A Language Support Environment for Complex Distributed Real-Time Applications

Alexander D. Stoyen*, Thomas J. Marlowe†, Mohamed Younis‡ and Plamen Petrov*

* *Real-Time Computing Laboratory,*
Department of Computer and Information Science
New Jersey Institute of Technology, Newark, NJ 07102, USA
alex | plamen@rtlab12.njit.edu

† *Department of Mathematics and Computer Science,*
Seton Hall University, South Orange, NJ 07079, USA
marlowe@inis.njit.edu

‡ *AlliedSignal Aerospace, Microelectronics and Technology Center,*
9140 Old Annapolis Road/MD 108,
Columbia, MD 21045, USA.
younis@batc.allied.com *

Abstract

Engineering of complex distributed real-time applications is one of the hardest tasks faced by the software profession today. All aspects of the process, from design to implementation, are made more difficult by the interaction of behavioral and platform constraints. Providing tools for this task is likewise not without major challenges. In this paper, we discuss a tool suite at New Jersey Institute of Technology's Real-Time Computing Lab which supports the development of complex distributed real-time applications in a suitable high-level language (CRL). The suite's component tools include a compiler, a transformer-optimizer, an allocator-migrator, schedulability analyzers, a debugger-monitor, a kernel, and a (simulated) network manager. The overall engineering approach supported by the suite is to provide as simple and natural an integrated development paradigm as possible. The suite tools address complexity due to distribution, scheduling, allocation and other sources in an integrated manner (largely) transparent to the developer. To reflect the needs of propagation of functional and non-functional requirements throughout the development process, a number of robust code transformation and communication mechanisms have been incorporated into the suite. To facilitate practical use of the suite, the developed programs compile-transform to a safe subset of C++ with appropriate libraries and runtime support.

1 Introduction

Two exciting areas have emerged in recent years from, in part, the real-time community. One, *compiler support for real-time systems*, is studying and modifying well-understood compiler technology in the context of real-time and complex computer systems. The other, arising from the intersection between real-time systems, software engineering, and fault-tolerance and reliability, is the newly-recognized field of *complex computer systems*, dealing with large applications with complex dependencies running on distributed heterogeneous platforms. These interact, and also affect and are affected by the choice of real-time language, by aspects of the run-time system such as scheduling, by specification methodology, and so on.

1.1 Motivation and related work

The motivation for developing real-time programming languages is to facilitate writing of correct and maintainable real-time programs, through more effective use of abstraction, compilation, and *a priori* analysis. Among the many computer languages, design has not only considered environment (e.g., batch or interactive) and paradigm (e.g., imperative

*This work is supported in part by the U.S. ONR Grants N00014-92-J-1367 and N00014-93-1-1047, by the U.S. NSF Grant CCR-9402827, by the U.S. NSWC Grants N60921-94-C-0106, N60921-94-M-1250, N60921-93-M-3095 and N60921-93-M-1912, and by the AT&T UEDP Grant 91-134.

or object-oriented), but also intended application domain. Thus, FORTRAN was designed specifically for scientific computing and COBOL for business applications; real-time computing can similarly benefit from special-purpose languages. While several languages have been designed or designated to be used in real-time computing, until recently they have lacked, to a significant extent or entirely, the notion of real time as a first-class entity.

Schedulability analysis — a key requirement for a high-level real-time language — was originally defined by Stoyen (formerly, Stoyenko) [19, 38, 39, 40, 43], and refers to any pre-execution or symbolic language-level analysis of programs that either determines whether programs will meet their critical timing constraints when executed, or derives programs' timing characteristics. While the theory and implementation of schedulability analyzers are beyond the scope of this paper (see for example [19, 22, 24, 29, 30, 31, 43]), it is important to realize that such analyses and tools cannot be applied to conventional higher-level languages, sequential or concurrent, for two largely independent reasons. First, such languages lack the syntactic and semantic support to define *real-time* processes and constraints. Second, constructs such as while loops, recursion, access to dynamic memory, and some forms of interprocess communication and synchronization may have unbounded (and unpredictable) worst-case execution times. Thus, conventional languages need to be either modified or extended with additional higher-level directives or "pragmas" to facilitate their use in real-time computing.

In addition, schedulability analysis requires predictable system behavior from both hardware and software. Ideally, the time taken for execution of each machine instruction should be known, with no unpredictable delays from hardware or system calls. In practice, timable units are more typically basic blocks, so that variations in execution times of individual instructions (and effects of instruction scheduling and simple pipelining) tend to average out and can be accounted for [15]. In many cases, time-critical real-time applications are being implemented on such predictable platforms, assembled from existing components (see for example [13, 14]). Much of the effort lies in making instruction execution, memory access, and I/O externally predictable.

1.2 The real-time language scene

Higher-level real-time languages have been proposed beginning with the Plankalkül of Zuse in 1946 [53], through the creation and standardization of CCITT and Pearl in the 1970's, to modern approaches using Ada [50, 1, 4, 34] and Real-Time Java [26]. All have, however, significant shortcomings. The early languages naturally do not include now standard high-level language constructions, and the modern languages have their own problems.

The approach we have chosen to follow in CRL [47] had its origins in Real-Time Euclid [19, 40], which was the first language to include a complete schedulability analyzer independent of scheduling methodology, and to explicitly refuse to compile programs which contained segments (other than the outermost driver loop) whose execution time was unbounded, or otherwise failed to meet their timing constraints. Related languages include Tomal [18], Flex [23], RTC++ [16], and High-Integrity PEARL [42].

