demonstrated the efficacy of finer grained, lighter-weight concurrency [Stroustrup84 Donner83 BrinchHansen75 Jones86 Swinehart86]. This style of concurrency has very different applications to those to which heavy-weight concurrency is suited. With light-weight concurrency it is possible to structure a system with hundreds or more threads executing concurrently, offering the system builder entirely new ways of organizing large programs.

The syntax for concurrent execution in ORE is a list of processes enclosed in square brackets:

[
$process_1$
$process_2$
...
$process_n$
]

The concurrence terminates after all of the compontent threads have been terminated.

Preemption

Techniques for managing threads of control in concurrent systems can be classified into two groups. The first technique is to give each thread a name and require the application programmer to manipulate threads by name. This offers a lot of power, but the costs include:

- unpleasant bookkeeping burdens on the application programmer
- difficult comprehension of the resulting system
- almost impossible debugging
- validation by the runtime system of process names, since they might be corrupted or become invalid while under the control of the application code.

The second technique attempts to manage concurrency with fork/join notions like cobegin-coend in Concurrent Pascal [BrinchHansen75] and path expressions [Campbell73]. These notions are an improvement, but they also reduce the available power unacceptably. The problem with fork/join systems is that there remains a bookkeeping problem if a cluster of threads is to terminate before all of them have agreed to do so. A deadlocked thread can keep the entire fork/join construct from completing.

A solution to this problem is offered by preemption; a mechanism by which the scope structure of the active processes can be exploited to enable one process to control the execution of others, but without any process name bookkeeping required of the user. A primitive form of preemption was provided by OWL [Donner84].

In ORE simple preemption is implemented with the *preempt* statement. For instance:

```
[
    < ... >;
    < ... >;
    < ... ; preempt; ... >;
    < ... >;
]
```

Figure 1. Before preempt.

```
[
    x                          /* dead */
    x                          /* dead */
    < ... >;                   /* executing */
    x                          /* dead */
]
```

Figure 2. After preempt, only one process survives.

When the third process in the example above executes its preempt statement, the runtime system terminates the other three processes and then permits the third process, the preempting one, to proceed. The runtime system arbitrates in the case when preemption is requested by more than one sibling process simultaneously. This is, of course, only significant in a multiprocessor implementation.

Actual experience with the walking robot has demonstrated that this total preemption is sometimes too brutal. In controlling a physical system it is sometimes desirable to stop a collection of threads from executing just long enough to make some minor adjustment or repair to the system and then permit the other threads to go on from where they were halted. ORE has a softer preemption named *freeze* that is useful for this. When a thread executes freeze, the runtime system causes all of the siblings of that thread to be suspended. As soon as all are suspended, the requester is permitted to continue. When the requester has completed its job, it executes the *thaw* statement, telling the runtime system to permit its siblings to resume. This can be viewed as a selective priority enhancement in a dynamic priority system, but without any need for special bookkeeping of priorities.

```
[
    < ... >;
    < ... >;
    < ... ; freeze; ... >;
    < ... >;
]
```

Figure 3. Before freeze.

```
[
    < - >;                          /* suspended */
    < - >;                          /* suspended */
    < ... ; ... ; thaw; ... >;
    < - >;                          /* suspended */
]
```

Figure 4. After freeze and before thaw.

```
[
    < ... >;
    < ... >;
    < ... >;
    < ... >;
]
```

Figure 5. After thaw.

Uses of Preemption

There are several ways in which preemption has proved useful. One is in permitting a control program to be constructed as a simple-minded business-as-usual thread running concurrently with one or more problem handling threads. Another is by letting several threads run concurrently, each producing progressively better estimates of some function, with the computation being terminated by the arrival of the deadline and returning the best result so far completed. A third is a construct we call the *concurrent while*.

Business as Usual

The business-as-usual technique results in code that looks like:

```
[
    < business-as-usual >;
    < when problem_1 do preempt; fixup_1 >;
    ...
    < when problem_n do preempt; fixup_n >;
]
```

A language like Ada with an explicit exception mechanism can achieve some of this kind of structure, but not as easily. This mechanism lets the code that is looking for trouble have its own execution thread before the exception is raised. This relieves the business-as-usual code of the responsibility for detecting and raising exceptional conditions.