A second motivation and area of related work for CRL is recent work on compiler and environment support for program development, and for tools such as compilers or debuggers (such as [7, 8, 9, 27, 44]) and to refine static timing computation ([5, 24, 28]).

One of the common themes is the perceived need for a constraint granularity finer than entire process frames. A common suggestion is to allow timing constraints on loop bodies or called functions; another is to allow constraints between more-or-less arbitrary pairs of observable events. Our own approach accords with that independently proposed by [7], which allows these, but also the beginnings and ends of arbitrary statements, to serve as anchors for absolute or relative constraints.

1.3 Our contribution: an integrated development environment for complex real-time systems

In this paper, we describe an integrated development environment — under construction at the Real-Time Computing Lab at NJIT — for complex real-time systems. Currently, the prototype environment is based on CRL. The integrated environment will eventually make heavy use of a full range of compiler analyses and tools, and will include among its components:

- A tool for assigning modules of complex computer systems to processors in an arbitrary configuration, and for statically evaluating the resulting assignment.
- A reasonable language for programming real-time or complex applications, providing for flexibility of program idiom and constraint expression while remaining as safe for complex real-time applications as Real-Time Euclid [19] is for the applications for which it was designed.
- Compiler analyses and other support for schedulability analysis, verification of constraints (including constraint propagation and consistency checking), elision of run-time checks on assertions, and extraction of hints for the optimizer (e.g., where timing constraints are violated) and the kernel scheduler (e.g., where timing constraints or resource constraints are tight), and the rest of the run-time system (e.g., bounds on message sizes), plus the schedulability analyzer itself.
- A tool, automatic or semi-automatic, for determining the safety of standard compiler transformations for optimization, parallelization, and speculative execution, in the presence of timing and other constraints, and for applying those transformations if their safety and profitability can be demonstrated.

- Compiler and environment tools for off-the-clock profiling, monitoring, debugging, testing and evaluating complex computer systems, both in the development phase, and for software maintenance. These would include, for instance, a logger and various data-flow tools for displaying def-use chains, call graphs, interfaces, and so on.

- A workload generator, a simulator, and a testbed for determining the net profitability in applying certain types of heuristics, transformations, and/or policies to applications with synthesized realistic workloads, constraints, and dependencies. These tools will also permit testing of partially-developed systems.

- A run-time environment, including kernel support for communication and dynamic scheduling, both for symbolic execution/simulation and for actual execution.

- Interfaces and hooks for other tools, including on-the-clock observers and transformations to support fault tolerance and other requirements, and for the analyses needed to support those tools.

- A user interface and help system to aid in development.

This integrated environment aims to provide comprehensive tools for all phases of the development and deployment of complex real-time systems. We will further describe each component in the following sections.

2 Overview of the platform components

The platform consists of seven major components, as shown in the Figure 1.[1] A more detailed discussion follows in the balance of the paper.

The input for the prototype may include, in addition to source code, an architecture file describing the target processors' architecture, an instruction time map for that architecture, and a global assertions file providing user annotations to be used by the transformation engine, in, for example, performing partial evaluation. (There may be other, local assertions embedded in the source.)

The first component is the *compiler* for the CRL language. The front end of the compiler generates intermediate code (in this implementation, a safe subset of C++), including run-time checks, and extracts constraints, assertions and other directives, and creates files containing this information, for use by static analyzers and by the run-time environment.

The compiler also generates a set of representations: a call (multi-)graph capturing caller and callee relationships and bindings, and a data dependence graph and control flow graph for each process and each

[1]The assignment tool is currently neither CRL-based nor integrated, and is thus not discussed subsequently.

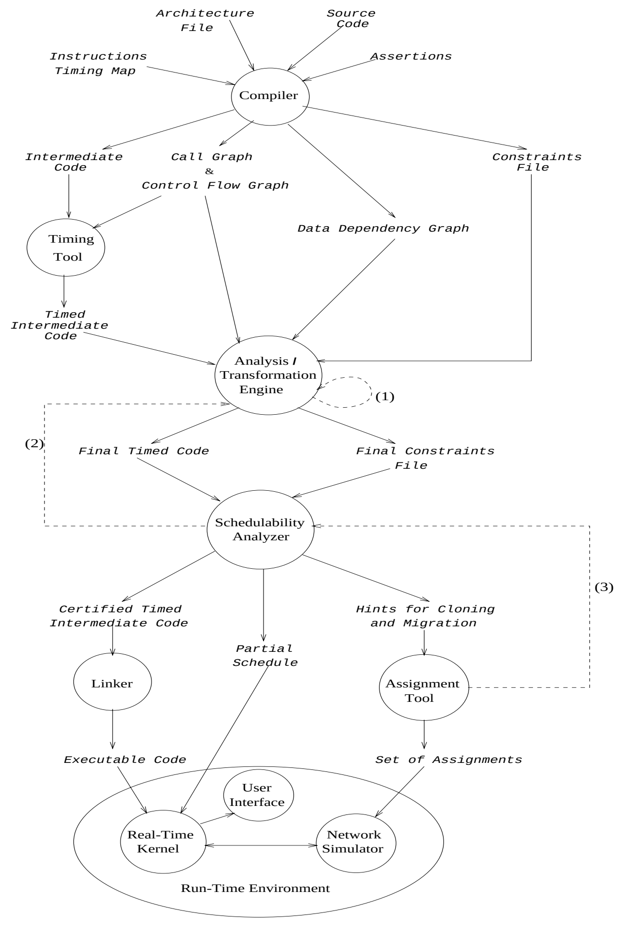

Figure 1: The platform software components

method, to be used both by the timing tool and by the transformation engine. Currently we are not generating any low-level or machine code, but relying on the intermediate code in our analysis. The timing tool then uses the instruction timing map plus the architecture file to assign times to atomic statements (but not structured statements or calls) of the intermediate code.