A particular intent of providing this mechanism was to make it possible for the author of the fixup code to be able to ignore any potential bad behavior by the business-as-usual code. If this were to be mediated with exclusive locks on the key resources, a problem would arise when the business-as-usual code held the locks, a common occurrence, because it would prevent the fixup code from executing until the business-as-usual code relinquished the locks. This problem has previously been recognized and named the *priority inversion* problem [Sha86] because it can cause a higher priority task to wait for a lower priority task to complete and release its locks. Priority inversion could be resolved by establishing a meta-lock that the preempter would assert when it needed to preempt and which the business-as-usual code would observe, relinquishing all its locks whenever it observed the meta-lock asserted. Such a solution increases the burden on the business-as-usual code, since it must watch the meta-flag at all times and hence cannot use blocking operations. On the other hand, preemption permits the higher priority process, the fixup code, to force the termination of the process holding the locks, thus avoiding the problem completely. Thus, while the semantics of preemption can be achieved with more traditional techniques, it is clear that they are far more complex to program and far less easy to get right.

Concurrent While

The construct known as the *concurrent while* arose from the use of preemption in the construction of a complex robot program [Donner84]. Code of the form:

```
[
    < when done do preempt; break; >;
    < body_1 >;
    ...
    < body_n >;
]
```

appeared in a large variety of places in the program. The attraction of such a construct is that the termination condition is awake continuously, not just at the end of each execution of the loop body. This permits the body to be written without any regard for the termination condition. In normal circumstances code of this type must be written using non-blocking I/O to protect the termination test from deadlocks or large delays. With the concurrent while this problem is bypassed.

Multiple Resolution Computation

Multiple resolution or imprecise computations have recently attracted the attention of the real-time community as a potential way to find design tradeoffs that take advantage of the statistical properties of real-time systems [Lin87]. With lightweight concurrency and preemption, multiple resolution computation looks like this:

```
[
    var best : answer-type;
    < when time-to-stop do preempt; return(best); >;
    < low-resolution; best := result; break; >;
    < medium-resolution; best := result; break; >;
    < better-resolution; best := result; break; >;
    < highest-resolution; best := result;
            time-to-stop := true; break; >;
]
```

This assumes, of course, that each computation takes strictly longer than the lower resolution computations. If that is not so,

then the assignment of *result* to *best* must be protected by some arbitration.

Of course, the code for the multi-resolution computation could make the intermediate values available before the deadline and the design of the system could take into account the availability of lower resolution results. An example of this might be a frequency-agile radio receiver in which the tuning of the narrowband tuner to the vicinity of the target frequency was initiated in response to low resolution estimates provided before the final, ultimate resolution, measurement was complete.

Last Will and Testament

An inadequacy in the early design of preemption facilities was that the responsibility for taking control of all resources relinquished by a terminated process devolved on the preempter. This problem nullified the bookkeeping advantages offered by not having to manipulate process names. The solution to this problem, a new feature in the ORE system, is the provision of a *last will and testament* (LWT) facility. This facility enables a process to establish a strip of code to be executed if the process is preempted and before the process is considered to have terminated. In order to prevent priority inversion, the LWTs have to be executed at high priority. This might be used as follows:

```
[
    < ... ; {start-dangerous-action;
            LWT{stop-dangerous-action};} ... >;
    < ... >
    < ... ; preempt; ... >;
]
```

Notice that starting the dangerous action and establishing the LWT are done indivisibly so that no race condition is permitted in which the action has been started but the LWT has not yet been established. Of course, it will often be possible to establish the LWT before executing the statement that it is intended to undo, but not always. A situation in which the LWT should not be executed unless the preceding action has definitely been initiated we call the *toxic antidote* case. Lest the reader imagine that the

toxic antidote case is imaginary or frivolous, consider the case in which the dangerous action is the launching of a nuclear missile and the stop action is the detonation of range safety charges.[1]

Note that in ORE each LWT established supersedes any previously established LWT. Each sequence has a single LWT associated with it, which may be null. There are limits on the complexity of the code associated with each LWT, about which more will be said in the section on implementation.

Watch and When

The threads of control from which a real-time program is composed usually alternate between waiting and running. Most programming technology is very good at running, but waiting is often a very expensive operation. If the waiting is well understood, then hardware provisions can be made to wake the program up when the wait is done. These hardware provisions are usually cast in the form of interrupts to the processor. If the pause is not well understood, as is typically the case during the development of a system, then it must be done by means of some form of polling or busy waiting. If a large number of threads are waiting, the cost of the regular reevaluation of the conditions can become a significant load on the system.

ORE possesses statements named *watch* and *when* that permit waiting to be done very efficiently. The key observation is that a Boolean expression that may terminate a wait can only change when one of the variables on which it depends changes. Thus the ORE compiler can add side effects to assignment to wake up any threads that are waiting on a change to a specific variable.

Preemption enables a process to terminate or suspend another without knowing the name of that process. Watch and when enable a process to wake up another process without knowing its name.

Watch and when both put a thread to sleep. Wakeup of the sleeping process happens when one of a list of variables named in the watch or referred to in the when receives an assignment.

1. Of course, for security reasons nuclear missiles don't have range safety charges.

Syntactically watch is as follows:

... ; watch(v_1, v_2, ... , v_n); ...

When a process encounters a watch it goes to sleep, removing itself from any eligible-to-execute queue. It is awakened after the first subsequent assignment to any of the watched variables.

... ; when E(v_1, v_2, ... , v_n); ...

A process executing a when goes to sleep, as in the case of the watch. Each time any of the watchable variables named in the expression in the when receives an assignment, the expression is reevaluated and if it has become true the process is awakened.

There is a potential for race conditions with watch and when. For instance, if we have the code:

< ... ; when E ; S; ... >;

and it is possible for the process to go to sleep between the *when* statement and S, it could happen that the action of some concurrent process during that interval invalidates E. In order to provide a way to avoid that race condition, ORE guarantees that when marked with the following syntax:

< ... ; when E do S; ... >;

the statement S will be executed immediately after the condition is asserted, without giving any other processes an opportunity to execute. The statement in question may be an indivisible sequence:

< ... ; when E do {$stmt_1$; ... ; $stmt_n$;}; ... >;

This still does not guarantee that race conditions cannot occur in the handling of the watch and when primitives. If there are two whens active concurrently with conflicting subsequent actions, there is no deterministic guarantee of the results:

```
[
    < ... ; when B do { ... ; B := false;} ... >
    < ... ; when B do { ... ; B := false;} ... >
]
```

This code has the property that one of the two sequences following the whens will start executing with the variable B false. Since the purpose of the semantics of guaranteeing high priority execution to the succeeding statement in a watch or when is to enable the construction and proof of invariants, we can see that there is a relatively easy way to invalidate this guarantee. Techniques to protect against this problem include forbidding multiple processes from waiting on the same condition, something that the *with* directive proposed by Hoare [Hoare85] does, or providing a queue of wakeups for each variable. Rather than handicap all watchable variables with a queue of wakeups, ORE permits a variable to be marked as one that only wakes up one waiting process on each assignment, rather than all of them. This preserves the efficiency of the watch mechanism and makes the implementation of exclusion simple and straightforward.

Watch and when are useful in a variety of situations. A common need in real-time programming is to be able to attach some code to an interrupt. If the deadline requirements for the handling of that interrupt aren't too short, it may suffice to schedule the associated code for execution rather than execute it directly in the interrupt handler. An easy way to achieve this is to have a watchable variable associated with the interrupt and have the interrupt handler perform an assignment to the variable:

```
var TimerInterrupt : watchable integer;
IntHandler(Timer, {TimerInterrupt := TimerInterrupt+1;});
...
< watch(TimerInterrupt); ... >;
```

so that the code after the watch gets scheduled each time the TimerInterrupt is assigned. Of course since ORE wakeups are not queued, if the code associated with the watch is too slow, the next timer tick may be missed. This problem must be resolved by means of verification of real-time deadline guarantees [Mok85].