The *analysis/transformation* engine uses the timed intermediate code generated by the timing tool (implemented by defining a time variable incremented while traversing statements of every basic block by the execution time of statements of that block) and applies static analysis and various transformations, as discussed in earlier sections, to improve the code. Moreover, it tries to eliminate some checks, and to detect certain classes of errors, resulting in a final version of the code and of the constraint file.

The *schedulability analyzer* then takes the transformed code and constraint file, and certifies schedulability modulo the validity of constraints and assertions. The schedulability analyzer also reports a possible object-to-processor assignment (in which some objects may be replicated on every processor), and a partial or complete static scheduler.

The *run-time preprocessor* (linker) translates the intermediate code into executable code. The *run-time kernel* uses the executable code and the final constraint file and consults the static schedule generated by the schedulability analyzer to schedule tasks, allocate resources, and manage object queues. The *network simulator* provides the kernel with the delays due to communications (transmissions and message queuing). Finally, the *user interface* component displays some measurements, such as performance, processes missing deadlines, and average case improvement.

3 The compilation process

Inputs to the compilation process include (1) the source code, (2) a file containing descriptions of architectural components, describing processors, links, devices, etc. (for now, we assume a homogeneous network with an arbitrary topology), including instruction-class/time maps, network topology, and other interconnection details, and (3) a (possibly empty) file of global compile-time assertions for the partial evaluator. The output from the compiler will be an intermediate code program (in C++) and a timing constraints file. In addition, the compiler will construct the graphical representations described above: a *call graph* (caller and callee relationship and bindings), and for each process and method, a *data dependence representation* used by the timing tool, and a *control flow graph*, to be used by the analysis/transformation engine, as illustrated in *Section* 5. Currently, the compiler generates use-def and def-use chains [2] as a data dependence representation, with monolithic handling of arrays, that is, a reference to one entry of an array is considered as using the whole array. (However, since records in CRL cannot contain access/pointer fields, each field accessor is represented by a separate variable.) The intermediate code is then subject to transformations by the analysis/transformation engine after timing analysis. Correspondence between generated code and control flow graph is currently maintained by two pointers per basic block, to the starting and ending line numbers of the translated block, respectively, which is sufficient for our current transformation set.

Some restrictions have been imposed to facilitate the compilation process. As in Pascal, use of, or reference to, any variable or object must be preceded by an explicit declaration of that variable or object. All parameters of objects, methods and threads are passed by value, result, or value-result (with the compiler free to optimize if appropriate); in addition, object parameters must be explicitly specified as imported or exported (or neither). The compiler will match any call to a method or a thread against the interface of that method or thread. The language provides only static scoping and at present disallows all aliasing. This places severe restrictions on the use of array index expressions. These restrictions, both on array index expressions and aliasing between array elements, will probably be relaxed in the future (particularly as arrays are treated as monolithic in our analyses).

Currently, we do not assume any target architecture for the compilation process. Given such a platform, the transformed C++ code will be further compiled and linked with other library routines, and the kernel will be responsible for invoking the generated executable code.

4 The timing tool

The timing tool is used to provide a safe static estimate of the execution time of programs. Inputs to the tool include the timing map of instructions executed by the target architecture, given as a table of (instruction type, processor type, required execution time) triples, As we currently assume a homogeneous platform, the timing map is currently a set of (instruction type, required execution time) pairs. We also do not currently model the effects of architectural features such as addressing modes, memory hierarchies, or pipelining. We hope to consider these in our future work.

To resolve the execution time of calls, the timing tool computes for each method the total time for instructions and calls, propagating backward from leaf methods in the call graph, which is unwound if necessary in the presence of (bounded) recursion. For remote calls, the tool must consider communication delays that messages may anticipate due to contention. We assume an upper bound on the propagation of messages throughout the network; delays can be left as parameters in timing expressions, or this upper bound substituted.

The timing tool calculates two types of execution times: first, the worst-case execution time for each process, to resolve references to other methods through calls; and second, a timing annotation on every executable statement, both simple and structured, to aid in transformation, using the timing map.

Currently, the execution time of each basic block is computed by the timing tool and stored in the process or method control flow graph, to be used in justifying safety and profitability of transformations, which then update the timing information. The timing tool also adds a statement to the output intermediate code at the end of each basic block, which adds the block execution time to a local time variable; the value of this variable is used to propagate timing information to the run-time environment, as discussed in *Section* 8. The worst-case execution time of the whole method/thread is then deduced using the execution time of basic blocks and the time it spends in calls; this is then stored with the method/thread entry in the call graph.

As we are generating intermediate rather than target code in the current implementation, we use a map for the basic data types (classes) defined by the language. The execution time of instructions (methods in basic classes) is based on assumed properties on the architecture and operating system. Compound statements and calls are annotated by the compiler with their initialization time and any other constant execution time, exclusive of the cost of the statements in the body; for example, for a loop, the timing costs include initialization time and time for a worst-case number of instances of jumping to the header and evaluating the loop condition. The time for the entire composite statement can be derived by time-attribute combining rules for each type of structured statement.