Another use of when is to enable two processes to synchronize or rendezvous using a watchable boolean as a semaphore:

```
var Flag : watchable boolean := true;
[
    < ... ; Flag := not(Flag); when Flag ; ... >;
    < ... ; Flag := not(Flag); when Flag ; ... >;
]
```

Whichever process reaches the assignment to flag first changes it from true to false and then waits because the value is false. When the other process arrives at the synchronization point it toggles the flag back to true, thus permitting itself and the other process to proceed together. Similarly, multiway synchronization can be achieved using an n-stage semaphore.

Finally, efficient data driven code may be constructed using watch:

```
[
    < ... ; commodity := produce(); ... >;
    < ... ; watch(commodity);
            consume(commodity); ... >;
]
```

Watchability is an attribute of the type of a variable in ORE. Any object or component may be declared to be watchable:

```
var x : watchable integer;
```

Or we may say that an array is watchable:

```
var y : watchable array[0..31] of integer;
```

or we may designate that the array and its elements are independently watchable:

```
var z : watchable array[1..10] of watchable real;
```

So that the user may write the following code:

```
[
            /* any change to the array wakes up here */
    < watch(z); ... >;
            /* assignment to z[1] wakes up here */
    < watch(z[1]); ... >;
    ...
]
```

And similarly with records and pointers. A design question is raised with watchable pointers and arrays. Should the process watching a pointer be awoken when the object pointed to is assigned? The simple answer is no, but how does a piece of code watch a dynamic structure like a tree or a list? We have not yet reached a final decision on which way to resolve this question.

Implementation Considerations

When designing a language for real-time system building, implementation issues must be considered early in the design. The consequence of putting implementation considerations off until the end is a design that may not be efficiently implementable. As a result, the design of the ORE language, compiler, and runtime system have proceeded together, with the semantics of the language being influenced by what was feasible for the runtime system and the design of the runtime system being directed by the needs of the language. This is in contrast to the traditional operating system design philosophy in which the goal is to provide a clean virtual interface, making absolutely no assumptions about the behavior of the application code that will be executed.

The benefits of designing the language, compiler, and runtime system together are a tremendous reduction in the cost of concurrency. Since the compiler constructs the schedulable units, the amount of state that has to be saved at each context switch can be kept to an absolute minimum. Since the application is assumed to be a collection of cooperating threads with no incompetent or antagonistic code, the cost of address space separation and protection may be dispensed with. Since the runtime system's internals are known to the compiler, the compiler can generate code to manipulate directly runtime system data structures rather than going through the costly protections of a system call interface.

Compiler Generated Schedulable Units (strips)

The programmers of the Pluribus multiprocessor [Ornstein75] devised the notion of code strips, segments of code of guaranteed execution time that could be interleaved by a special piece of hardware to achieve high performance multithreaded execution. For Pluribus the strips were constructed by hand and their execution bound was to be proved by hand. The limitation was the fact that since the strips were hand constructed in a low-level language, it was extremely difficult to build a strip and even more difficult to prove its execution bound. The TOMAL compiler [Hennessy77] improved on this situation substantially by enabling the application programmer to insert markings in the code that defined the strip boundaries. OWL [Donner83] took this technique one step further with a compiler that constructed the strips automatically, with the strip boundaries inferred from an analysis of the code.

The ORE compiler generates code strips and the ORE run-time system multiplexes them for execution. Some of the advantages of this approach include:

- Minimal state saving required at process switch. This results from the fact that no partially evaluated expressions or other state information will be found in registers at context switch time, thus reducing the amount of information to be saved in a process to a single pointer to the heap-allocated block of memory from which it is executing. In addition, since all the threads execute in a single address space, no virtual memory management is associated with context switching.
- Compiler derived performance information. Since no strip may exhibit unbounded looping or recursion, the worst case execution time for a strip can be calculated at compiler or link time. This information is valuable for scheduling algorithms that seek to provide deadline guarantees.

In addition, ORE's explicit strip notation {...} permits the application writer to designate critical sections and guarantee that they will be treated as such. The compiler, since it knows about the schedulable units, is able also to reject strips designated by the

programmer that are illegal, for instance because they contain blocking operations.

A strip generated by the ORE compiler consists of statements from a single thread of execution. Strip boundaries come either when a blocking operation is encountered or when the worst case execution time of the strip exceeds a given limit. Blocking operations include I/O and the ORE parallelism and waiting constructs.

Runtime Scheduler

On a uniprocessor the illusion of concurrency is achieved by interleaving strips from the live processes in some order. The program that determines the order of execution of processes is known as the scheduler. On a multiprocessor it is often convenient to have a scheduler on each processor so that the number of processes is not limited by the number of processors. ORE is designed to run on a heterogeneous multiprocessor, with an instance of the scheduler on each machine.

The default scheduler in the ORE runtime environment implements a simple round-robin policy. In a round-robin policy, each live process gets a chance to execute a strip once before any process gets to run again. This policy is simple and is known not to be optimal for most applications, however it is sufficient for a variety of applications.

Extensive research has been conducted into scheduling policies, the results of which are beyond the scope of this paper. The seminal paper by Liu and Layland in 1973 [Liu73] demonstrated the optimality of a policy called *rate-monotonic* under certain assumptions. Subsequent research, of which a good summary is presented in a paper by Ramamritham and Stankovic [Ramamritham84], has developed and analyzed a variety of other schedulers, many of which are optimal for certain evaluation criteria.

In an the absence of a scheduler that is optimal in all circumstances, the best strategy is to provide to the application programmer as much flexibility as possible. In ORE that is achieved by permitting the interception of the scheduler by the application. The programmer does this by defining one data structure, the UserArea, and writing four routines: InitUserArea,

FreeUserArea, GetNextPid, and PutAwayPid. Basically, the activation record describing an active process contains a user word that may be used as data or may be used as a pointer. The InitUserArea routine is called by the CreateProcess code in the operating system, at which time a user area data object may be allocated, initialized, and linked to the activation record. If the needs of the system are satisfied with a small amount of data, the user area word in the activation record may be used directly as data instead of as a pointer to a larger area. The FreeUserArea routine is called by the KillProcess routine to return the data structure to its free list or storage pool.

The scheduler has the following structure:

```
var ActivationRecord : ^p;
while true do
    p = GetNextPid();
    RunStrip(p);
    PutAwayPid(p);
    end;
```

Where the code for GetNextPid examines the tree of activation records and user areas and determines which process should run next. RunStrip is a kernel routine that executes the strip associated with the process p and performs any cleanup operations associated with termination or preemption that may have occurred during the execution of the strip.

The code for GetNextPid and PutAwayPid is assumed to be written using kernel primitives that permit read access to the process data structures. This includes things like Sibling(pid), Parent(pid), Child(pid), and so on. It is assumed that the UserArea structure will be used to hold such items as deadlines, priorities, elapsed times, and any other data that the application programmer deems to be of use in the computations in GetNextPid and PutAwayPid. The application programmer will presumably also write functions to provide the ORE code with access to the UserArea data structure. These functions might include SetPriority, ReadPriority, SetDeadline, IncreasePriority, or what have you.

Watch and When

The semantics of watch and when are relatively simple to describe. The implementation, however, requires considerable attention to detail if it is to perform well. There are two places in a program that the compiler needs to insert watch and when code:

- the watch or when site
- the assignment of each watchable variable

In the rest of this discussion we will concentrate on the implementation of watch. When is very similar, with the addition of some code to evaluate the expression and put the process back to sleep if the wakeup condition is not satisfied, so we will omit it here.

The data structure that points at a PID must have a pair of pointers to link it into the list of watchers on a variable, a pointer to the PID in question, and a fourth component, *incarn*, whose use will be described later.

```
type PidItem = record of
        up      : ^PidItem;
        down    : ^PidItem;
        incarn  : integer;
        pid     : ^ActivationRecord;
end;
```

Figure 6. Type definition of the piditem data structure.

Each process instance will have on its stack the object

var *PidItemArray* : array[1..*WatchedByProc*] of *PidItem*;

which has a PidItem for each distinct watchable variable ever watched by that process.