The output of the timing tool is a timed intermediate code. The transformation engine then uses that output and the timing constraints file generated by the compiler to check the feasibility, safety and profitability of transformations, as elaborated in the next section.

5 The analysis and transformation engine

As mentioned above, the transformation engine uses the data dependence graph, the call graph, and the control flow graph generated by the compiler to detect various possible code transformations [45, 46, 51]. The timing constraints file is consulted to test the safety of proposed transformations; the timing profile generated by the timing tool is used to measure profitability.

Currently the tool supports a limited number of transformations. Ultimately, the tool will support a much larger number of transformation rules aimed at improving code and/or facilitating analyses while not worsening the timing of the code. The engine applies the transformations as a sequence of steps. In each step, a different transformation or kind of transformation is considered. The order in which the transformations are to be applied remains an issue to be addressed in future experiments. It may also be useful to repeat some steps because of successful transformations in other steps. For example, we can re-apply branch/clause transformations if a condition is eliminated by the conditional linker. This dependence is represented by the feedback arrow (1) in *Figure* 1.

The analysis/transformation component has two effects: first, it changes the code according to the rules of the transformations applied; second, it may relax some constraints or strengthen assertions, most likely through interaction with the developer/user. Some transformations, such as branch/clause transformations, change only the timing analysis and/or final code, without affecting timing constraints, while others, such as dead-code elimination, may affect both code and constraints.

Apart from per process transformations, we have developed an approach to collect, manipulate or delete delays across processes (see [52]), either for process optimization, or to provide "free time" for monitoring and debugging, context switch, or checkpointing for fault tolerance. The tool will eventually consider non-functional criteria beyond timing, such as fault-tolerance or security.

The output from this tool is updated timed intermediate code, as well as an updated timing constraints file. These outputs are then used by the schedulability analyzer, as illustrated in the following section. The schedulability analyzer may need to call the transformation engine if it is not able to guarantee schedulability (branch (2) in *Figure* 1), as clarified in the next section.

6 The schedulability analyzer

The schedulability forms of the code (produced by the analysis and transformation engine) and of the constraint file are passed to a schedulability analyzer, which may use either an exhaustive or a heuristic analysis to produce an assignment and a certificate of schedulability. The analyzer may also report a partial static schedule to be used by the run-time environment. In addition, it may generate directives for migration and cloning to the assignment tool.

The schedulability analyzer may also consult the assignment tool for the feasibility and profitability of certain transformations, as in the case of parallelization and speculative execution (feedback (3) in *Figure* 1). If some of these are either infeasible or unprofitable, the schedulability analyzer will report this fact (feedback (2) in *Figure* 1) to the transformation engine, requiring it to undo the transformation. Moreover, if the analyzer cannot find a feasible schedule, it may request more effort to be spent on analyses and transformations, either by focused optimization, or in the sense of [8], to enhance the schedulability of the code.

Finally, the schedulability analyzer generates certified intermediate code from which the compiler back-end will generate executable code. In the current implementation, we use the C++ compiler and linker, as discussed in the following section.

7 The linker

As mentioned, we are not considering a specific architecture at the moment. The target code machine implementation is a mixture of native C++ statements for some control statement support, and a set of C++ class objects, types, and resources and for the kernel interface. Thus, the linker is simply the C++ compiler, which compiles the certified intermediate code generated by the schedulability analyzer, and links that code with kernel code, as well as with the basic C++ classes.

The executable code generated in this stage is executed by the run-time environment, which simulates distributed processing of the code over a network of processors.

8 The run-time environment

The run-time environment consists of a kernel, a network simulator and a user interface.

8.1 The kernel

While the kernel is currently physically implemented as a single process, it maintains the abstraction of distributed operation and can be easily split up should our platform become physically distributed.

The kernel is executed as a continuous loop; every iteration, it checks an event list, selects the next event, and performs the appropriate action. Events include: scheduling a thread, executing a call to a method, sending a message to a remote object (making a call to a method of an object currently residing on a different processor), and updating object queues. Every entry in the event table has a time-stamp to determine when the kernel should react to that event, and every object has a queue to serialize access to all methods exported by that object. The order in that queue depends on the scheduling criteria used and the arrival order of messages. A kernel replica is emulated for every processor in the network.

Synchronized clocks are emulated (maintained by the kernel) for the entire network. Should the implementation migrate to a physically-distributed system, we would recommend use of GPS or standard time sources for synchronization, as advocated in [14]. The time is measured in abstract real-time units. All events are stamped with time of occurrence.

The kernel responds to an event by initiating the required activity; for example, by activating thread execution or initiating the execution of methods. Thus, calls (except some calls to local methods or system libraries) are directed as requests or as events to the kernel. The kernel actually makes the call by executing the callee method. This implementation has an implicit problem with the values of *out* parameters at the conclusion of the callee, when the execution of the caller is resumed; moreover, there is no way in this design to remember the old state in case of preemption. We address these problems later in this subsection.