In addition to this structure, there are two compile-time data structures available:

v[i] a mapping from indices of the PidItem array to the corresponding watchable variables

list(w) a list of all the indices of items watched by watch statement w.

Now we can exhibit the code inserted at each watch statement w:

```
foreach i in list(w) do
            /* it is not linked in, do so now */
    if PidItemArray[i].up = null then
        InsertInVariableQueue(v[i], PidItemArray[i]);

            /* We are in watch number w now */
    PidItemArray[i].incarn = w;
    MyActivationRecord.incarn = w;
    RemoveFromReadyQueue(MyActivationRecord);
    end
```

Figure 7. Code to establish a watch.

Notice that it is possible that the PidItem for that variable may still be linked in to the list associated with its variable, so that there may be no need to link it in. That is the reason for the test of the *up* against NULL before linking in the item.

The following example illustrates the kind of situation that might result in an item remaining linked in:

< watch(*a, b*); ... ; watch(*b, c*); ... >;

If the first watch is terminated by assignment to the variable a, then the PidItem hooked to b will remain unless it is removed. Rather than do the work of removing items from lists, we leave linked but unassigned items in place. If the b variable should be assigned after the expiration of the first watch in the example above, we would detect it by comparing the incarnation number in the activation record with that in the PidItem. If they match, then the wakeup is valid, if not then the assignment is to a dead watch and may be discarded.

In view of the preceding, the code for assignment is:

```
                    /* Do the actual assignment */
object^.thing := Expression;
p := object^.pid;
while p ≠ null do
                    /* is it still valid? */
    if p^.incarn = p^.pid^.incarn
                    /* yes, perform the wakeup */
        then do
            MoveToReadyQueue(p^.pid);
            p^.pid^.incarn := AWAKE;
            end;
    q := p;
    p := p^.next;
                    /* prune it out of the list */
    q^.up := q^.down := null;
    end;
```

If the cost of moving the process to the ready queue from the sleep queue is prohibitive, we can simply mark the awakened processes and process them from the scheduler.

The amount of work done by the watch establishment code is limited to examining each PidItem, relinking those that have become unlinked by a previous assignment, and updating the incarnation number for each PidItem and the activation record of the watching process. The work done by the assignment is proportional only to the number of processes watching that variable at the time of the assignment.

Compiling for a Real Multiprocessor – Export/Import and Expose/See

Most non-trivial programs are partitioned into several modules. Each module implements some particular aspect of the system and contains its own procedures and variables for that purpose. Ultimately, all the modules are linked to form a complete program. If compile-time type checking is available then the compiler needs type information of objects referenced in one module but defined in another. This can be done by separating

the definition and implementation parts of a module and making the definition part available during compilation of some other module depending on that module. In Modula-2, for example, definition modules define and export objects. These exported objects may then be imported into other modules.

Now suppose that we wish to reference an object that is located on a specific machine, perhaps because it is associated with some physical device attached to that machine. The code in question may exist in a library or in a machine module, but we need to be able to have that code run on our behalf on the machine where it makes sense for it to execute. We would also like to retain the compile-time type checking mechanism. In ORE, this is done with *see* and *expose*, which are to interprocess access what import and export are to code.

For each processor there exists a main module called the *machine module*. This is the module where execution starts. This module can import code from library modules using an import mechanism similar to that of Modula. At link time, imported code is physically linked with the executable machine module. In addition, a machine module can *see* objects on another machine. The other machine must *expose* the object. Analogous to a definition module for library code, a machine module wishing to expose objects must have a corresponding *interface module*.

For example, suppose machine M has a procedure named P that machine N would like to run as a process. The code for machine N is

```
machine N;
see p on M;
process main;
    [                       /* start 3 processes */
        < ... >;            /* local process */
        M.p(arguments);     /* runs on machine M */
        < ... >;            /* local process */
    ];
```

Since P is exposed, M must have an interface part as well as a machine part.

```
interface M;
expose p;                          /* p is allowed to be seen */
process p(a, b, c);                /*by another machine */
```

while in the machine module ...

```
machine M;
process p;
<
    /* code to implement p here */
>
```

A machine module can also expose an object that comes from a library module. In this case, the interface module first imports the object and then exposes it.

Reasoning about Temporal Properties of Programs

At the outset of this paper we mentioned the fact that real-time systems had hard deadlines that had to be satisfied. For a system to truly satisfy real-time requirements, then, it must be able to provide performance guarantees. High performance is not sufficient. Satisfying the deadline most of the time is rarely satisfactory.

The astute reader will have noticed that the ORE language does not possess any features for describing or reasoning about the temporal properties of programs. There has been substantial work in the area of reasoning about the temporal properties of programs [Mok85] and of attempting to guarantee the performance of programs run under a variety of scheduling disciplines [Liu73 Ramamritham84]. ORE is intended to be a practical testbed for these techniques. The mechanism for replacement of the scheduler with code written by the user is intended to support experimentation with scheduling techniques. ORE will eventually also have a notation for marking the code with events and expressing assertions about the temporal constraints on those events.

References

T. P. Baker, Implementing Timing Guarantees in Ada, *Fourth Workshop on Real-time Operating Systems*, pages 129-133, IEEE, July 1987.

P. Brinch-Hansen, The Programming Language Concurrent Pascal, *IEEE Transactions on Software Engineering*, 1(2), June 1975.

R. H. Campbell and A. N. Habermann, The Specification of Process Synchronization by Path Expressions, *Lecture Notes in Computer Science* Number 16, pages 89-102, 1973.

M. D. Donner, The Design of OWL: a language for walking, *Proceedings of the Sigplan '83 Symposium on Programming Language issues*, pages 158-165, ACM, June 1983.

M. D. Donner, *Control of Walking: local control and real-time systems*, PhD dissertation, Carnegie-Mellon University, 1984.

M. D. Donner, *Real-time Control of Walking*, Birkhauser, Boston MA, 1987.

M. D. Donner and D. H. Jameson, A Real-time Juggling Robot, *Proceedings of the Real-Time Systems Symposium*, pages 249-256, December 1986.

M. D. Donner, Language and operating system integration for real-time systems, *Fourth Workshop on Real-time Operating Systems*, pages 106-109, IEEE, July 1987.

J. L. Hennessy, *A Real-Time Language for Small Processors: Design, Definition, and Implementation*, PhD dissertation, SUNY Stony Brook, August 1977.

C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs NJ, 1985.

M. B. Jones and R. F. Rashid, Mach and Matchmaker: Kernel and Language Support for Object-oriented Distributed Systems, *Object-oriented Programming Systems, Languages and Applications*, ACM, 1986.

K. J. Lin, S. Natarajan, J. W. Liu, and T. Krauskopf, Concord: A System of Imprecise Computations, Submitted for publication, 1987.

C. L. Liu and J. W. Layland, Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, *Journal of the ACM*, (20)1:46-61, January 1973.

A. K. Mok, SARTOR – a Design Environment for Real-Time Systems, *Proceedings Compsac*, October 1985.

S. M. Ornstein, W. R. Crowther, R. D. Kraley, A. M. Bressler, and F. E. Heart, Pluribus - A reliable multiprocessor, *AFIPS 1975 Conference Proceedings*, pages 551-559, AFIPS, 1975.

K. Ramamritham and J. A. Stankovic, Dynamic Task Scheduling in Hard Real-Time Distributed Systems, *IEEE Software*, pages 65-75, July 1984.

L. Sha, R. Rajkumar, and J. P. Lehoczky, The Priority Inheritance Protocol – An Approach for Real-Time Synchronization, Submitted for publication, 1986.

B. Stroustrup, The C++ Programming Language – Reference Manual, AT&T Bell Laboratories Computing Science Technical Report 108, 1984.

D. C. Swinehart, P. T. Zellweger, R. J. Beach, and R. B. Hagmann, A Structural View of the Cedar Programming Environment, *Transactions on Programming Languages and Systems*, (8)4:419-490, October 1986.

[*submitted Aug. 9, 1987; revised Oct. 20, 1987; accepted Oct. 23, 1987*]