Each emulated kernel replica maintains two sets of queues: object queues and processor queues. The object queue is a general priority-based queue. Access to an object will be serialized using its queue. All requests (calls) to services (methods) provided by this object will be added to its queue. Every processor may host multiple objects. The processor queue contains the highest priority requests from the object queues assigned to that processor. Every loop iteration in the kernel algorithm, the object queues of every processor are checked. If there are any calls (requests) still pending, one will be scheduled to run. The selection of the method to be executed will be based on some real-time scheduling criterion, where for simplicity we currently use Earliest Deadline First. However, any scheduling discipline can be used. The kernel executes the code of that method/thread, which may generate a new set of events. The kernel marks the new events with the correct time-stamp and add them to the event table. On completion of the task, the kernel is updated to reflect the time spent in execution (through use of the time-increment statements at the end of execution blocks); in principle, we could instead use the timing table for a static worst-case estimate of time if communication is costly. In either case, the updated time is used to stamp events produced by the executed method/thread. Message events are channeled through the network simulator (see subsection 8.2).

As mentioned earlier, the kernel makes the calls to callee object methods. This causes three problems. First, the kernel must remember values of *out* parameters of the call and pass them back to the caller, both for local calls, and for remote calls to methods of other objects assigned to different processors. The problem becomes still harder for remote calls that invoke other calls. The second problem is similar, but arises from preemption. The kernel must remember the values of local variables to correctly resume execution. Finally, the kernel must remember the method program counter, in order to determine the next statement to execute after resuming execution, and to keep track of the elapsed time. Nonetheless, these problems, and the transformations used to resolve them, will not affect the simulated behavior of the program.

We start by addressing the third problem. Every method/thread is subdivided into a set of non-preemptable units (submethods/subthreads); every unit then runs to completion without preemption. We believe strongly that robust real-time execution is well-supported by schedulers which preempt on the basis of major inter-object control transfers in the application program. For CRL (and many other languages), this implies that preemption should typically occur at largely at calls (or returns), and the criteria we use for determining preemption points are based on calls. In general, we subdivide into two methods/threads whenever we find a call. The first part ends at the call, while the second part resumes with the following statement (that is, with the return). We will further subdivide the second part if we find another call, and so on. As usual with such approaches, this scheme may be modified in some cases, as in the presence of sequences of data-independent calls, or for long blocks of (typically array-based) computation. We discuss how the kernel will handle the execution of these units later in this section.

To overcome the second problem, we change the scope of the declaration of local variables defined within a method to be the scope of the object (assuming all recursive calls are unwound). In other words, local variables for any method will become part of the object internal state. Variables are to be renamed, e.g., by using method name as a prefix, so that no two methods assign a common name incompatibly. Thus, in the case of local calls, the kernel does not have to worry about *out* parameters, as every variable (including the parameters) are part of the object state and can be seen by other methods in the object; the compiler restricts accesses to those legal under the original semantics. This will also hold for those submethods generated by inserting preemption points, as just discussed. *Figure* 2 shows the change in code due to insertion of preemption and changing the scopes of local declarations.

For external calls, the solution is quite different, as the caller and callee do not share state. We instead use a store-and-forward mechanism, as in [41], to remember the parameters of the previous call.[2]

[2]Actually, the CRL situation is easier to address than that

```
ORIGINAL                          TRANSFORMED

Object O1                         Object O1
  var v1,                             var v1,
  var v2                              var v2
                                    private:
  method m1                           var m1_mv1,
     var mv1,                         var m1_mv2
     var mv2,                       method m1_1
      .                                    .
      .                                    .
     call O2.m1()                     call O2.m1()
      .                             endmethod m1_1
      .                             method m1_2
      .                                    .
      .                                    .
     call O3.m5()                     call O3.m5()
      .                             endmethod m1_2
      .                             method m1_3
      .                                    .
      .                                    .
      .                                    .
  endmethod m1                      endmethod m1_3
```

Figure 2: Example of the insertion of preemption points.

For example, if the first call makes another external call, we need to retrieve the values of the parameters of the first call in order to resume execution after returning from the second call. In store-and-forward, we usually pass the values of input parameters of the caller in addition to the parameters required by the callee. Thus, calls to methods have a variable-length list of parameters. Whenever an external call is found within a method/thread, code must be added to store those parameters. All methods and threads will use a standard parameter list consisting of two stacks. The first stack has the parameters of the call. Statements will be added to the code of the method/thread to pop the parameters from that stack. All the parameters of that call will be pushed again onto the stack at preemption points (when making calls) so that they can be retrieved when the call returns. The second stack contains the source object and the next submethod to be executed. The kernel will pop that stack when a call returns to determine which object made that call.

Every call in the program will be replaced by a call to the function store-and-forward. The parameters needed for store-and-forward are: *source id* (where to return), *target id* (which method to call), and *the actual parameters of the caller.* The post-processor replaces external calls with a return statement. The id's referred to above can be addresses, or object id and method name. For example: if $O1.m1\_1$ calls $O2.m3$ then the source id will be the address of $O1.m1\_2$,

```
ORIGINAL                      TRANSFORMED

Object O1                     Object O1
  var v1,                         var v1,
  var v2                          var v2
                                private:
  method m1                       var m1_mv1,
    var mv1,                      var m1_mv2,
    var mv2,                    method m1_1
     .                                .
     .                                .
    call O2.m3(..)                store_and_forward(
     .                                   O2.m3_1,..)
     .                            return
     .                          endmethod m1_1
  endmethod m1                  method m1_2
                                      .
                                      .
                                endmethod m1_2
```

Figure 3: Example of the application of the store-and-forward mechanism.

while the target id will be the address of $O2.m3\_1$. *Figure* 3 shows an example of the code transformations performed by the post-processor to support store-and-forward: the external call has been replaced with a store-and-forward request to the kernel, and the method returns. Later, the kernel will send a message to the target object and resume execution at $m1\_3$ upon the return from the call to $O2.m3$.

As the motivation for this transformation of the code is to enable the implementation of the run-time kernel, we decided to implement them by a post-processor of the intermediate code just before integrating the code with the linker. The input to the post-processor is (restricted, annotated) C++ code; the output will also be annotated C++.

The kernel interacts with the other subcomponents of the run-time environment, as shown in *Figure* 1. First, it calls the network simulation routine to calculate communication delays through the network when invoking an external call. In addition, the kernel measures the execution time of various threads and methods and reports that to the user along with other statistics through the user interface module, as discussed in *Subsection* 8.3. A local time variable is added to every method/submethod. A statement at the end of every basic block is added to increase that variable by the time of the block, as computed by the timing tool. *Figure* 4 shows the C++ translation of a CRL method *update_status*. The value of time is pushed onto the stack, and the kernel will pop it to determine the execution time of that method (not including any communication delays). The purpose of providing measures of execution time at run-time is to correlate the basic timing measures based on the timing map with actual execution. For some operations, the amount of time is an integer constant, while for others it is expressed as a parameterized expression. In prac-

of general C, C++, Lisp and other languages, as discussed in SUPRA-RPC [41], as CRL does not allow out-of-scope references, so that only explicit parameters (or explicitly-generated auxiliary variables) need to be supported in this situation.

```
int update_status_1(System_Stack *sp ) {
   long time = 0;
   cin >>  x ;
   time += 3 ;
   cin >>  y ;
   time += 3 ;
   cin >>  theta ;
   time += 3 ;
   // CALL  to vel.get(theta,speed)
   sp->Param_Stack.pushPointer((void*) &theta);
   sp->Param_Stack.pushPointer((void*) &speed);
   sp->Obj_Stack.pushPointer((void*) this);
   sp->Obj_Stack.pushPointer((void*) &update_status_2);
   sp->Obj_Stack.pushPointer((void*) &vel);
   sp->Obj_Stack.pushPointer((void*) &vel.get_1);
   sp->Param_Stack.pushLong(time);
   store_forward(self.id, "navigation.update_status_2",
                 vel.id, "vel.get_1", sp->no);
   return(1);
}
```

Figure 4: An example of the final code to be linked with the kernel.

tice, some of these parameterized expressions depend only on compile-time knowable information such as operand list length, or iteration and time constraint requirements, and are thus easily resolved and specialized into constants statically at link or elaboration time. However, timing expressions may also depend on the distribution of operands and objects and processes across the network (this is relevant in calls), and on the usage of shared resources.

Once again, the reader is asked to note that while our physical implementation is single process, the kernel fully supports distribution in the application. Naturally, there would be some differences in the actual implementation, but these would be rather mundane and well-understood. For instance, in a true distributed implementation we would need to extend store-and-forward elements with full-fledged stubs, which would (un)marshal and convert call- and return-parameters — a problem overcome in the late Eighties-early Nineties.

8.2 The network architecture simulation tool

The network simulation tool provides the timing delay that thread execution anticipates due to distributed allocation of objects. The simulator uses architectural information including a description of the network topology, various distances between nodes, and the transmission medium, as provided in architecture description file.

Initially, the simulator reads an assignment file generated by the assignment tool, providing a mapping for every object to a processor. Interaction with the kernel is in the form of requests providing the source object and the target object as well as the size of the message to be sent. The simulator consults the object map, and determines the source processor and the target processor. Using the topology description, it then finds the appropriate route along which to transfer the request.

There is a message queue in every node maintained by the network simulator. If a message is to be transferred on a busy link, it will be queued until the link is free. The transmission rate will be dependent on the medium and the distance the message has to travel. The simulator consults some internal table (data sheet) to calculate the transmission time over that line. The kernel will not block waiting for the results of that request. The results of that call are reported back using the same message format but the previous target object becomes a source for the return, and conversely. The total communication delay time is the sum of the transmission times and the communication queuing time (forward for the request and backward for the results). The total service time for the kernel request is the sum of the communication delay and the execution time of the specified method within the target object, plus the object queuing delay, as illustrated in the previous section.

There is no interaction between the network simulator and the user interface in the current implementation. All results and status reported to the user come only from the kernel. In the future, we may provide a graph to show the current status of the network, including communication queues and bottlenecks. In the next section, we describe the user interface subcomponent of the run-time environment.

8.3 User interface

In the current implementation, the user interface is used only to display measurements and statistics on the applicability of transformations and their effects on performance, deadlines, and processor utilization. Development of a graphical interface is work in progress. It eventually will be possible to draw execution progress figures, providing the user with information on what every processor is doing. Moreover, the measurements and statistics mentioned above will also be presented using graphs. In the future, we may extend these capabilities to include a facility for affecting the execution behavior and for providing run-time assertions.

9 Status

The CRL support environment is currently under development in the Real-Time Computing Lab at NJIT. The compiler, linker and the runtime are largely operational, though we are in the process of providing a generalized symbol table and general timing constraint support. A number of transformations, such as speculative execution, have been supported and work is on the way to support more. Basic timing and schedulability analysis tools are in place. As already stated, the work on the assignment tool for CRL is in its early stages, though there are other assignment and allocation tools in operation (developed for other Lab projects). The tools can be demonstrated, with care, to interested parties. We anticipate being able to distribute the tools sometime in 1998 or earlier.

10 Acknowledgements

We would like to thank Ananth Ganesh, Robert Kates, Jeff Venetta and many *cis611 (Real-Time Systems)* students for their contributions to the CRL environment and tools. We are indebted to the Office of Naval Research and, recently, the National Science Foundation for providing generous support for this project. All members of the Real-Time Computing Lab at NJIT have contributed to productive and fruitful discussions, related to the continuous research on this project. Last but not least we thank the four anonymous ICECCS'97 referees for their most useful comments.

References

[1] *Ada 9X Integrated Language Specification* , Ada 9X Mapping Team, Intermetrics, Inc., Cambridge, Massachusetts 02138 (22 December 1992).

[2] A. Aho, R. Sethi and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachussetts, 1986. ISBN 0-201-10088-6.

[3] T. Baker and G. Scallon, "An Architecture for Real-time Software Systems," *IEEE Software*, May 1986, 50-59; reprinted in tutorial *Hard Real-Time Systems*, IEEE Press (1988).

[4] T. Baker and A. Shaw, "The Cyclic Executive Model and Ada," *The Real-Time Systems Journal 1,1* (June 1989) 7-26.

[5] R. Chapman, A. Wellings, A. Burns, "Integrated Program Proof and Worst-Case Timing Analysis of SPARK Ada," In *Proceedings of the Workshop on Language, Compiler, and Tool Support for Real-Time Systems,* June 1994.

[6] G. Chroust, "Orthogonal Extensions in Microprogrammed Multiprocessor Systems - A Chance for Increased Firmware Usage," *EUROMICRO Journal*, Vol. 6, No. 2, pp. 104–110, 1980.

[7] T. Chung, "CHaRTS: Compiler for Hard Real-time Systems," *PhD Thesis Proposal,* Purdue University, April 1994.

[8] R. Gerber and S. Hong, "Compiling Real-Time Programs with Timing Constraints Refinement and Structural Code Motion," *IEEE Transactions on Software Engineering*, Vol. 21, No. 5, May 1995.

[9] R. Gupta, M. Spezialetti, "Busy-Idle Profiles and Compact Task Graphs: Compile-Time Support for Interleaved and Overlapped Scheduling of Real-Time Tasks," University of Pittsburgh Technical Report TR-94-24, April 1994.

[10] V. Haase, "Real Time Behavior of Programs," *IEEE Transactions on Software Engineering,* SE-5 (7):494–501, September 1981.

[11] W. Halang, "A Proposal for Extensions of PEARL to Facilitate Formulation of Hard Real-Time Applications," *Informatik-Fachberichte 86,* 573–582, Springer-Verlag, September 1984.

[12] W. Halang, "On Methods for Direct Memory Access Without Cycle Stealing," *Microprocessing and Microprogramming*, 17, 5, May 1986.

[13] W. Halang, "Implications on Suitable Multiprocessor Structures and Virtual Storage Management when Applying a Feasible Scheduling Algorithm, in Hard Real-Time Environments," *Software - Practice and Experience*, 16(8), 761-769, 1986.

[14] W. Halang and A. Stoyenko, *Constructing Predictable Real-Time Systems*, Kluwer Academic Publishers, Dordrecht-Hingham, 1991.

[15] M. Harmon, T. Baker, D. Whalley, "A Retargetable Technique for Predicting Execution Time," *Proceedings of the IEEE Real-Time Systems Symposium*, IEEE (December 1992).

[16] Y. Ishikawa, H. Tokuda, C. Mercer, *Object-Oriented Real-Time Language Design: Constructs for Timing Constraints*, Department of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-90-111, March 1990.

[17] *KE-Handbuch*, Periphere Computer Systeme GmbH, Munich, 1981.

[18] R. Kieburtz, J. Hennessy, "TOMAL – A High-Level Programming Language for Microprocessor Process Control Applications," *ACM SIGPLAN Notices,* Vol. 11, No. 4, April 1976, pp. 127-134.

[19] E. Kligerman and A. Stoyenko, "Real-Time Euclid: A Language for Reliable Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9, pp. 940–949, September 1986.

[20] D. Leinbaugh, "Guaranteed Response Times in a Hard-Real-Time Environment," *IEEE Transactions on Software Engineering,* SE-6 (1):85–91, January 1980.

[21] D. Leinbaugh and M.-Yamini, "Guaranteed Response Times in a Distributed Hard-Real-Time Environment," In *Proceedings of the IEEE 1982 Real-Time Systems Symposium,* 157–169, December 1982.

[22] D. Leinbaugh and M.-Yamini, "Guaranteed Response Times in a Distributed Hard-Real-Time Environment," *IEEE Transactions on Software Engineering,* SE-12 (12):1139–1144, December 1986.

[23] K.-Lin, S. Natarajan, "Expressing and Maintaining Timing Constraints in FLEX," *Proceedings of the IEEE 1988 Real-Time Systems Symposium*, December 1988.

[24] A. Mok, P. Amerasinghe, M. Chen, K. Tantisirivat, "Evaluating Tight Execution Time Bounds of Programs by Annotations," *IEEE Workshop on Real-Time Operating Systems and Software,* Pittsburgh, PA, pp. 74–80, May 1989.

[25] D. Niehaus, "Program Representation and Translation for Predictable Real-Time Systems," *IEEE Real-Time Systems Symposium,* pp. 53–63, San Antonio, Texas, December 1991.

[26] K. Nilsen, *Issues in the Design and Implementation of Real-Time Java*, Iowa State University, Ames, Iowa, 1995.

[27] V. Nirkhe, W. Pugh, "A Partial Evaluator for the Maruti Hard Real-Time System," *Real-Time Systems,* Vol. 5, No. 1, pp. 13–30, March 1993.

[28] C. Park, "Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths," *Real-Time Systems,* Vol. 5, No. 1, pp. 31–62, March 1993.

[29] C. Park, A. Shaw, "Experiments with a Program Timing Tool Based on a Source-Level Timing Schema," *IEEE Real-Time Systems Symposium,* Orlando, FL, December 1990.

[30] P. Puschner, C. Koza, "Calculating the Maximum Execution Time of Real-Time Programs," *International Journal of Time-Critical Computing Systems*, Volume 1, Number 2, pp. 159-176, September 1989.

[31] Rate Monotonic Analysis for Real-Time Systems Project, *Handbook of Real-Time Systems Analysis*, Software Engineering Institute, Carnegie-Mellon University, May 1992.

[32] K. Schleisiek-Kern, *Private Communication*, DELTA t, Hamburg, 1990.

[33] G. Schrott, *Ein Zuteilungsmodell fuer Multiprozessor-Echtzeitsysteme*, PhD Thesis, Technical University, Munich 1986.

[34] L. Sha, and J. Goodenough, "Real-Time Scheduling Theory and Ada," *Computer 23,4*, IEEE (April 1990) 53-62.

[35] M. Shaw, "A Formal System for Specifying, Verifying Program Performance," Carnegie-Mellon University, Computer Science Department, Technical Report CMU-CS-79-129, June 1979.

[36] A. Shaw, "Reasoning About Time in Higher-Level Language Software," *IEEE Transactions on Software Engineering,* pp. 875–889, SE-15, No. 7, July 1989.

[37] A. Shaw, "Deterministic Timing Schemata for Parallel Programs," University of Washington, Department of Computer Science and Engineering, Technical Report 89-05-06, May 1990.

[38] A. Stoyenko, *Turing goes Real-Time...*, Internal Programming Languages Report, Department of Computer Science, University of Toronto, May 1984.

[39] A. Stoyenko, "A Schedulability Analyzer for Real-Time Euclid," *Proceedings of the IEEE 1987 Real-Time Systems Symposium*, pp. 218-225, December 1987.

[40] A. Stoyenko, *A Real-Time Language with A Schedulability Analyzer,* Ph.D. Thesis, Department of Computer Science, University of Toronto, 1987.

[41] A. Stoyenko, "SUPRA-RPC: SUbprogram PaRAmeters in Remote Procedure Calls," *Software-Practice and Experience*, Vol. 24, No. 1, pp. 27–49, January 1994.

[42] A. Stoyenko, W. Halang, "High-Integrity PEARL: A Language for Industrial Real-Time Applications," *IEEE Software*, July 1993

[43] A. Stoyenko, V. Hamacher, R. Holt, "Analyzing Hard-Real-Time Programs for Guaranteed Schedulability," *IEEE Transactions on Software Engineering,* pp. 737–750, SE-17, No. 8, August 1991.

[44] A. Stoyenko, T. Marlowe, "Schedulability, Program Transformations and Real-Time Programming," *IEEE/IFAC Real-Time Operating Systems Workshop,* May 1991, Atlanta, Georgia.

[45] A. Stoyenko and T. Marlowe, "Polynomial-Time Transformations and Schedulability Analysis of Parallel Real-Time Programs with Restricted Resource Contention," *Journal of Real-Time Systems*, Vol 4, No. 4, pp. 307–329, November 1992.

[46] A. Stoyenko, T. Marlowe, W. Halang and M. Younis, "Enabling Efficient Schedulability Analysis through Conditional Linking and Program Transformations," *Control Engineering Practice*, Vol 1, No. 1, pp. 85–105, January 1993.

[47] A. Stoyenko, T. Marlowe and M. Younis, "A Language for Complex Real-Time Systems," *The Computer Journal*, Vol. 38, No. 4, pp. 319–338, November 1995.

[48] T. Tempelmeier, "A Supplementary Processor for Operating System Functions," *1979 IFAC/IFIP Workshop on Real-Time Programming*, Smolenice, 18-20 June 1979.

[49] T. Tempelmeier, "Operating System Processors in Real-Time Systems - Performance Analysis and Measurement", *Computer Performance*, Vol. 5, No. 2, 121-127, June 1984.

[50] United States Department of Defense, Ada Joint Program Office. *Reference Manual for the Ada Programming Language* ANSI/MIL-STD-1815A-1983 (February 1983).

[51] M. Younis, T. Marlowe and A. Stoyenko, "Compiler Transformations for Speculative Execution in a Real-Time System," *Proceedings of the* 15*th Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

[52] M. Younis, T. Marlowe, G. Tsai, A. Stoyenko, "Applying Compiler Optimization in Distributed Real-Time Systems," *Technical Report CIS-95-15*, Department of Computer and Information Science, New Jersey Institute of Technology, 1995.

[53] K. Zuse, *Foreword* to Wolfgang A. Halang, Alexander D. Stoyenko, *Constructing Predictable Real-Time Systems*, Kluwer Academic Publishers, Dordrecht-Hingham, 1991.